Development and Implementation of a RAS Framework Prototype for HPC Environments

Realtime Data Reduction of Monitoring Data

A Dissertation
Submitted In Partial Fulfillment Of
The Requirements For The Degree

MASTER OF SCIENCE

In

NETWORK CENTERED COMPUTING,
High Performance Computing

in the

FACULTY OF SCIENCE
UNIVERSITY OF READING
by

Swen Böhm

03/30/2010

University Supervisor: Prof. Dr. Vassil Alexandrov
Placement Supervisor: Dr. Christian Engelmann

Acknowledgments

I will use this place to express my gratefulness to all the people who helped me to make all of this happen. At first there are my parents, without their help I would never have come so far. Then I will say thank you to all my colleagues at Computer and Mathematics Division at the Oak Ridge National Laboratory, especially to Dr. Christian Engelmann. They all welcomed me very friendly at the lab and provided all the help I needed to proceed with my work. Thanks to Teresa Finchum, George Bosilca and Wendy Syer from the University of Tennessee, Knoxville for their support and help. I will also thank Prof. Vassil Alexandrov and Prof. Vesselin Iossifov whitout whom nothing of this would ever had happened, Prof. Johan Schmidek and Prof. Dieter Kranzlmüller. Then there are some very good friends I have to thank as well for their kindness and their time to listen. Jörg Manitz, Manuela Lewin, Kjell Haustein, Tiaan Tromp, Gunter Kanitz, Jan Balzereit, Ingo Buse, Anja and Mathias Walzer, my brother Dirk and my cousins Christoph and Carsten Rätsch.

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

Statement of Authorship

I hereby certify that this thesis is my own work. I did not get any unlawful assistance of someone else. The use of all material and sources has been fully acknowledged. The work was not submitted previously in the same or similar form to another examination committee and was not yet published.

Swen Böhm

Abstract

The advancements of high-performance computing (HPC) systems in the last decades lead to more and more complex systems containing thousands or tens-of-thousands computing systems that are working together. While the computational performance of these systems increased dramatically in the last years the input/output (I/O) subsystems have not gained such a significant improvement. With increasing numbers of hardware components in the next generation HPC systems maintaining the reliability of such systems becomes more and more difficult since the probability of hardware failures is increasing with the number of components. The capacities of traditional reactive fault tolerance technologies are exceeded by the development of next generation systems and alternatives have to be found. This paper discusses a monitoring system that is using data reduction techniques to decrease the amount of the collected data. The system is part of a proactive fault tolerance system that may solve the reliability problems of exascale HPC systems.

Contents

List of Figures

Listings

Acronyms

AMTTF application mean-time to failure. 5, 6

API application programming interface. 2, 3, 16, 19

BE back-end. 8, 11–16, 20, 29, 31, 54

BLOB binary large object. 44

CPU central processing unit. 1

FE front-end. 8, 9, 11–16, 19, 20, 31

FT fault tolerance. 2, 7, 11

GPGPU general-purpose computing on graphics processing units. 2

HPC high-performance computing. iv, 1–3, 5, 7, 10, 11, 19

I/O input/output. iv

IC intermediate child. 9, 11–13, 15, 16, 39, 54

ID identifier. 14

MPI Message Passing Interface. 17

MRNet multicast/reduction network. 8, 9, 11–21, 23, 25, 27–29, 32, 33, 42, 51, 52, 55, 63

NRPE Nagios remote plugin executor. 4

NSCA Nagios service check acceptor. 4

OpenIPMI Open Intelligent Platform Management Interface. 7, 10, 14

ORNL Oak Ridge National Laboratory. 7, 24, 54

OS operating system. 2, 9, 14

PFT proactive fault tolerance. 5

RAS reliability, availability and serviceability. 5, 7, 9

RRD round robin database. 10, 11

SNMP Simple Network Management Protocol. 4

SQL structured query language. 4

STL Standard Template Library. 40, 41

TBŌN tree based overlay network. 8, 9, 11–16, 18, 20–27, 29, 30, 32, 39, 40, 42, 50, 54, 55, 61, 64

ToM TAUoverMRNet. 8

XDR external data representation. 4

XML extensible markup language. 4

1 Introduction

The microchip is one of the inventions of the last century that has changed many aspects of the modern world. Microchips are used in nearly every electronic device and are present in nearly every household of the modern world. They are found in entertainment and control systems, cell phones and hand held devices, in engines, and of course in systems that are called computer.

The use and appearance of a computer has dramatically changed over the last decades. In the early beginnings, they were large-room-sized machines, used by the military and the scientific community. With the rapid development they soon reached the economy as well. The invention of the micro circuit has increased the performance of the computer and decreased the size in the same way. Additionally the power consumption and their price decreased and therefore the computer entered the homes. Today computers are the foundation for many companies that either produce hard- or software or provide services to a vast amount of customers.

While the average user benefited from these inventions and most of todays computers provide more power than actually is needed, advanced users are always in need for more performance. If the advanced user is a gamer, who is in need of a fast graphic card and central processing unit (CPU), or an employee who is using the computer in commercial applications, such as computer aided design or computer aided engineering. But the most performant computers of their time were always used by the scientific community or the military for research. Many improvements to the computer were driven by these communities.

The need for performance is the nature of the applications running in most of these research or development projects. The most time consuming applications that are running on the fastest computers of the world are either some kind of simulation or are used to evaluate huge amounts of data.

During the last decades improving the performance of computers was mainly achieved by increasing the clock rate, shrinking structures and improvements of the architecture of the CPU. From scalar over super scalar and vector to today's many core processors. Although the performance gain of the CPU is remarkable, it was not sufficient to satisfy the needs of the operators of such HPC systems. So the development in the HPC community started to find other ways to improve performance. In the 1980's, the systems with the highest performance were vector

computers which could issue one instruction to a sequence of data. And in the 1990's, parallel computers were taking the place of the vector machines. These systems are constellations of many single computers working together to solve a problem and the current approach to satisfy the performance needed for today's computational problems.

Actual petascale HPC systems consists of thousands or tens-of-thousands of compute nodes and upcoming exascale HPC systems are expected to scale to hundreds of thousand nodes which will use multicore processors and additional general-purpose computing on graphics processing units (GPGPU). The overall amount of hardware components in current systems exceeds a million of components and the next generation systems are expected to have ten to hundred times more.

With the increasing amount of components in HPC their reliability is decreasing. The vast amount of hardware components, whose reliability has not changed significantly in the last years, leads to an increasing failure rate.

The current way to deal with the reliability problems and to achieve fault tolerance (FT) in HPC applications is to create checkpoints and restart the applications from the last checkpoint if a failure occurs. An application a creates checkpoint of its current state, either using application specific approaches or mechanisms of the underlying application programming interfaces (APIs) or the operating system (OS), and writes it to a file system. Failures in the application or hardware that causes the application to hang on one node or a node to crash, can cause the entire application to fail. If this happens an administrator or a job submission system has to restart the job beginning from the last checkpoint.

To meet the challenges of future HPC systems a new approach to increase FT has to be found.

1.1 State of the Art

1.1.1 Reactive Fault Tolerance

Todays FT mechanisms for HPC applications rely on checkpointing and restart. A checkpoint can be a copy of the processes memory and registers that is stored to a storage device. In the case of an error the last checkpoint is used to recover the process and to continue its execution from this point.

The checkpointing functions have either to be implemented by the application developer or can rely on mechanisms provided by the job submission system or the underlying OS. For an application level implementation of an checkpointing mechanism the application developers can implement it by themselves or by utilizing

mechanisms provided by an API for parallel applications.

Today's most common checkpointing mechanisms take coordinated checkpoints. All processes on all nodes create a checkpoint to the same time.

Checkpointing of petascale applications produces an enormous I/O load on the network and a huge amount of data on the storage. A HPC system with 10,000 nodes and an average memory usage of 2GB would produce an amount of 20,000 GB that has to be stored. Most HPC systems use a networked file system and the transport of this vast amount of data can exceed the capabilities of the I/O subsystem [6, 24], whose performance has not increased significantly during the last years. With exascale HPC systems on the horizon that are going to be composed of much more nodes, it is inevitable that this mechanisms have to be improved dramatically or replaced by other mechanisms.

There are currently many research projects with the goal to improve checkpoint/restart mechanisms used. Improvements to checkpointing can be done by taking asynchronous or coordinated [4] checkpoints, by optimizing checkpointing intervals [5] or with incremental checkpoints [9]. Asynchronous intervals will reduce the current bandwidth needed to save a checkpoint. The different nodes take their checkpoints at different times and the file system utilization is distributed over time. Asynchronous checkpointing however will not reduce the amount of the data that has to be stored. Reducing the interval of the checkpoints on the other hand will produce a smaller amount of transported and stored data.

1.1.2 System Health Monitoring

To monitor the health of computational systems many software solutions are available. There are many popular open source monitoring systems available, and many hardware vendors offer monitoring solutions for their systems. In this section, two open source systems will be described in detail and their advantages and disadvantages will be discussed.

Ganglia

Ganglia [16, 19, 20] is a set of system daemon processes and tools dedicated to monitor HPC systems, such as clusters and Grids. The Ganglia monitoring system was started as the Millennium Project at the University of California, Berkeley. The system is based on a hierarchical design to scale to large installations. Ganglia is a set of daemons and tools that work together as a monitoring system. The monitoring data is captured by the (`gmond`) daemon. It collects system metrics, like CPU,

memory, disk, network, and process data. Additional metrics can be added using the `gmetric` tool, which adds a metric value to a local or remote `gmond` service. The monitoring daemons can be aggregated to groups which exchange their data using the external data representation (XDR) protocol. The `gmond` service can be queried on port 8679 (if not configured otherwise) and sends the data back in an extensible markup language (XML) format (see Listing D.1 for an example).

To get a hierarchical structure, a second daemon `gmetad` collects the data from the aggregated `gmond` groups. To achieve fault tolerance `gmetad` can query every `gmond` in a group. A `gmetad` daemon itself can again be queried by another `gmetad` daemon and form a tree structure to monitor big installations.

The representation of the monitoring data as XML dataset makes it easy to write software to post process and validate the monitoring data. Another benefit is that the data format is human readable as it is. The disadvantage of the XML output is that it produces much more overhead. The XML data needs more bandwidth for the transportation through a network and more space on the storage than a binary format for example.

Nagios

Nagios uses a Web front end to represent the states of the monitored hardware but the monitoring data can be dumped into a structured query language (SQL) database which produces much less overhead than the XML format Ganglia uses. Nagios uses a monitoring daemon that queries the nodes that have to be monitored. It differentiates between public and private services that have to be monitored with different methods. Public services are daemons that can be monitored through a network connection, like Web, database and print servers for example. Private services (in the Nagios notation CPU usage, loads and most other types of health monitoring relevant metrics) can be gathered using different methods. On one side there is the possibility to query Nagios remote plugin executor (NRPE) (a separate daemon that runs on the monitored host(s)) or a Simple Network Management Protocol (SNMP) daemon. On the other side, Nagios can use passive checks, were the monitored node can announce data using Nagios service check acceptor (NSCA). Using passive checks the monitored host can run scripts periodically (using `cron` or `at` on UNIX machines) and announce the results to the monitoring host with the `nsca_send` tool. NRPE enables the monitor host to execute Nagios plug-ins remotely. Both, NSCA and NRPE are Nagios add-ons that have to be installed separately.

1.2 Related Work

1.2.1 Proactive Fault Tolerance

A new approach to increase the reliability of HPC systems is proactive fault tolerance (PFT) [7, 21, 30]. It is an emerging technology that prevents the impact of compute node failures to a parallel application. A proactive reliability, availability and serviceability (RAS) framework [30, 32] (see Figure 1.1) increases the application mean-time to failure (AMTTF) of HPC systems by migrating application parts away from nodes that are "about to fail".

To predict [8, 31] failures that can cause a compute node or an application to fail, a proactive RAS system has to know the health state of all hosts. The health of all compute nodes is constantly monitored and the data is analyzed in a constant feedback loop (see Figure 1.2). If it is likely that a host is about to fail preventative action is taken to migrate application parts away from nodes that are going to fail. To move application parts a preemptive RAS system has to incorporate the resource manager [15] or the runtime environment [33].

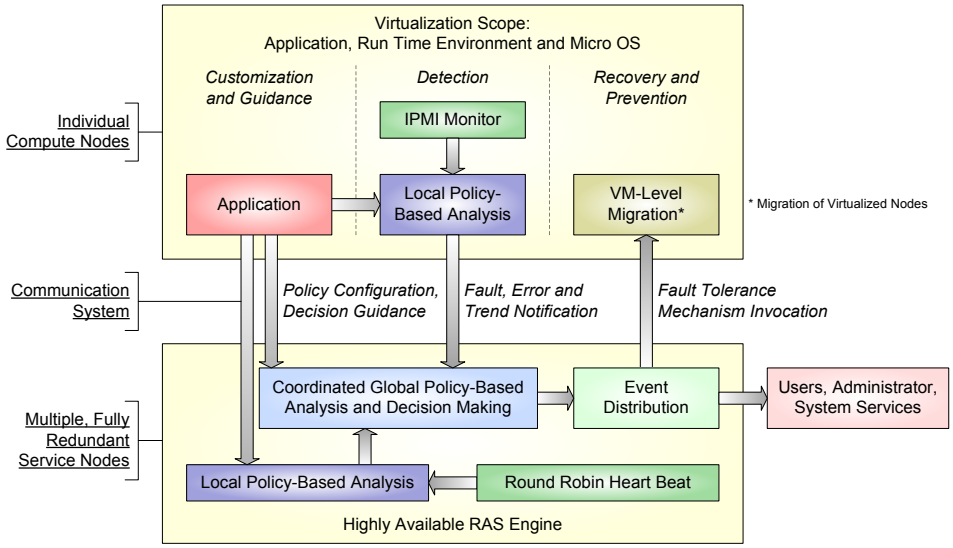


Figure 1.1: Previously proposed Proactive RAS framework ([30])

The feedback loop can be classified in 4 types (see figure 1.2.1) with different capabilities. Type I (see figure 1.3(a)) is the most basic form and provides coverage for the most basic failures. The migration is triggered by the events generated by the monitoring software on the compute nodes when a threshold is exceeded. It is prone to false negatives and positives due to the lack of data correlation abilities. Type II (see figure 1.3(b)) is an enhanced form of Type I. A filter on each node is

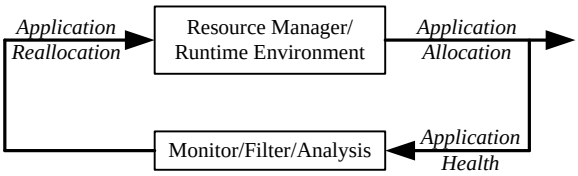
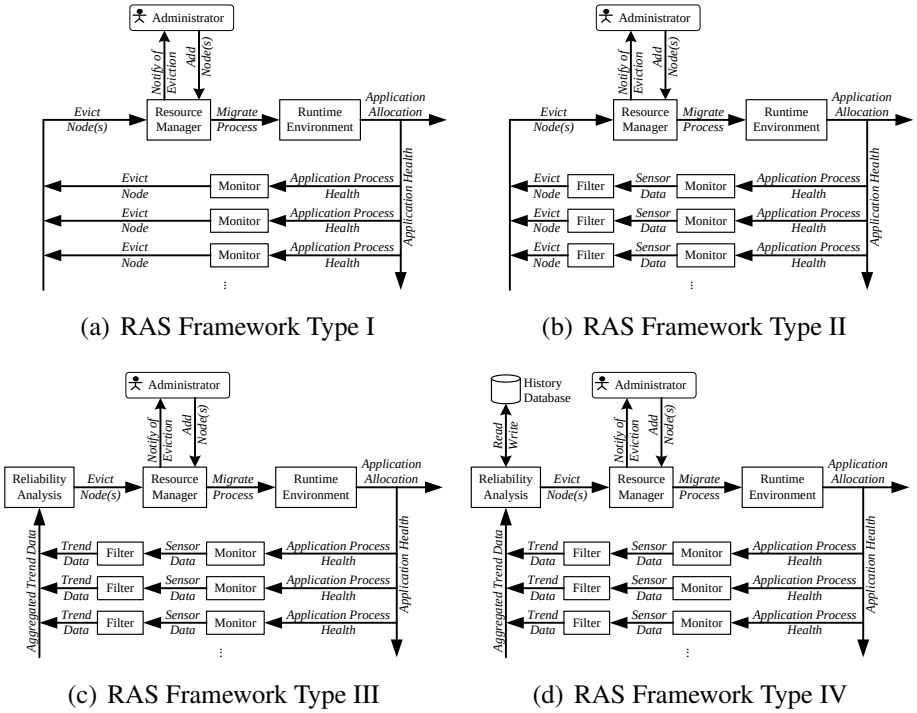


Figure 1.2: Control mechanism of proactive FT with preemptive migration using a feedback loop [7].

able to analyze the monitoring data over a short period of time. Type III enhances Types I & II by incorporating a reliability analysis. In Type IV a history database is used by the reliability analysis to record the monitoring data and to use it for the analysis.

To decrease AMTTF the RAS Framework is moving application parts away from a node that is going to fail. The application part is moved to another healthy node, which can either be a spare node or another node that is already running another part of the application. While the node itself fails, the distributed application sustains its work and is not interrupted.



(a) RAS Framework Type I

(b) RAS Framework Type II

(c) RAS Framework Type III

(d) RAS Framework Type IV

Figure 1.3: The different types of the RAS Framework (source [7])

It is obvious that it takes time to migrate an application part from one compute node to another. The time window to migrate the application varies [32, 33] and

depends on the used methods and technologies and the memory used by the application. Therefore the failure has to be predicted in advance to the failure, such there is enough time to successfully migrate the application part away from the failing node.

To predict failures, it is necessary to know the health state of the system. Constant monitoring of the system can provide the necessary information. The constant observation of the health state of a hundred of thousands of compute nodes is a challenge, since the monitoring produces a vast amount of data that needs to be processed. To achieve an accurate reaction time for the prediction, the monitoring data has to be processed nearly in real time.

Since current HPC systems have 1,000 to 20,000 nodes and up to than 200,000 cores, and are continuing to increase in scale, system monitoring and logging produces an increasing amount of data. The XTORC cluster (a small 64 node system in the Computer Science and Mathematics Division at the Oak Ridge National Laboratory (ORNL)) produces approximately 33 MB/h with a sampling interval of 30s [15]. The Jaguar [22, 23] system at ORNL with 18,772 compute nodes would produce a monitoring stream of almost 10 GB/h with a sampling interval of 30s and 30 GB/h with a sampling interval of 10s assuming that the monitored metrics are the same. To achieve the necessary reaction time, a smaller capture interval will be necessary and will increase the amount of data even further.

Since it is not feasible to process these vast amount of data in a proactive FT system and to store it in a history database [7] for a reliability analysis, the amount of data must be reduced while it is generated.

1.2.2 Monitoring and Analysis

Open Intelligent Platform Management Interface (OpenIPMI) [25] and lm-sensors [17] are libraries to access hardware monitoring data, such as processor temperatures and fan speeds. Ganglia [16, 19, 20] (see Section 1.1.2) is a distributed monitoring system that scales well in large installations. Ganglia and OpenIPMI were used in Type 1 RAS solutions [21, 33]. Another RAS framework solution is OVIS 2 [3]. OVIS 2 monitors nodes either directly or collects information from other monitoring solutions. It provides tools for statistical analysis of metric data. OVIS 2 provides Type 3/4 online analysis and Type 4 analysis using a history database is provided for offline data. OVIS 2 has not been used in a proactive FT solution until now. HPC vendors provide their own monitoring solutions (e.g. HP Cluster Management Utility [11], IBM Cluster Systems Management [12]).

1.2.3 Data Reduction

Data reduction is used to transform acquired information into an ordered and simplified form. The data can be sorted, rounded or classified by a set of criteria. Usually data reduction algorithms have to compute large amounts of data that is stored on the file system. Today many forms of data reduction techniques are used for different purposes. Compression algorithms, either lossy or lossless ones, filter and correlation tools to find the relevant data in a dataset, and tools to classify datasets.

The performance monitoring system TAUoverMRNet (ToM) introduced by Nataraj et al. [18] uses data reduction techniques to classify performance data of parallel applications on the fly. ToM utilizes multicast/reduction network (MRNet) [27, 29] to distribute the computation to a number of nodes. MRNet uses a tree based overlay network (TBŌN) [2], a tree based structure of processes, to communicate between a front-end (the root of the tree) and the back-ends (the leaves of the tree). Figure 1.4 depicts the layout of a TBŌN. A set of processes, the nodes of the tree, are used to run computations on the data that is transported through the TBŌN. A TBŌN is a powerful programming model that has proven to scale well in large distributed infrastructures. TBŌNs provide extensible data reduction and synchronization techniques, high throughput and low latency data flow, and a flexible data communication model.

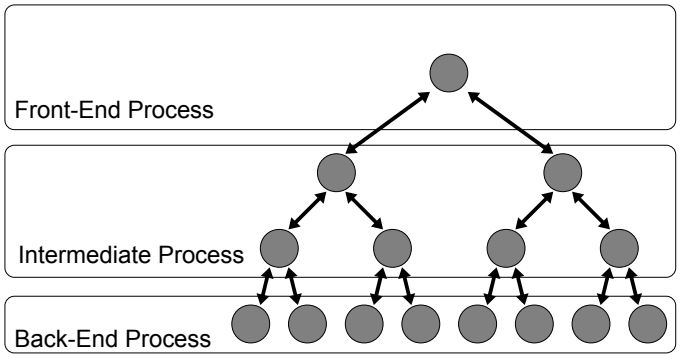


Figure 1.4: A tree-based overlay network. Packets in the TBON flow up and down the logical network through the communication processes (intermediate children).

MRNet is a software library that implements an overlay network. A tree of internal processes between the front-end (FE) and the back-ends (BEs) (see figure 1.4) is used to improve the communication performance. The internal processes (interme-

diate children (IC)) are also used to distribute control commands. The intermediate child (IC) can process the data to keep the FE load manageable. MRNet-based software uses logical channels, called streams, for communication. Filters can be bound to these streams by loading application specific filter functions in the IC. The filters are used to synchronize and aggregate the data that flows through the TBŌN. To reduce the cost of control requests and achieve high-bandwidth communications, MRNet uses multicast messages and a binary and compressed data representation for the communication.

1.3 Objectives

To face the challenges of monitoring large scale computing systems, the goal of the presented work is to develop and implement a prototype of a data reduction monitoring system as part of a proactive RAS framework. The monitoring system addresses the needs of Type III and Type IV of the RAS framework described in Section 1.2.1.

To reduce the data produced by the monitoring system, it will sort the actual metric values into given classes. The classification has to be configurable by defining the intervals for the different classes. Every metric has to have its own configuration. A global configuration has to be used to configure the monitoring system.

To increase flexibility and to address different hard and software architectures, the monitoring system has to provide the possibility to extend the system to capture metric values from new sources.

The monitoring system needs to be portable. It has to be able to run on different system architectures and OSs. As part of a RAS system it is important that the monitoring system will not cause any reliability issues for the RAS system itself. It needs to have a small footprint (low memory and processor utilization) and an effective way to communicate.

2 Preliminary System Design

2.1 Analysis

To gather the health metrics of computer systems, monitoring software systems (like Ganglia, Nagios or vendor software from Cray or IBM) can be used. These monitoring solutions usually provide a set of system daemon processes to capture and distribute monitoring data. They are either complex systems with a huge feature set or solutions for specific systems. Another way to gather system health data, is to use software libraries (libsensors, OpenIPMI) that can query the monitoring capabilities that are integrated into the hardware. These libraries can be used to develop a specialized software that fits the special needs of HPC environments.

Both monitoring solution (Nagios, Ganglia) (see Section 1.1.2) are freely available as open source software and deployed in many facilities. Both systems are using a top down approach where the monitoring node queries the monitored systems in periodical intervals. In both systems, monitoring data is transported without any data reduction and is not stored by default. Nagios has a database support and the `gmetad` daemons store the data in a round robin database (RRD) . To permanently store Ganglia data, it has to be fetched out of the RRD and to be stored with a separate solution. But this exceeds the capabilities of current systems, where huge installations have to be monitored. Therefore there is a need to reduce the data before it is stored.

While system monitoring data, e.g. temperature, fan speeds, core voltages, and so on are numerical, values which can be classified, system log data is passed as text messages from applications to a system logging daemon. Monitoring data values can be classified and transformed into histograms by sorting the value into a given number of bins inside one of the specified classes.

Since there is currently no experience about the metrics that are necessary for a fault prediction system and how accurate they have to be, there is a need for a configurable system. The intervals in which the metrics will be gathered and the different metric values (such as temperature, fan speeds, power supply) have to be configurable, as well as the ranges of the different bins for a metric have to be adjustable. To provide a system as flexible as possible it has to provide non uniform sizes for the different bins.

The overall impact of the monitoring system to the running applications has to be as low as possible. Therefore the monitoring system has to use as less processor time and memory as possible, and the network utilization has to be kept low as well. To work with different HPC systems, the monitoring system has to be flexible and portable.

Additional features like a RRD to store the accurate metrics for analysis and the reduction of system log data can be implemented. Another additional feature that would be useful to test failure prediction is an interface to inject failure patterns into the system.

2.2 Design

The monitoring system will use a TBŌN to transport and process the monitoring data. As implementation of the TBŌN the monitoring system will utilize the MRNet library [27]. Given the hierarchical structure of a TBŌN and prerequirements of the MRNet library, the monitoring system will consist of three elementary components.

Since the data reduction system is part of a proactive FT system, the most important requirement is that it has to be fault tolerant as well. Failing nodes must not affect the functionality of the FT system. Whereas the MRNet library has support for recovering form IC failures, FE and BE failures have to be handled by the FT system. Although a BE-node failure is critical for the HPC system, it is not critical for the FT system itself. However, it can lead to problems with filter plug-ins, because the filter functions are awaiting packets from all their children. Therefore a mechanism is needed, to remove dead nodes from the TBŌN and reintegrate them when they are up again.

The structure of the TBŌN has to be configurable to meet the requirements of the underlying architecture. To gather system health metrics a system-independent wrapper is needed, so that the software can be ported to different architectures.

First there is the FE daemon. The FE will receive and store the collected and classified monitoring data from the compute nodes. It is also responsible to manage the TBŌN and to configure the monitoring system.

The second part of the system is the BE process. It runs on all compute nodes that are the leaves of the TBŌN. The BE process has the task to gather the monitoring data and to send the data to the FE process. Instead of gathering the monitoring data from a third party software, the BE will capture the monitoring data itself. The advantages of capturing the data directly are, that only one process is involved in the

monitoring and there is no need to parse the output format of a third party software.

Part three is the MRNet filter plug-in. It will be loaded by the ICs of the TBŌN. The filter plug-in collects the data from all its child processes and repacks the monitoring data to send it up to its parent until the FE is reached.

Figure 2.1 depicts the layout of the monitoring system and how the different components correspond to the RAS framework proposed in [7]. The FE, the BE and the ICs are the root, nodes, and leafs of the tree. The FE is part of a redundant RAS system and is executed on the system management node [7]. The BE processes are connected to the front-end process trough the ICs. While the BE and FE are part of the monitoring system, the IC is a program provided by the MRNet library [27]. To execute application specific code the IC can load filter plug-ins. The plug-ins are shared libraries and provide filter functions that can be applied to the data flowing through the TBŌN.

To reduce the produced amount of monitoring data, the actual metric values of a certain capture time will be classified by the back-end. Only the class of the value will be transmitted to the FE and stored into a history database [7]. The transmitted and stored data can be reduced even further by just transferring a class value when it has changed since the last capture interval.

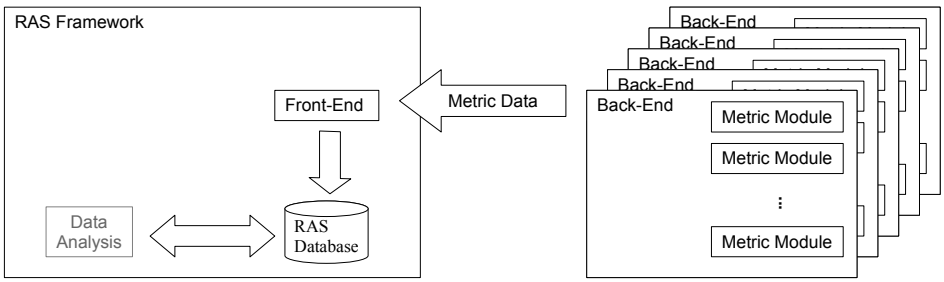


Figure 2.1: The monitoring system as part of a RAS Framework. The Back-End is executed on the compute nodes. It is constantly monitoring the nodes health state and classifying the monitoring data. The data aggregation is performed on the (IC). On the (FE) the data is written into the database.

To achieve the necessary fault tolerance, the front-end has to be highly available. This can be done by having a redundant installation for the FE. The BE is the actual monitor and is executed on all nodes that have to be monitored. In the case that a node is going down (due to a failure or to a scheduled event) it has to be removed from the TBŌN. To discover unscheduled downtimes the front-end needs to heart

beat the BEs. The MRNet library has an build in mechanism to recover from an IC failure, which requires the filter plug-ins to have a separate function to extract filter states.

2.2.1 Front-End Damon

The FE is the head node of the monitoring system. It is responsible for the setup of the TBŌN and to configure and start up the back-end processes. As all monitoring data is send to the front-end, it is responsible to store the monitoring data for the reliability analysis. The font-end will use a MySQL [26] database to store the classified metric data.

To instantiate the TBŌN the FE has to pass a topology file to the network constructor of the MRNet library. The topology file contains a description of layout and the participating hosts in the overlay network. The MRNet project provides a tool to create the configuration file. The tool creates the topology file based on an input file which contains the hostnames of the nodes that are part of the TBŌN.

The MRNet library has two different modes to set up a network. In the first mode the MRNet library creates all internal and back-end processes using a remote shell. If the target system uses a process management system and the back-end processes can not be started directly, the second mode starts only the internal processes and creates a file that contains a list of startup parameters for the back end processes. Then it waits for the BEs to connect autonomously.

After the network is set up, the FE has to establish a configuration stream to send the configuration to the BEs. The configuration of the monitoring system is stored in a second configuration file. It is structured in sections for different aspects of the configuration (e.g. database, metric). To provide the back-end processes with the metric configuration (capture interval, the intervals for the classification) the front-end has to read the metric section of the configuration. After the configuration is passed to the back-end processes, the front-end waits for the back-ends to acknowledge. If the configuration is acknowledged by the back-ends, the front-end sends a message to start the monitoring process.

The monitoring data is received by a thread that listens for incoming packets. The listener thread receives the incoming monitoring data. The data is stored into the history database.

2.2.2 Back-End Damon

BE processes are the most essential part of the monitoring system. A back-end process captures the system and system health metrics of its particular node.

There are different sources available to gather system related or hardware health metrics. System related metrics can be found in the `/proc` directory on the most Unix-type systems. The `/proc` directory is a virtual file system that is dynamically created by the OS kernel during runtime. It provides information to the running processes, the system hardware, and the kernel. Hardware health metrics can be read from integrated sensor chips or special monitoring hardware by using the different available open source libraries (libsensors, OpenIPMI) or vendor specific libraries.

After the startup, either directly by the frond-end, a process management system or an administrator, the back-end connects to the TBŌN provided by the MRNet library. After the network is set up, the back-end waits to receive a configuration message from the front-end. The configuration message includes all the necessary information for the different metrics to collect.

The back-end will use modules to access different kinds of metrics. To reduce the overhead as much as possible, the back-end will only load the modules that are needed. While processing the different sections in the configuration received from the FE, the back-end will load the necessary modules. For each module the BE will register the metrics that have to be captured and assign an identifier (ID) to each metric. The ID is used to refer a transmitted metric class value to the according metric.

When the configuration is finished, the process will send a message to the front-end to either communicate failures or to acknowledge the configuration. If the configuration was loaded without failures, the message contains the metric configuration with the associated IDs. Otherwise, it is used to transfer an error message. After sending the acknowledgment message, the BE suspends its execution until the FE sends a message to start the monitoring process.

To capture the monitoring data the back-end starts a separate thread. The capture thread executes an infinite loop until the thread is stopped. Inside the loop, the values for the registered configurations are read and stored in an internal structure.

To reduce the amount of data that has to be transported the BE will classify the metric value. Therefore the BE determines the class of each metric value according to the configuration of the intervals for the particular metric. To reduce the message size even further, the BE will only store the class of a metric into the internal

structure if its class value has changed since the last capture interval.

After all metrics for a certain interval have been captured the thread sends the metrics stored in the internal structure to the FE. To send the captured and classified data to the FE, the system will use a dedicated stream to transport the metric values.

After the metric class values are transfered to the FE, the capture thread will suspend its execution until the next capture interval. To keep the capture intervals as accurate as possible the `pselect(...)` function will be used, since it uses the most accurate time structure and timer available. To determine the timeout, the program will take a time stamp when the loop is started and a second time stamp before it suspends. The difference of these two timestamps is the time that the capturing process took, and that has to be subtracted from the timeout to the next interval.

2.2.3 MRNet Filter Plug-in(s)

The filter plug-in is loaded by the intermediate children during runtime and associated to the stream that is used to transport the metric class values. To load the plug-in into the IC it has to be built as dynamically loadable library (a shared object file on UNIX systems).

The intermediate children receive all packages from their particular children (either a BE or another IC), and apply a filter function to the data associated with a particular stream. The filter function for the metric stream aggregates the metric class values of the received packages and arranges the data into a new packet. The new packet is sent up the TBŌN, either to another IC or the front-end.

The MRNet library provides the possibility to store the current state of a filter. The state data is used to recover the packet states if an IC fails.

3 Implementation Strategy

3.1 Prerequirements

The monitoring system relies on some third party libraries that have to be installed on the system and will be described in the following sections.

3.1.1 MRNet - A Multicast/Reduction Network

The MRNet software distribution is the foundation for the presented monitoring system. It provides a library, `libmrnet.a`, and the `mrnet_commnode` program that runs on the intermediate children and provide the communication between the frond-end and the back-end processes.

The library provides an API to access the functions of the TBŌN. MRNet is an implementation of a TBŌN provided by the Paradyn Project [27] from the Computer Sciences Department of the University of Wisconsin.

The MRNet library implements an efficient way to communicate with a large amount of nodes and to distribute processing functionality across multiple nodes. The front-end application is the root of the logical tree and the back-end processes are the leaves of the tree. The nodes between the root and the leaves are the ICs. They are used to process the data flowing up or down the tree. MRNet provides functions to assign filter functions to the data by loading plug-ins on the IC.

Since there is currently no binary distribution available, it is necessary to build and install the MRNet library and the executables, before it is possible to compile the monitoring system. The sources for the MRNet library and the documentation are available at the MRNet project page (http://www.paradyn.org/mrnet/).

The library has to be installed on all nodes that are part of the TBŌN (FE, ICs and BEs). Furthermore it has to be ensured that the library and executables can be found on the nodes by setting the according environment variables.

The following subsections will explain the major parts of the API in more detail.

End-Points

An MRNet end-point is an application process (back-end) running on the leaf nodes of the overlay network. The communication between front-end and back-end works through streams. While the front-end can communicate to all back-ends or groups of back-ends (by using communicators) in a unicast or multicast fashion, back-end

processes can only send messages to the front-end, but not directly to each other.

Communicators

A communicator is the MRNet way to represent a group of back-ends similar to a communicator in Message Passing Interface (MPI). Communicators provide a handle that identifies the end-points for point-to-point, multicast, or broadcast communication. Where MPI applications typically have a non-hierarchical layout, MRNet enforces a tree layout for all processes with the front-end as its root. Therefore the front-end is responsible to create and manage the network and communicators.

Streams

The streams are the logical channels for the communication from front-end to the back-end processes. All communication uses a stream either as downstream, between the front-end and the back-end processes or as upstream in the opposite direction. Streams transport the data in a specific type that can be specified with format strings (see Appendix B) similar to the C style formatted I/O.

Filter

Filter are functions that are contained in the filter plug-ins. To process the data, while it is flowing through the network, a filter function can be associated to a stream. A filter function is associated to a stream at the streams creation. MRNet uses two different filter types, synchronization and transformation filter. The synchronization filter organizes the flow of the data through the network and the transformation filter works on a packet of a specific type.

A synchronization filter is only working on the upstream. The MRNet library currently supports two types of synchronization filters:

SFILTER_WAITFORALL The filter plug-in waits to receive the packets from all its children before the packets can be processed.

SFILTER_DONTWAIT A received packet will be processed as soon as it arrives at the filter.

The transformation filter can be used in both directions. They are used to combine multiple packets and perform computational operations on them. Therefore, a specific format string has to be specified for the filter. The format string of a particular packet and the stream filter function format have to be the same that the packet is processed.

Startup modes

The library supports two modes to initialize the TBŌN.

Mode 1 In the first mode MRNet creates all intermediate and back-end processes for the TBŌN using the specified topology. The topology is defined in a configuration file (see Appendix A.1.3) that contains the nodes and their position in the TBŌN. The front-end starts the first level of the tree processes using a remote shell. The newly created processes will establish a network connection to the process that created them. After the network connection is created the newly created processes will receive the configuration of the subset of the TBŌN. The configuration is used to instantiate the sub-tree of the according processes. This is done, until the TBŌN is completely instantiated.

Mode 2 The second mode is used, when the system is instantiated with a process management system. In this mode the library instantiates only the internal nodes as in the first mode. The back-end processes are not created. The program that uses MRNet has to wait for the back-ends to connect autonomously.

3.1.2 Boost C++ Libraries

Boost is a collection of free, widely useful and usable software libraries that are working well with the C++ standard library. The monitoring system is using the program options and the serialization library and some data types defined by Boost. Boost is available as binary distribution for many architectures.

3.1.3 lm-sensors

Libsensors is part of the lm-sensors project [17] that is available for most available Linux distributions. The lm-sensors project provides access to many hardware monitoring capabilities in today's computer systems. The project continues to be in development and provides access to the monitoring capabilities by using OS kernel drivers, a user-space library, and tools.

The monitoring system accesses hardware monitoring capabilities by utilizing the functions provided in libsensors. To use the library lm-sensors has to be installed and properly configured. The functions provided by the library are defined in the `sensors/sensors.h` header file.

3.1.4 MySql++

To store the captured monitoring data the monitoring system is using a MySQL database. To access the MySQL database the monitoring system utilizes MySql++

[1]. MySql++ is a C++ wrapper for MySQL's C API. It provides functions to open a connection, handle queries and results, and to deal with exceptions.

3.1.5 pthreads

To use threads inside the programs, both, the front-end and the back-end, utilize the pthreads library. It provides a Linux implementation for the IEEE POSIX 1003.1c standard. The pthreads library provides functions to manage threads, mutexes to lock critical sections, condition variables to communicate between threads that share a mutex and routines for thread synchronization.

3.1.6 libltdl

The libltdl library is part of the libtool package [14]. Libltdl provides an interface that hides the complexity of the usual dynamic object loading mechanisms using the dynamic loader provided with libdl. Modules are loaded with a simple call to `lt_dlopen` which returns a handle to the loaded module. Each module provides a `getInstance()` function that is called to instantiate the module. To find the function in the module and to instantiate it, the address to the `getInstance()` function is looked up with a call to `lt_dlsym`. Due to the different naming scheme of C and C++, the `getInstance()` function in the modules has to be marked as C code using the "`extern "C"`" declaration.

To avoid problems with different library versions, the library itself is contained in the project and will be compiled during the build process. Libtool needs to be provided with the `-module` switch to build libltdl compatible modules, which is done in the corresponding `Makefile.am` in the modules subdirectory of the back-end source tree.

3.1.7 libconfig

Libconfig [13] is a library to process structured configuration files. It is used in the FE to read the configuration. The library uses a compact file format. Additionally it is type aware, so no string parsing is necessary.

3.2 Development Environment

The software is implemented in a Linux environment because most HPC environments are running on it. As database management system MySQL is used. To build the project the GNU autotools [10] are used. Since the MRNet API is written in C++ it is also the programming language for this project.

The source and header files are structured inside the project directory. All header files are located inside the `include` directory, the source files are to be found in the `src` directory. The sources again are structured in directories. Every part of the system has its own directory (`be, fe, filter...`).

All directories contain a `Makefile.am` file to configure the make system for the corresponding directory.

3.3 Implementation

The monitoring system consists of three parts, front-end, back-end and filter plug-in (see Section 2.2). The implementation of the monitoring system will start with the very basic implementations of front-end and back-end. The goal for the first implementation phase is to have a working network communication between front-end and back-end. The next step is to implement the module loading mechanisms, the modules for the back-end and the classes needed to load, serialize and transmit the back-end configuration. In the last step the filter plug-in and possible add-ons will be implemented.

The following sections will give a brief description of the parts of the monitoring system and how they have to work together to capture and reduce the monitoring data, and to store the health state of a HPC system into a history database.

3.3.1 Front-end

The front-end is the root process of the monitoring system. It has to evaluate the command line, read the configuration and is responsible to start up the TBŌN and all belonging processes. After instantiating the TBŌN the metric configuration has to be transmitted to the BE processes. The BE processes will acknowledge the configuration with a message, that has to be verified by the FE. After the initialization and configuration, the FE will start the monitoring process. The monitoring data has to be received by the FE and stored into the database.

To read and validate the command line parameters, the front-end will utilize the `boost_program_options` library, which provides an easy to use interface to an extendable command line parser. The command line is used to pass the path to the topology file, the configuration file and to the back-end executable (see Appendix A.1.1) for the command line parameters). If the BE processes can not be started by the FE the number of back-ends has to be defined instead of the back-end executable. This will cause the FE to use the second startup mode of the MRNet library (see Section 3.1.1).

To read the configuration file the front-end uses libconfig [13] that provides a C++ interface to a configuration file parser. Since the objects provided by libconfig provide only access to a static data structure that is hidden in the library there is a need for an own data structure to send the configuration to the back-ends. The configuration structure therefore needs to be serializable. To have a serializable data structure a set of classes will be implemented that can be serialized using the `boost_serialization` library and represent the structure of the metric configuration.

The front-end builds the data structures according to the configuration. To pass the configuration objects to the back-ends they have to be serialized. To serialize the objects the boost serialization library will be used. Therefore all serializable objects have to implement an accessor method that is called by the Boost serialization library.

To pass the configuration to the back-end processes the front-end uses TBŌN that is provided through the MRNet library. Both, the front- and the back-end have to be linked against libmrnet to utilize the library. To use the TBŌN provided by MRNet the front-end has to create a network object and to associate a topology to it. The topology has to be configured in a separate configuration file. The file's location is passed to the front-end by command line and used to create the network object.

The MRNet library supports two different kinds of network creation. The first method starts all processes needed by the software (intermediate and back-end processes) and the second method only starts the intermediate processes. In the second mode the back-end processes have to be started separately either by a job submission system or an administrator. The front-end will support both start up mechanisms. If the second mechanism is used, the front-end has to create a file, that stores the connection parameters for the different back-end processes.

If the network object is created successfully, the front-end can create a communicator that contains all back-ends (broadcast communicator) and create a stream object with all back-ends in the communicator subsequently. The created stream (configuration stream) is used to pass the metric configuration to the back-end processes by sending the message that contains the configuration. After the configuration is send, the front-end waits for the back-end processes to acknowledge the configuration and to start the monitoring process. To start the monitoring process, the front-end will create a second stream for the monitoring data and assign a filter

function to it. To receive the incoming packets a listener thread that receives all messages from the TBŌN is started and the the monitoring process is started by sending a message to the back-ends.

3.3.2 Back-end

The back-end is responsible to capture the metric values from the different sources. To be as portable and flexible as possible the reader for different metrics are implemented as modules that can be loaded dynamically by the back-end. As part of this work the program will support two different kind of metric modules: one that uses the information provided by the kernel in the `/proc` file system and a second that uses lm-sensors [17] to read metric values from hardware sensors. To load the modules the back-end will utilize libltdl (see Section 3.1.6) that is part of the libtools [14] package.

Through the modular design the software can easily be extended to capture monitoring data from other sources by adding a new modules to the monitoring system. Therefore all modules have to use the same interface to access the metric reader. To to achieve the required portability and modularity the modules will all use the same base class that defines all methods to instantiate the reader in the module.

A metric reader is implemented for each metric covered in a certain module. It implements the methods to capture and return the currrent metric value.

To load the modules during run time, the back-end needs to know what metrics will be captured and which module is needed. Therefore it has to be configured. Although the back-ends could load the configuration for their own it is more convenient to have a centralized configuration. Separate configurations for front-end and back-end would make it necessary that all compute nodes either share a network file system or the configuration for every node has to be maintained. So the configuration will be provided by the front-end.

To receive the metric configuration, the back-end has to instantiate the TBŌN. In the case of the back-end the connection parameters will be provided by the command line. The parameters for the start up of the back-ends are either provided by the front-end if the automatic start up is used or they can be read out of the file written by the front-end for the separate back-end start up.

After the network is set up, the back-end processes wait for the configuration to be transmitted by the front-end. When the configuration message is received, the content is deserialized and the configuration is accessible in the back-ends. The configuration is structured. A configuration object contains an object for each module

that has to be loaded. The module objects again are a container for the configuration of the different metrics that have to be captured by the module. The back-end iterates through the configuration and loads all modules and registers the readers for the configured metrics. After setting up all reader the back-end acknowledges the received configuration and waits for a message to start the monitoring process.

The monitoring is done by a separate thread. The monitoring thread iterates over the registered reader and sends the different metric values in to the front-end using the TBŌN. The thread is stopped, when an quit message is received from the front-end and the back-end process exits.

3.3.3 Filter Plug-in

The filter plug-in is executed on the intermediate processes of the TBŌN. A filter is a shared object, that can dynamically loaded by the intermediate processes. It defines filter functions which can be associated to a data stream. To aggregate the data send by the back-end (actually by the child processes of the intermediate process) the filter plug-in will implement a function to merge the different incoming packets into a new packet and send it to its parent. To load the filter plug-in the front -end uses a dedicated MRNet method to load the filter on all intermediate processes. All processes that need to load the plug-in must know where to find the plug-in (by providing the path in the environment).

The filter plug-in can implement different filter functions. MRNet defines a naming scheme for all parts of a filter function. The function name itself has to end with a "`_func`" suffix.

As a filter function is assigned to a stream and it can be possible to use a stream with different packet types (by defining different format strings, section 3.1.1) a filter function needs to know the format of the packet it can work on. The format string has to be defined by a symbol that is named with the filter function name and the "`_format_string`" suffix (e.g. `Merge_func_format_string`). All packets using the specified format are processed by the filter function.

3.3.4 Communication

The MRNet library uses streams to send data through the TBŌN. To send data to a stream, the data has to be packed into a packet. The format of the packet has to be specified to match the containing data. To specify the data format MRNet uses format strings (see Section B) similar to the standard c library. To classify the data in a packet a tag has to be assigned to the packet. The tag is an unsigned integer and

can be chosen by the developer. Since it is used for MRNet internal communications as well, it has to be higher than the value defined in `FirstApplicationTag` (`mrnet/Types.h`).

To send simple messages like "start monitoring", "stop monitoring" and so on it is sufficient to send a packet and determine the message on the tag. In this case an empty format string can be provided. If the packet contains data it is necessary to provide a data format and can be added to the packet in the same fashion as in the standard libraries `printf(...)` function. The format that will be used in the monitoring system is "%auc" which specifies the data as an array of unsigned chars (bytearray).

The monitoring system will use two different streams. One stream for the configuration and command data and one stream for the metric values. The metric stream will have a filter function assigned to it to merge the packets flowing up the TBŌN.

3.4 Testing

To test and debug the software and to collect data for the evaluation the system will be tested on the development workstation (preferably for testing and debugging) and on the XTORC cluster (a small 64 node system in the Computer Science and Mathematics Division at the ORNL).

There are different tests to conduct to either measure the amount of data produced by the monitoring system over a defined period of time or how it handles faults in the underlying tree (e.g. back-end or communication node failures).

To collect the monitoring data over a period of time the system has to run with a reasonable configuration either on one machine with multiple instances or on a cluster system. A reasonable configuration would be a similar or same configuration to the tests in [15]. To gather accurate data that show a usual usage pattern, the machine should have some work load system during this time. Additionally it may be usable to cover up fan intakes to artificially increase the systems temperature and therefore a change of the gathered data.

4 Detailed Software Design

This section describes the design of the implemented monitoring solution. All programs deployed with the monitoring system and their components will be discussed in detail and the interaction of all components will be explained. (see Figure 4.1) shows the basic process of starting the monitoring system and capturing the monitoring data.

The Front-End is started on the monitoring host. After its start up it loads the configuration and initializes the TBŌN. The MRNet library starts the Back-End processes while the TBŌN is created. After the TBŌN is set up and the Back-End are running the Back-End configuration is read by the Front-End and transmitted to the Back-Ends. All Back-End processes load the Modules according to the configuration and send a validation message back to the Front-End. The messages are validated by the Front-End and if no errors are reported the monitoring process is started.

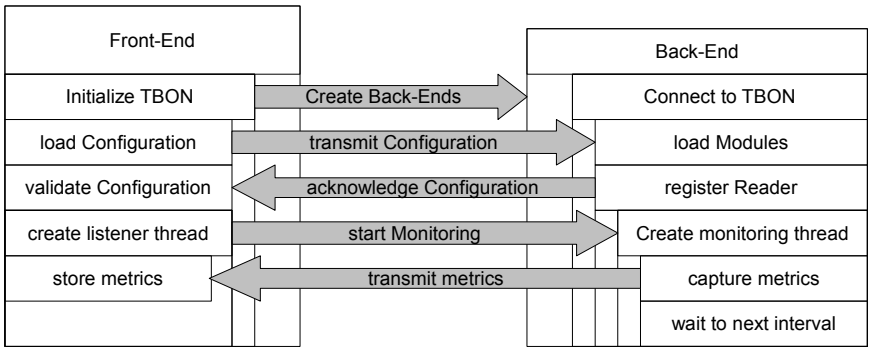


Figure 4.1: The basic procedure of the monitoring process. The Front-End initializes the TBŌN, loads and transmits the configuration. The configuration is acknowledged by the Back-End and validated on the Front-End. If the configuration is valid the actual monitoring process is started. The Back-End captures and transmits the metrics and the Front-End receives and stores the metrics into the database.

4.1 Front-end Daemon

The font-end daemon is the main application of the monitoring system. It should run on a monitoring node and is responsible to control the monitoring system, for

it's start up and the configuration of the system. All captured monitoring data will be send to the front-end daemon which will store the metric data into a MySQL database.

The front-end program main source code file is `rasmonfed.cc` which contains the main function and some functions to start the front-end application as a system daemon. After the program start an `Application` object (see Section 4.7.1) is instantiated. The `Application` class is implemented using the singleton design pattern what makes it possible to access the `Application` object from every routine of the program by calling the static `getInstance()` method. Inside the `Application` class all the functions to initialize and set up the monitoring system, as well as the main loop to handle incoming packets are implemented.

The basic program flow of the front-end is divided into tree parts, initialization and configuration of the TBŌN, receiving and processing of the monitoring data and the shutdown of the monitoring system. The following sections will describe these parts.

4.1.1 System Initialization & Configuration

To initialize the front-end the first step for the program is to create the programs main object, the `Application` (see Section 4.7.1) instance and to parse and validate the command line parameters. To parse the command line parameters the `main` arguments are passed to the `initialize(...)` method of the `Application` object (see Section 4.7.1). If all required parameters are provided, the program continues otherwise a error message is written to the Log and the program exits.

After parsing the command line parameters the program determines either if it has to run as daemon or as interactive program. The default is to run as a daemon and call the `daemonize()` function.

To run the program as a daemon, the first step is to fork a new process. While the original process will exit, the new created process will call `setsid()` to detach the process from its parent. To ensure that just one daemon is running the process opens a lock file and tries to lock the file with the `lockf(...)` system call. If the lock file can not be opened or locked the program exits.

The next step that is required for running as a daemon is to register the signals that it wants to receive. When the system is quitting a program it sends a `SIGTERM` to the program. If the configuration has to be reloaded a `SIGHUP` can be send. To receive the signals and to handle them, the program has to register a callback

function to handle the signal. For simplicity there is only one signal handler in the font-end `signal_handler( int sig)` function that handles all registered signals.

When the front-end is set up the configuration files are parsed. The locations of the two configuration files for the monitoring system are provided by command line parameter and stored in variables in the `Application` object. One file will describe all the necessary settings for the monitoring system itself and a second file will describe the topology of the TBŌN and is needed by the MRNet library.

A call to the `loadConfiguration(...)` method loads and parses the system configuration file utilizing the libconfig library [13] (see Section 3.1.7). The file contains two main sections, one section for the front-end configuration and on section for the metric configuration. The file format is described in the program documentation (see appendix A).

If libconfig returns no errors, the next step for the `Application` class is to setup the TBŌN. To set up the network MRNet requires a configuration file that contains the description of the of the networks tree structure. The location to the configuration file is passed to the network constructor. To set up the network the `Application` class calls the `setupNetwork()` method.

After the network is set up, the metric configuration will be parsed and passed to the back-ends. To represent the configuration and to send it to the back-ends the front end builds a serializeable structure of the configuration. The following classes are needed to represent the configuration:

- MetricSectionConfiguration
- MetricModuleConfiguration
- MetricConfiguration
- MetricSetting

The configuration is explained in detail in section 4.6.

If the configuration is parsed without error the object tree will be serialized and send to all back-ends. The details of the communication through the TBŌN are explained in section 4.3. After the configuration was sent, the front-end has to wait for the back-ends to acknowledge the configuration and to return the ids of the metrics. The ids are used to associate the package containing the class value to the according metric.

4.1.2 Processing Monitoring Data

Before the monitoring process is started, the front-end waits for the back-ends to acknowledge the transferred configuration. The Acknowledgment is necessary to verify that all back-ends have initialized the metric modules successfully and no error has occurred during the initialization. Additionally the back-ends assign an id to the metrics that have to be the same for all of them. If the verification shows any differences or errors, the front-end writes an error message end will shut down.

If the verification succeeds the front-end initializes a new stream to transport the metric data. The metric stream has a filter assigned to it, that is responsible to merge the packets from its children to one single packet (for details see 4.5). After the filter is loaded and the stream is created without errors, a `PacketQueue` (see Section 4.7.22) and a `PacketListener` (see Section 4.7.21) are instantiated. The `PacketListener` is started and the back-ends are notified to start the monitoring. The notification is send through the metric stream. This is necessary that the back-end can store and utilize the instance of the metric stream to send the metric values.

As soon as the listener is started it monitors the network event provided by the MRNet library. If a packet is send the event is triggered and the listener returns from the `pselect` call, takes a time stamp and reads the available packet. The time stamp and the packet are stored into a structure and the structure is added to a queue.

The second thread is responsible to process the received messages.

Processing Packages

The processing is done by the main thread. After all initialization is done the `run()` method of the `Application` object is called. The run method instantiates the `PacketListener` and the `PacketQueue` objects, creates the metric stream and sends the message to start the monitoring. After starting the `PacketListener` thread, the application enters a infinite loop that determines if packets form the back-ends are available in the `PacketQueue` by calling the queues `getPacket()` method. If a packet is available, the method returns the first packet in the queue, otherwise it blocks the call until a new packet is available. This implementation ensures that it is not necessary to poll the queue for available packages.

The MRNet library requires to associate a packet with a tag, which are used to determine the type of the packet. There are several types for different purposes,

including the current packet to transfer the metric values from the back-ends to the front end. Inside the loop the tag is used to determine the action by a switch statement.

All packets have a tag, based on which the different packets can be distinguished. The packet tags are defined in `mrn-communicator.h` and are used through all programs.

4.1.3 System Shutdown

To trigger the shutdown of the system, the program has to receive either the `SIGKILL`, if it is running as daemon, or the `SIGINT` signal if it is running in interactive mode. The shutdown is handled in the signal handler that is registered for the according signals.

To shutdown the system properly the front-end informs the back-end processes by sending a quit message, to inform the BE that the system is going to shut down. The back-ends stop the execution of the capture threads and exit. The front-end stops the listener thread, empties the buffers and closes all open handles.

After all threads are stopped, allocated memory is cleaned up and the front-end exits.

4.2 Back-end Daemon

The back-end daemons are going to run on all hosts that have to be monitored. In the current implementation of the monitoring system they have to be started through the frond-end process.

As well as the front-end, the back-end has to initialize the TBŌN using the MRNet library. Instead of passing a topology file to the network constructor, the back-end needs a server address and a port number to initialize the network. These parameters are passed to the back-end during the start up. The connection parameters are set with command line parameters, either by the MRNet internal mechanisms for the network instantiation or by an administrator or resource manager if the second startup mode is used.

After the network is set up, the back-end enters a infinite loop and listens on the network for incoming packets. The incoming packets are categorized by their tag. Inside the loop a switch statement controls the process flow. The current implementation differentiate three tags: `RASMON_CONF`, `RASMON_START_MONITOR` and `RASMON_EXIT`.

The detailed actions to the packages with these tags are described in the follow-

ing sections.

4.2.1 Configuration

After the back-end process joins the TBŌN it listens for incoming packets. If a packet with the tag `RASMON_CONF` arrives the pointer to the stream associated with the packet is stored in the `mp_ControlStream` variable and the `setUpBackEnd(...)` function is called and the packet containing the configuration provided by the front-end is passed to it. The configuration message is described in section 4.6.

The `setUpBackEnd(...)` function unpacks the packet and deserializes the containing message using `boost::archive::binary_iarchive`. As a result the `MetricSection` (see Section 4.7.16) object is available in the back-end. The metric section is passed to the `loadModules(...)` function.

To load the metric modules (see Section 4.4) the back-end processes utilizes libltdl which is part of the GNU libtool project [14]. Before `libltdl` can be used, a call to `lt_dlinit()` is required.

To load the modules the `loadModules(...)` function iterates through the content of the `MetricSection` object. The `MetricSection` object contains a `MetricModuleConfiguration` (see Section 4.6.2) object for each configured module. The module filename scheme is mod<name>.la and the has to be assembled by attaching the module name, that is included in the `MetricModuleConfiguration` object, to the "mod" prefix and appending the ".la" postfix (e.g. `modmemory.la`).

A call to `lt_dlopen(...)` loads the module and returns a handle to the module. With that a call to `lt_dlsym( lt_dlhandle, "getInstance")` can be used to receive a pointer to the function with the provided symbol, in this case "getInstance". A call to the function pointer initializes the module and returns a pointer to the module object, that is stored in a `std::map` with the module name as key. The function returns an integer value after all modules are loaded that is indicating if an error has occurred.

After all modules are loaded successfully the metric readers for the according metrics are instantiated by calling the `registerReader(...)` function and passes the pointer to the `MetricConfigurationSection` to it.

The `registerReader(...)` function processes all configuration objects in the `MetricConifurationSection` in an outer loop. The `MetricModuleConfiguration` object contains a set of `MetricConfiguration` that are

processed in an inner loop. For each metric configured inside a particular module configuration the `getMetricReader(...)` function of the according module is called and the corresponding configuration is passed to it. As result a `MetricReaderCollection` object containing the pointer to a `MetricReaderInfo` (see Section 4.7.14) object (in case of the network module, the pointer to the `MetricReaderInfo` objects for each configured network device) is returned.

The pointer to the `MetricReaderInfo` objects contained in the `MetricReaderCollection` are stored into a multimap using the interval as the key. When all `MetricModuleConfiguration` objects are processed, the function returns, the configuration is send to the front-end for validation and the back-end waits for a new message from the front-end.

4.2.2 Capturing Metric Values

When the back-end has initialized all metric reader objects and the metric configuration is send to the front-end, the back-end suspends its execution until a packet is received. To start the monitoring, the BE waits for a packet that contains the `RASMON_START_MONITOR` tag. The packet containing this tag is send by the FE through the metric stream.

As soon as a back-end receives this packet the pointer to the stream is stored to a dedicated variable and all metrics are send to the front-end using this stream. To capture the metrics the back-end instantiates a `MetricController` (see Section 4.7.5) object and assigns the network and the metric stream to the object.

The `MetricController` class is derived from `Thread` (see Section 4.7.23) class and runs in a separate thread. After instantiating the `MetricController` the `start()` method is called and the capturing of the metric values begins.

The `MetricController` executes an infinite loop that is capturing the metric values. A detailed description of the `MetricController` can be found in section 4.7.5. After the `MetricController` is started the main thread enters the listening loop again and waits for new instructions.

4.2.3 Back-end Shutdown

To shut down the back-ends, the front-end will send a message that is tagged with the `RASMON_QUIT` flag. If the quit message is received by the back-end it stops the `MetricController` thread and waits until it joins the main thread. Subsequently the back-end frees all allocated resources and exits the receive loop.

4.3 Communication

The MRNet library is implementing a TBŌN that connects all back-ends with the front-end. To instantiate a network the library needs a configuration file, that contains the layout of the network. After the network is set up, a communicator has to be instantiated. A communicator is network specific and the creation methods are are functions of a instantiated network. There are two ways to instantiate a communicator, either using `new_Communicator` to create an empty communicator or the `get_BroadcastCommunicator()` function to create a communicator containing all back-ends available in the network to the time of its call.

To send messages, the MRNet library uses streams. Streams are also network specific and created by calling the `new_Stream` method of the network instance. A stream is associated with a number of endpoints, that are expressed by a communicator passed to the `new_Stream` call. The filters are also associated to a stream object and there are three more parameters to pass to the call, to set up the filters that have to be used in the according stream.

MRNet defines three types of filter, the upstream filter that is applied to all data flowing from the back-ends to the font end, the synchronization filter which determines whether a intermediate node of the network waits for all child nodes to send their data or not and the downstream filter that is applied to all packets from the front-end to the back-ends. The filter can either be associated by their filter id or by using a string. Using the filter id applies the filter to all nodes in the described by the communicator associated with the stream. Applying the filter with a string offers the possibility to assign a filter to a subset of nodes in the stream.

4.3.1 Send Data

Data packets are send through a specific stream, so the MRNet library provides a `send` method for a stream object. To send a packet a tag and a format string are required. The tag is used to classify the data in the packet. It can be freely chosen by the developer with the restriction that it is greater than or equal to the constant `FirstApplicationTag` defined by MRNet (`mrnet/Types.h`). The format string describes the containing packet data (see appendix B).

To ensure the that the packets are send immediately after the send method returns without error a call to `Stream::flush()` commits a flush of all packets in the buffer of the stream.

The monitoring system uses a wrapper to call the send operation. It is imple-

mented as a static function in the `MRNCommunicator` class (see Section 4.7.19).

4.3.2 Receive Data

The MRNet library provides a network and a stream specific method to receive data packets. Both methods return an integer value to indicate the return status of the call and have to be provided with pointers to the variables that store the tag and the packet and a flag to indicate if the call has to block or not. The network specific read method additionally requires to pass a pointer to a variable to store the stream where the packets have been read from.

The variable to store the tag is an integer value and has to be passed as a pointer. To store the packet, the MRNet library provides a `PacketPtr` class to store the content of a packet and has to be passed as reference.

4.4 Metric Modules

The metric modules are the part of the software solution that is actually responsible to capture the monitoring data. Each module is a dynamic loadable module and has designated capabilities. The modules are responsible to capture the monitoring data from a specialized data source. Picture 4.4 depicts the basic layout of a metric module.

The systems currently has 4 different metric modules (Network, Memory, LoadAvg and Sensors). Each of this modules implements a set of metric reader specific to the modules domain. The actual implementation of a single module varies with the data source but the methods to access the module have to be consistent for all modules.

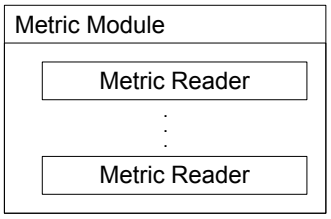


Figure 4.2: A metric module is a dynamic loadable object. Each module contains the methods to initialize the module and to return the metric reader that are contained in the module.

To achieve this all modules provide a similar interface to access the internal functionality by extending the `Module` class. The functionality of a module is

hidden from the back-end process through a class hierarchy.

Each module implements a class that is derived from the `Module` class (see Section 4.7.18). All modules are implemented by using a singleton design pattern to ensure that only one instance of a certain module will exist during run time. The abstract `Module` class defines a virtual function (`getMetricReader(...)`) to instantiate the reader objects responsible to access the metric data provided by the certain module.

The (`getMetricReader(...)`) needs a `MetricConfiguration` object as parameter and returns a pointer to a `MetricReaderInfoCollection` object. The information provided inside the `MetricConfiguration` is used to determine which reader has to be instantiated. The `MetricReaderInfoCollection` object is a container that allows to return more than one reader. This is used by the network module. All reader objects returned are of the type `MetricReaderInfo`, what allows the system to capture the metric values without knowing the actual implementation of the reader. The `MetricReaderInfo` object contains the pointer to the actual metric reader, a pointer to the configuration of the metric associated with this reader and some variables needed by the capture process (see Section 4.7.5).

The class for a particular reader object derives from the abstract `MetricReader` class which defines the interface for all metric reader. The reader classes will be discussed in the following sections that are covering the 4 modules.

To instantiate the a module, the back-end has to call the `getInstance()` function that has to be implemented by every module. Since the modules are loaded through `libltdl` the `getInstance()` function has to be enclosed by a `extern "C" { ... }` definition. The `getInstance()` function creates the according module object and returns the object pointer.

4.4.1 Network Module

The network module is responsible to capture the metric values of the network devices. It is defined in the `module-sensor.h` file. The implementation can be found in the `module-sensor.cc` file. It contains the metric reader that return the information provided by the kernel in the `/proc/net/dev` file.

Since there is a metric for each of the activated network devices it is necessary to differentiate between the devices. This goal is achieved by adding another class into the hierarchy, that has a method to set and store the according device for the corresponding reader. The `MetricNetReader` class is derived from `Metri-`

cReader but does not implement the methods defined in MetricReader.

For each of the metrics the module provides a class that is derived from the MetricNetReader class. Each metric reader class in the module implements the getMetricValue(...) method that is defined in MetricReader class to return the current metric value to its caller.

To read the metric values from the proc file system the module provides a NetProcFileReader class which does the actual work of gathering the metric values. The NetProcFileReader is implemented using a singleton pattern. This is done to avoid unnecessary overhead since the reader itself will query the /proc/net/dev file only once in an certain interval and return the values valid in this interval to each of the reader.

Metric Readers

The different network metric readers derive from the MetricNetReader (see Section 4.7.10) class. They implement the getMetricValue(...) metho that returns the current metric value.

The network module currently supports 6 metrics.

1. BytesTransmit
2. BytesReceive
3. PacketsTransmit
4. PacketsReceive
5. ErrorsTransmit
6. ErrorsReveive

4.4.2 Memory Module

The statistics for the system memory are gathered by the memory module. It is defined in the module-sensor.h and implemented in the module-sensor.cc file. It is a very basic module, that reads its data from the /proc/meminfo file which contains very accurate statistics for the usage of the memory.

For each of these metrics a Reader is implemented deriving directly from the MetricReader class. Again the data is read from the file and processed by a class derived from FileReader, the MemoryProcFileReader which is using the singleton pattern as well. To get the current metric values the reader searches the

content of the file, this time for the keywords associated with the according metric. The keywords are defined in the header file `module-memory.h`.

MetricMemoryProcFileReader

To capture the memory metrics, the `MetricMemoryProcFileReader` reads the information contained in the `/proc/meminfo`. The metric values are ordered in lines. They start with the metric name followed by a colon and the value in kB. To read the values the file content is read into a buffer (a std::string). The string is split up on the line ending and each substring is again split up on the colon. The metric name is used as a key to store the metric values for the current capture interval into a std::map and the value is stored as a integer value.

Metric Readers

The current implementation for the memory modules supports the following 10 metrics:

MemFree The amount of physical RAM, in kilobytes, left unused by the system.

Buffers The amount of physical RAM, in kilobytes, used for file buffers.

Cached The amount of physical RAM, in kilobytes, used as cache memory.

SwapCached The amount of swap, in kilobytes, used as cache memory.

Active The total amount of buffer or page cache memory, in kilobytes, that is in active use. This is memory that has been recently used and is usually not reclaimed for other purposes.

Inactive The total amount of buffer or page cache memory, in kilobytes, that are free and available. This is memory that has not been recently used and can be reclaimed for other purposes.

SwapFree The total amount of swap free, in kilobytes.

Dirty The total amount of memory, in kilobytes, waiting to be written back to the disk.

Writeback The total amount of memory, in kilobytes, actively being written back to the disk.

Mapped The total amount of memory, in kilobytes, which have been used to map devices, files, or libraries using the mmap command.

4.4.3 LoadAvg Module

The LoadAvg module is implemented the same way as the two modules above. It definition can be found in the `module-loadavg.h` and the implementation in the `module-loadavg.cc` file. The `LoadAvgModule` class is derived from the `Module` class. The module implements the module instantiation as well as the `getMetricReader(...)` method to return the `MetricReader` objects for the metrics provided by this module.

The module contains a `LoadAvgProcFileReader` class that is derived from `FileReader` and implements the necessary functions to read the load average values that are provided by the kernel in the `/proc/loadavg` file.

The `getMetricValue(...)` method, implemented in each of the reader classes, calls the `readMetricValues(...)` method in the `LoadAvgProc-FileReader` class and returns the current metric value.

MetricLoadAvgProcFileReader

The `MetricLoadAvgProcFileReader` class implements all the functionality to extract the information to the load average out of the `/proc/loadavg` file in the `readMetricValues` method. If it is called, the values for the three different load average values are read from the file and stored in a float variable for each of them using the `sscanf` function.

To receive the values a method for each of the three values is implemented (`getLoadAvgOne`, `getLoadAvgFive`, `getLoadAvgFifteen`). The functions compares the provided time stamp (value of type `timepec`) with the last capture time value and if they are the same it simply return the value stored in the corresponding variable. If the values are different the get function calls the `read-MetricValues` method and returns the value after the new values where read from the `/proc/loadavg` file.

Metric Readers

Again there is a reader class that derives from `MetricReader` for each metric provided by the module:

- MetricLoadAvgOneReader
- MetricLoadAvgFiveReader
- MetricLoadAvgFifteenReader

4.4.4 Sensors Module

The sensors module is different to the 3 modules above. The class definition can be found in the `module-sensor.h` file, `module-sensor.cc` contains the implementation. The Sensors module can query the growing amount of hardware sensors provided by libsensorslibsensors [17] a Linux hardware monitoring solution. Libsensors provides an interface to query the different hardware sensors provided by the system and configured in the according configuration file. It requires a working installation of lm_sensors and is configured in the `sensors` section of the metric configuration.

The sensors module initializes the interface to libsensors during its instantiation. While the module is initialized it searches the system for available sensors and the metrics provided by these. To find all available sensors, the module calls the `initSensors(...)` method inside its own constructor. In the `initSensors(...)` method the libsensors library is initialized by calling the `init_sensors(...)` function. The function needs the path to the sensors configuration file, that is currently hardcoded into the function. If the library is initialized properly the module continues its work, otherwise the method returns.

To load the metrics the module has to find the available sensor chips. Therefore the `sensors_get_detected_chips(...)` function is called in a while loop as long as the function returns a sensor chip.

To detect the available "features", the libsensors equivalent to metric, of the current sensor chip, the `sensors_get_features(...)` function has to be called in another while loop. As long as a feature is returned it is stored in the sensorfeature map. The different metrics are identified by a label that is defined in the lm-sensors configuration.

The sensors module implements the class `SensorMetricReader` that is derived from `MetricReader`. Again the different reader objects are instantiated by a call to the `getReader(...)` method of the module. To instantiate a SensorMetricReader it is necessary to pass a `sensorfeature` structure to the constructor. The structure contains all necessary information to read the metric of the according sensor. It can be received from the sensorfeature map by the label provided in the configuration.

SensorMetricReader

The `SensorMetricReader` class is a generic class. All possible metrics provided by lm-sensores are covered by it. To instantiate an `SensorMetricReader` object the `sensorfeature` structure has to be passed to the constructor. The `sensorfeature` structure contains all information to query the current metric value with a call to `getMetricValue`. The `getMetricValue` itself calls the `sensor_get_feature` function, provided by the libsensors library. A call to `sensor_get_feature` returns the current metric value. The value is of type double. Since the current implementation of the monitoring system uses only integer and float data types, the value is casted to a float. It is not likely that a loss of accuracy occurs with the cast.

4.5 Filter Plug-in

The filter plug-in is associated with the network stream that transports the monitoring data from the back-end daemons to the front-end. It is provided as a shared object file and will be loaded by the intermediate children of the TBŌN. To load the filter plug-in the frond-end has to call a method of the network object which returns a filter id that afterwards can be associated with a network stream.

Since MRNet uses `dlopen` the all C++ symbols must be exported as C symbols by surrounding the functions with `extern "C" {` and `}`.

The structure of a MRNet filter is explained in the MRNet documentation. A filter function has to use the following signature:

```
void filter_name(
        std::vector< PacketPtr > & packets_in,
  std::vector< PacketPtr > & packets_out,
  std::vector< PacketPtr > & packets_out_reverse,
  void ** filter_state,
  PacketPtr & params)
```

To recover from failures of IC the filter has to implement another function to set the state of the children moved to its sub tree of the TBŌN. The signature of this function has t be the following.

```
PacketPtr filter_name_get_state(
          void ** filter_state,
          int stream_id);
```

The current task of the filter plug-in is to repack the metric data in the incoming packets into a single new outgoing packet. this is done in a loop, that iterates over all packets and extracts the metric data and adds it into a new packed that after the processing is done is send to its parent in the TBŌN

4.6 Configuration of Metrics

All parts of the software that are used by more than one program of the solution (for example by the Back- and Front-end daemon) are part of convenience libraries that are linked to the corresponding program during the compilation process.

4.6.1 MetricConfigurationSection

This class is a container class that stores the configuration objects for all modules that are configured in the configuration file. Internally it is using a `std::vector` to store the module configuration objects. To serialize the object the class utilizes the serialization provided by the boost c++ library.

The serialization library requires the implementation of a `serialize` function, that is called by the boost serializer. The boost serializer itself uses a access class to access the `serialize(...)` function of the configuration. To allow access to the internal data of the class the `boost::serialization::access` class needs to be declared as a friend class.

To add `MetricModuleConfiguratio` objects to the container class it implements a `add(...)` function.

4.6.2 MetricModuleConfiguration

The MetricModuleConfiguration class is responsible to collect all configurations for the metrics defined in a specific metric module. Again the class is a wrapper for a vector that contains the objects with the configuration for the metrics associated with this specific module.

4.6.3 MetricConfiguration

In this class all necessary informations to configure the according metric are stored. It is a wrapper for a map container of the Standard Template Library (STL) which contains the MetricSetting object with the according value and the associated key for the setting defined in the configuration file.

It implements a function to set a value which requires the key and the MetricSetting object and a function to receive the MetricSetting for the according key. Again

a serialize function is implemented to meet the requirements of the serializer.

4.6.4 MetricSetting

This class contains the value of a setting, which can be an Integer, a Float, a String or an array of the associated types. The class implements functions to set and to get the corresponding value(s) to query the value type, the amount of the stored value(s) and if the MetricSetting object is an array. The values itself are stored in a vector of the STL.

4.7 Classes

This sections discusses all classes in detail that are part of the software and are not discussed in a different section.

4.7.1 Application

The `Application` class is the main object of the front-end. It uses a singleton pattern to enable the access to the object from every other class of the program without passing the pointer to it. A reference to the instance can be received by calling the `getInstance()` method, that returns the instance to the object. If no instance is created the `getInstance()` method will call the class constructor and create an instance. The pointer is stored in a static variable.

The application class implements the following methods:

void Error (string message)

bool daemonize ()

void initialize (int argc, char **argv)

void run ()

bool setupNetwork ()

bool setupBackEnds ()

void shutdown ()

static Application * getInstance ()

static void Log (string message)

setupNetwork()

The network initialization is implemented in the `setupNetwork()` method. It instantiates the network object provided by the MRNet library. Since the network has two different ways to be set up, the function has to determine which instantiation method has to be used. For the automatic instantiation of the network with back-end start up the back-end executable file has to be specified as command line parameter otherwise the back-ends will be started manually and the front-end process has to wait for all back-ends to connect.

Automatic Back-end instantiation

If the parameter for the back-end executable is specified, the containing variable stores the path to the executable file and the network has to be set up with back-end instantiation. To do so the `setupNetwork()` method calls the network constructor with the path to the topology file and the path to the back-end executable as parameter. The MRNet library starts the back-ends and the intermediate children automatically and returns the network instance on success.

Manual Back-end instantiation

If the parameter is not specified, the constructor is only called with the path to the topology file and the front-end has to wait until all back-ends have connected to the TBŌN. To provide the administrator or a automated system with the necessary information for the start up of the back-end processes, the front-end writes all information to a file that contains the hostnames of the intermediate children and the ports for the back-ends to connect. After the file is written the function determines the number of nodes that have to connect to the TBŌN and enters a loop that only stops if all back-ends have joined the network.

setupBackEnds()

To configure the back-ends, the front-end calls the method `setupBackEnds()`. It first serializes the `MetricSection` (see Section 4.7.16) object. To serialize the object the method uses a binary archive provided by the boost serialization library. The binary archive is stored in a `std::string` which is subsequently send to the back-ends using the `MRNCommunicator` (see Section 4.7.19). After the serialized configuration is send to the back-ends the method waits for a message from each back-end, that have to acknowledge the configuration. After all back-ends have acknowledged the configuration and the messages have no differences in the

acknowledged configurations, the method returns with a true otherwise with a false.

Error()

The Error method is used to terminate the application due to an error. It logs the message that is passed as parameter to the log file and adds a "Error: " prefix to the message. After logging the message, all objects currently in use are deleted and the application terminates using the `exit(...)` function.

run()

This methods is the actual implementation of the message handling. After some initializations, the method enters an infinite loop. The brake condition for the loop is a bool value indicating if the application is still running or not. After the method has entered the loop, it reads a packet out of the `PacketQueue` (see Section 4.7.22) by calling the queues `getPacket()` method. This method suspends the thread as long as no packet is available otherwise it returns the MRNPacketInfo structure containing the packet.

After the `getPacket()` methods return, the returned pointer is checked for validity, since in case the `PacketListener` (see Section 4.7.21) gets stopped it has to return a last packet to release the `getPacket()` method. If the pointer is valid, the packet can be processed. The tag of the packet is used to determine the further actions in a switch statement, even though there currently are only packets with metric values expected there it might be needed to process additional packets in the function. The metric packet is then stored into the database using the `insertPacket(...)` method of the `DBCommunicator` (see Section 4.7.2) object and the loop will start over again.

If the application has to be stopped, the `running` flag will be set to false and the loop will stop its execution. After the loop is stopped, the `run()` method waits for the `PacketListener` thread to stop its execution and calls the `join()` method. The execution of the main thread is suspended until the called thread has joined. When the `PacketListener` has joined, the application resume its execution and exits the program.

4.7.2 DBCommunicator

The `DBCommunicator` is used to write the incoming metric packets to the database. This is done with the `insertPacket(...)` method. To store the metric values, the method is using a prepared SQL statement, that contains the SQL query and placeholders for the actual values. The table format to store the metrics is rather

simple. It stores a unique id, a time stamp, the rank of the host and the metric value pairs for the current sample. The first three values are stored into integer fields whereas the last one is stored into a binary large object (BLOB) field.

Before the metric values can be stored into the database, the metric packet has to be read and the values have to be extracted. The data inside the metric packet is structured. It contains a chunk for every host. These chunks again have a small header, containing the rank of the host and a length value that contains the number of metric values following the header. Subsequently to the header there is a pair of two bytes for each metric value in the packet. The first byte represents the id of the metric and the second byte the current metric class.

The rank and the number of values are stored into separate variables and the bytes are stored as an unsigned char into a `stringstream` for each node in the message. If all metric values for a node are read from the packet, the values are stored into the database using the `execute(...)` method of the `mysqlpp::Querry` object provided by libMySQL++. The execute method expects the parameters in the order of the prepared statement.

4.7.3 FileReader

The `FileReader` call provides the functionality to read the contents of the provided files. It is the base class for the `MetricNetProcFileReader` (see Section 4.7.9), the `MetricMemoryProcFileReader` (see Section 4.7.7) and the `MetricLoadAvgProcFileReader` (see Section 4.7.6) classes.

It implements the read function that is used by all the reader mentioned above to read the content of the according file in the proc file system. The file name has to be provided and the function returns the file content as a C++ `std::string`. Additionally it implements a function to remove the leading spaces in a `std::string`.

4.7.4 MetricConfiguration

This class contains the actual configuration for a dedicated metric. It stores a set of key value pairs that contain the configuration setting. There are three settings that have to be set for each metric, the class count, the intervals for the different classes and the capture interval. For the network and the sensor module an additional setting is required (device for the network and label for the sensor module). Additionally the metric name and the metric id are stored directly. (see Section 4.7.17) object that can store one or more values of the different possible value types (Integer, floating point and string values).

It implements a set of methods to access the setting values. A setter and a getter for the metric id and the metric name, that return the value stored in the object itself. Methods that return the capture interval and the class count directly as integer value. And methods to set and to get the stored setting objects.

4.7.5 MetricController

The `MetricController` class is derived from the `Thread` class (see Section 4.7.23) and is implemented using a singleton pattern so that only one instance of the metric controller is instantiated at any time. To create an instance of the `MetricController` a call to the static function `getInstance()` is required.

To ensure that only one instance is available, the `MetricController` has an instance variable holding the pointer to the instance and an instance flag of type bool to indicate if an instance was created or not. A call to the `getInstance()` function tests if the instance flag is set or not and returns either the pointer to the object instance or calls the constructor to create an instance. If the object has to be created, the instance flag is set to true and the pointer to the object is stored in the instance variable and returned.

The `setReaderMap(...)` method stores the pointer to a `std::multimap` that contains the pointers to the different metric reader objects (see Section 4.2.1) in the `mp_readerMap` variable.

Run() method

If the start function of the `MetricController` is called, the `Run()` method is executed in a separate thread. The `Run` function has to be implemented and is used to capture the metric values based on a timer. After the declaration of the necessary variables the function enters a loop that is only exited when the thread gets stopped.

To have a proper timing value all times used in the class are based on a `timespec` value that is defined in the `time.h` header file and stores the time in seconds and nano seconds. The current time can be set using the `clock_gettime` function which also is defined in `time.h`.

After entering the loop the first operation is to receiving the current time and store it in the `starttime` variable. Then the metric controller iterates over the reader map, that stores the `MetricReaderInfo` (see Section 4.7.14) objects. The `MetricController` decides whether to capture a metric or not based on the `nextRead` value stored in the `MetricReaderInfo` class. The `nextRead` is an integer value and acts as a counter that will be decreased by the last interval

timeout value each time the metric controller is active. If the counter value in the `MetricReaderInfo` object is 0, the metric has to be captured.

The capturing of the metric is done with a call to the `getClass(...)` method that requires a pointer to a `ReaderInfo` object to capture the metric from. It returns an `unsigned char` value that represents the current class of the metric value. The current class is stored into a `MetricPair` structure, containing the metric id and the class. If the class value is different to the value of the last capture interval, the `MetricPair` is added to the `MetricPacket`

After capturing the current metric value, the `nextRead` field in the according `MetricReaderInfo` object is decreased by the last capture interval and the `lastCaptureClass` variable is set to the current class value.

To determine the capture timeout , the current value of the `nextRead` variable is compared to an integer value that stores the time to the next capture interval in the variable `nextinterval`. If the `nextRead` value is smaller then the current value stored in `nextinterval` the value is updated to the new value. After the iteration is finished, the interval in seconds to the next timeout is stored in `nextinterval`.

The `MetricPacket` is send to the front-end and the timeout is determined.

As timer function the thread uses the `pselect(...)` function, which is actually used to determine if a file descriptor is ready for usage. Without passing a file descriptor to observe and based on the fact that `pselect(...)` is a blocking call, it works as a timeout. After `pselect(...)` returns, the loop starts over again.

getClass(...) method

The `getClass(...)` method returns the current metric value class. It calls the `getMetricValue(...)` method of the `MetricReader` object that is stored in the `MetricReaderInfo` object that has to be passed to the method. The `MetricReaderInfo` also contains the class value of the last capture interval.

The value returned by the `getMetricValue(...)` method is of the type `value_t`. The current class is an `unsigned char`. The values are stored in local variables. Then the interval configuration for the metric classes are received from the configuration and stored in a `MetricSetting` object (see Section 4.7.17). At this point all necessary information is available to classify the current metric value.

To classify the current metric value the type of the `MetricSetting` is determined with a switch statement. This is necessary to receive the value either as an integer value by calling the `getIntValue` method or as an float value by calling

the `getFloatValue`.

The method to classify is the same for all possibilities of value pairs (`integer` setting, `integer` value; `integer` setting, `float` value; `float` setting, `integer` value; `float` setting, `float` value) but the implementations are a little different since the values are received differently.

Assuming that a metric class is not changing frequently, the current metric value is tested against the upper bounding of the class interval. Depending on the result the metric value is then either tested against the lower bounding of the class if the metric value is smaller then the upper interval bounding or the upper bounding of the next higher class.

If the correct class is found the loop is stopped with a break and the class value is returned.

4.7.6 MetricLoadAvgProcFileReader

This class implements all functions needed by the the readers of the LoadAvgModule to access the metrics. For a more detailed description see Section 4.4.3.

4.7.7 MetricMemoryProcFileReader

This class implements all functions needed by the the readers of the MemoryModule to access the metrics. For a more detailed description see Section 4.4.2.

4.7.8 MetricModuleConfiguration

This class is used to store the necessary information regarding the configuration of a metric module. The class is defined in `metric-module-configuration.h` and implemented in the `metric-module-configuration.cc` file. It is a container class that stores the module name and the `MetricConfiguration` objects that corresponds to the respective module. The module name is stored in a `std::string` and the objects for the metric configuration in a `std::vector`.

To serialize the object with the boost serialization library it implements a private `serialize` method and declares the `boost::serialization::access` class as a friend class. In the `serialize` method the required fields are added to the archive, that is passed to the method. Similar to the `MetricSection` class it implements a `add(...)` a `begin()` and a `end()` method to either add a new object to the container or receive an iterator of the `MetricModuleConfigurationIterator` type. The iterator is defined in the classes header file.

4.7.9 MetricNetProcFileReader

The `MetricNetProcFileReader` is used by all metric readers of the network module. It is responsible to read the content of the `/proc/net/dev` file that contains detailed information for each network device installed in the system. It implements a function for each supported metric that returns the current metric value. These functions have two parameters, a string to specify the device for which the value has to be returned and the time stamp of the current interval. The time stamp is used to read the file content only once in an interval.

4.7.10 MetricNetReader

The `MetricNetReader` derives from the `MetricReader` class and is needed in the network modules. It is used to bind the reader of the network module to a network device. All reader in the network module derive from this class instead of the `MetricReader` class.

4.7.11 MetricPacket

The `MetricPacket` class is a container class which stores all `MetricInfo` structures generated during a capture interval and the rank of the back-end. A `MetricInfo` structure contains two unsigned char fields to store the id associated to a particular metric and the class for the value of the according capture interval.

Internally the metric packet uses a `std::vector` to hold the `MetricInfo` structures. The rank of the back end is stored into an unsigned integer value.

4.7.12 MetricPair

`MetricPair` is a structure to associate a metric id and the corresponding class value. The structure is used to insert the classified metric value and the corresponding metric id into a `MetricPacket`.

4.7.13 MetricReader

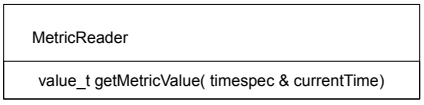


Figure 4.3: The `MetricReader` defines the interface for all metric reader.

The `MetricReader` class is an abstract class that defines the interface for the

back-end to read the metric value from a specific reader. All classes that return a metric value have to derive from this class.

4.7.14 MetricReaderInfo

This class is used by the `MetricController` to determine the necessary information to decide whether a metric value has to be captured or not and to access the according `MetricReader`. Therefore it stores a pointer to the `MetricReader` and the `Metric` class as well as a `timespec` structure with the last capture time of the metric in public variables.

4.7.15 MetricReaderInfoCollection

This is a container class, that contains the `metricReaderInfo` objects that are returned by the `getReader(...)` method of the metric modules.

4.7.16 MetricSection

This class is the root object for the metric configuration. The implementation can be found in the `metric-section.cc` file and the definitions are stored in the `metric-section.h` file. It represents the metric section of the configuration file and stores a `MetricModuleConfiguration` object for each configured module in a `std::vector`. It implements the `serialize` method that is required by the boost serialization library. The method is a private method and therefore it is necessary to declare the `bosst::serialization::access` class as a friend class. The `serialize(...)` method is called by the access object, that passes archive and the archive version to the serialization method. Inside the method all required field are added to the archive.

Additionally there are implementations for a `begin()` and `end()` method that return an iterator of the type `MetricSectionIterator` that is defined in the `metric-section.h` header file and can be used to iterate over all `MetricModuleConfiguration` objects stored in the object.

4.7.17 MetricSetting

The `MetricSetting` class is designed to store one or more integer, float or string values. It uses a boost::variant class which stores the vector with the values. The boost::variant class is a template class provided by the boost library. Once a value is set, the data type is fixed to this value and can not be changed.

The class implements a `getType(...)` methods to determine what data type of the stored value. To return the amount of values hold in the class a `size()` func-

tion is implemented. To get and add values to the object methods for the supported data types are implemented as well.

4.7.18 Module

`Module` is an abstract class to define the method that is used to define a interface for the back-end to access the metric reader contained in the module. To instantiate a `MetricReader` from an instantiated module, the `getMetricReader(...)` method has to be called. The `getMetricReader(...)` has to be implemented inside the specific module.

4.7.19 MRNCommunicator

The MRNCommunicator is defined in the `mrn-communicator.h` file. It implements static methods to send the different packet types using a specified stream. The Stream, Packet Tag and the data are provided by method parameter.

To send the data through the TBŌN the data structure is interpreted as a continuous stream of unsigned char values. This ensures, that any data type can be transferred. Therefore the data that has to be transferred has to be accessible either as array or a string.

The MRNCommunicator needs to determine the length of the data packet as a multiple of 1 Byte. The length of the data stream as well as the pointer to the in memory data structure is then passed to the wrapped `MRN:Stream::send(...)` method. The return parameters are checked and in case of an error a message is written to the log. The function itself returns a bool value to either indicate a success or a failure.

There are three different implementations of the `Send(...)` method. In some cases the monitoring system sends empty packets as trigger for a certain action. In this cases it is not necessary to process any data. Therefor the `MRNCommunicator` just needs to know the tag and the steam.

The first implemented send function simply sends a tag through the stream and is used to send plain commands that do not contain any data.

The second command can be used to send generic packages. The use of the std::string makes it possible to either send real strings but also to send binary information due to the fact that it is not null terminated. This ability is used to send the serialized configuration object to the back-ends. The message interpreted as an array of unsigned char values.

The third method is used to send the metrics for a certain capture interval to the

front-end. It serializes the content of the `MetricPacket` (see Section 4.7.11) and sends it as an array of unsigned char values. To serialize the `MetricPacket` the method determines the size of the packet.

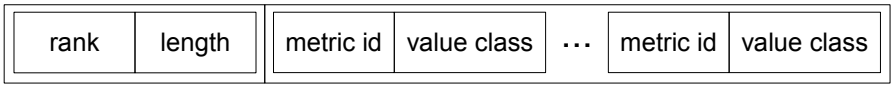


Figure 4.4: The format of the transfered metric packet. The header contains the rank of the back-end and the amount of metrics. The metrics are submitted as value pair, the metric id identifies the metric, and the value class field contains the class of the metric value.

The actual format of the packet send through the network has a header and a body (picture 4.4 depicts the format of the transmitted packet). The header contains two fields, the rank of the back-end and the amount of metric values send in this package. The body contains the serialized `MetricInfo` structures.

After the length is known, the method allocates memory to store the data. Two pointers are used to store the values to the allocated memory area. The first points to the start of the memory area and is used to free it after the packet is send. The second pointer is incremented after writing the corresponding value into the memory about the value size. The rank and the size of the body are stored directly and the `MetricInfo` structures are stored in a loop.

After the content is written into the memory area, the data is send through the stream as an array of unsigned integer values and the memory block for the packet data is freed.

4.7.20 MRNPacketInfo

This structure is a container that contains a MRNet packet and the time when the packet was received. It is used by the front-end receiver thread. The `MRNPacket-Info` object is inserted into the PackedQueue. The times stamp is the time that is inserted into the database.

4.7.21 PacketListener

The packet listener is implemented in the `PacketListener` class and is derived from `Thread` (see Section 4.7.23). It listens to a MRNet stream for incoming packets and stores the packets into a queue. To initialize the `PacketListener` the pointer to the MRNet network object and to the stream has to be passed to the

constructor. A call to the `start()` method starts the thread and calls its `Run()` method, where the actual work is done.

To determine if a packet is available the class uses the event notification provided by MRNet and the `pselect()` system call. The MRNet library returns a file descriptor of a specified type with a call to the `get_EventNotificationFd(...)` method. As the `PacketListener` is interested in data events the event type is `DATA_EVENT`. The returned file descriptor is used with the `pselect(...)` function to determine if packet is available.

The `pselect(...)` function is called inside a loop. The function is used in a blocking mod and only if a packet is available. After the `pselect(...)` returns the file descriptor that caused the return is determined and the according action is taken. There are two possible events that can cause `pselect(...)` to return. Either a packet is available in the network or the listener has to be stopped. If the listener has to be stopped the program is either terminating or reloading the configuration. If a packet is available in the network stream, it is read and sored into a `PacketInfo` object which is then added to the packet queue.

4.7.22 PacketQueue

The `PacketQueue` is a thread safe wrapper to the `std::queue` container class. It implements a function to add content and a function to remove a `MetricInfo`. The class constructor instantiates a mutex that is used to lock the critical section of adding and removing data and a condition variable that is used to.

The add function is secured by a mutex and returns after the packed was added and the condition is signalized to the consuming thread. The get function removes the first packet from the internal queue and returns unless the internal queue is empty. If the internal queue is empty the calling thread is suspended by a call to `pthread_cond_wait` until the condition for an available packet is signalized. If a packet gets available, the consuming thread resumes the execution by returning from the `pthread_cond_wait` call.

4.7.23 Thread

The thread class is an abstract wrapper class to the `pthread` library. It implements a start, stop and a join method and defines a virtual Run() method that has to be implemented in the child classes.

The `Thread` class stores three variables to manage the thread. The `bool` flag running indicates if the tread is currently running or not, the thread ID stored in

a pthread_t variable to hold the thread identifier and a mutex variable to lock the access to the running flag.

The `start` method creates a pthread by calling the `pthread_create` function which requires four arguments. A pointer to the a variable of the type `pthread_t` (defined in `sys/types.h`), a pointer to a `pthread_attr_t` variable, that can be NULL, the pointer to the start routine of the thread and pointer to the thread argument. As start routine a pointer to the `ThreadEntry` function is passed and a pointer to the object instance as argument.

The `isRunning` method can be called to determine whether the thread is running or not. The method uses a mutex to ensure that the access to the running flag is thread safe.

A call to the `stop` method sets the running flag to false to indicate that the thread has to be stopped.

The `ThreadEntry` casts the pointer to the thread and calls the threads `Run()` method.

5 Conclusion

5.1 Evaluation

The implemented system shows that it is possible to significantly decrease the amount of data produced by the monitoring system. The classification of the monitoring data reduces the produced and stored amount of the monitoring data. A run on the XTORC cluster (a 64 node system in the Computer Science and Mathematics Division at the ORNL) produced a database file of 919,680 bytes and an index file of 159,744 bytes for a 4 hour test run. This corresponds to an accumulation rate of ≈ 300 kB/h or ≈ 2.5 kB/interval. For the test run 32 compute nodes and the head node of the cluster where used. The front-end application was executed on the head node and the automatic startup was used. The test run was done considering the worst case scenario (The classes are changing steadily). It included 18 metrics that where sampled at an interval of of 30s. The configuration and topology file used for the test can be found in the appendix (C and C).

To test the ability to recover from IC failures a IC was stopped using the `kill` command. The ability to capture and store monitoring data was not affected by the loss of a IC. As long as there is an adequate number of IC available to handle the connections to all BEs the loss of IC does not disturb the monitoring. On the other hand, loosing an IC leads to a higher load on the remaining ICs since they have to process more packets and handle more client connections.

Killing BE processes did not affect the system either. The processes where removed from the underlaying TBŌN and where simply not delivering data anymore.

The use of a TBŌN also prevents the monitoring system to reach a single systems socket limits by the distribution of the communication layer to the different children in the tree structure of the overlay network.

5.2 Future Work

There are several improvements for the programs that can be implemented in the future. The first thing to improve is the error handling of the back-end and the modules. Currently some failures do not produce an error message and in some rare cases the back-end will have a segmentation fault due to an error.

Since there is currently no way to reintegrate leaf or intermediate nodes when

they where removed from the TBŌN due to errors. To get all nodes back into the monitoring system it has to be restarted. To implement the reintegration it would be necessary to listen on a dedicated port where a back-end process can query the status of the monitoring system and communicate its availability. Normally the topology of the TBŌN is defined in the configuration file and setup during the network creation. But there are several ways to access the topology and add or remove nodes "manually". After the node is added to the network, the communicators and the streams have to be reinitialized. Unfortunately the methods needed for this are not documented in the MRNet documentation. The reintegration of intermediate nodes seems to be more difficult. Since they are usually started by the front-end daemon with the network creation and there is no official way to move child nodes to another parent. The methods to add and remove sub-graphs are all private and an extension of the MRNet library itself seems to be necessary to achieve the designated goal.

Another improvement is to deal with the different time drifts on different leaf nodes. It is inevitable that the local timers on the leaf nodes that are used to determine the timeouts for the capture intervals will drift over the monitoring time. Since the filter wait for all their child nodes to send a metric packet, the node with the biggest time drift will rule the time drift for the entire sub tree and the time drift will accumulate when the packages flow up through the network. To correct this, the intervals have to be monitored in the front-end and when they exceed a threshold an action to adjust the drift will be taken. This can be done by sending a signal to the back-end processes and they will subtract the threshold value from the next timeout.

To be able to manage clusters that are using different host types it would be an improvement to define metric configurations for different host groups. Therefore the configuration would need a separate section to define the different groups and the hosts of the groups. The Metric configuration will need another property to define on which host group(s) the metric configuration should be applied.

If the fault prediction is examined in detail and the important metrics and the according correlations to other metrics are known, it might be possible to move parts of the prediction process to the filter plug-ins and therefore improve the reaction time of a possible predictor even further since the prediction can be calculated closer to the source of the metric values and avoid the latencies that are inevitable during the transport through the TBŌN.

Bibliography

[1] MySQL++ - A C++ wrapper to the C MySQL API.

[2] D.C. Arnold, G.D. Pack, and B.P. Miller. Tree-based overlay networks for scalable applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[3] Jim M. Brandt, Bert J. Debusschere, Ann C. Gentile, Jackson R. Mayo, Philippe P Pébay, David Thompson, and Matthew H. Wong. OVIS-2: A robust distributed architecture for scalable RAS. In *Proceedings of the* 22^{nd} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2008:* 4^{th} *Workshop on System Management Techniques, Processes, and Services (SMTPS) 2008*, Miami, FL, USA, April 14-18, 2008. ACM Press, New York, NY, USA.

[4] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2006*, page 18, Tampa, FL, USA, November 11-17, 2006. ACM Press, New York, NY, USA.

[5] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computing Systems (FGCS)*, 22(3):303–312, 2006.

[6] Elmootazbellah N. (Mootaz) Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 1(2):97–108, 2004.

[7] Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of the* 17^{th} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2009*, pages 252–257, Weimar, Germany, February 18-20, 2009. IEEE Computer Society.

[8] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2007*, pages 1–12, Reno, NV, USA, November 15-21, 2007. ACM Press, New York, NY, USA.

[9] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing and Networking (SC) 2005*, page 9, Seattle, WA, USA, November 12-18, 2005. IEEE Computer Society.

[10] GNU. An Introduction to the Autotools.

[11] HP. Cluster Management Utility.

[12] IBM. Cluster Management Utility.

[13] libconfig. libconfig - C/C++ Configuration File Library.

[14] libtool. GNU Libtool - The GNU Portable Library Tool.

[15] Antonina Litvinova, Christian Engelmann, and Stephen L. Scott. A proactive fault tolerance framework for high-performance computing. In *Proceedings of the* 28^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2010*, Innsbruck, Austria, February 16-18, 2009. ACTA Press, Calgary, AB, Canada. Submitted.

[16] lm_sensors. Ganglia Monitoring System.

[17] lm_sensors. Linux hardware monitoring.

[18] Aroon Nataraj Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet. International Workshop on Scalable Tools for High-End Computing (STHEC 2008), June 2008.

[19] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[20] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[21] Arun B. Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2007*, pages 23–32, Seattle, WA, USA, June 16-20, 2007. ACM Press, New York, NY, USA.

[22] National Center for Computational Sciences, Oak Ridge, TN, USA. Jaguar, 2007.

[23] National Center for Computational Sciences, Oak Ridge, TN, USA. Jaguar Cray XT system documentation, 2007.

[24] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the* 24^{th} *IEEE Conference on Mass Storage Systems and Technologies (MSST) 2007*, pages 30–46, San Diego, CA, USA, September 24-27, 2007. IEEE Computer Society.

[25] OpenIPMI. Open Intelligent Platform Management Interface.

[26] Oracle. MySQL Database Software.

[27] Paradyn Project, Computer Sciences Department, University of Wisconsin. Multicast Reduction Network.

[28] Paradyn Project, Computer Sciences Department, University of Wisconsin. Multicast Reduction Network Library (2.1) Documentation.

[29] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools". In *Proceedings of the ACM/IEEE International Conference on High Performance Computing and Networking (SC) 2003*, Phoenix, AZ, USA, November 15-21, 2003. IEEE Computer Society.

[30] Stephen L. Scott, Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, Anand Tikotekar, George Ostrouchov, Chokchai (Box) Leangsuksun, Nichamon Naksinehaboon, Raja Nassar, Mihaela Paun, Frank Mueller, Chao Wang, Arun B. Nagarajan, and Jyothish Varma. A tunable holistic resiliency approach for high-performance computing systems. Poster at the 14^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 2009, Raleigh, NC, USA, February 14-18, 2009.

[31] Jon Stearley and Adam J. Oliner. Bad words: Finding faults in Spirit's syslogs. In *Proceedings of the* 8^{th} *IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2008: Workshop on Resiliency in High Performance Computing (Resilience) 2008*, Lyon, France, May 19-22, 2008. IEEE Computer Society.

[32] Geoffroy R. Vallée, Kulathep Charoenpornwattana, Christian Engelmann, Anand Tikotekar, Chokchai (Box) Leangsuksun, Thomas Naughton, and Stephen L. Scott. A framework for proactive fault tolerance. In *Proceedings of the* 3^{rd} *International Conference on Availability, Reliability and Security (ARES) 2008*, pages 659–664, Barcelona, Spain, March 4-7, 2007. IEEE Computer Society.

[33] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2008*, Austin, TX, USA, November 15-21, 2008. ACM Press, New York, NY, USA.

A Software Documentation

A.1 Front-end

A.1.1 Command Line Parameters

The front-end is started by running the rasmonfed program. The least two necessary options are the location of the configuration file and the location to the topologie file. Usually it starts the back-end processes autonomously and the back-end executable has to be specified for this mode. If no back-end executable is specified, the front-end assumes that it has to run in the second mode and therefore waits for the number of back-ends to connect that are specified per command line parameter (see below).

The front-end has several command line parameters:

-t this parameter is used to specify the location of the topology file

-c this parameter is used to specify the location of the configuration file

-b this parameter is used to specify the location of the back-end executable file

-C defines the number of back-ends that have to connect

-i this switch is used to start the front-end in interactive mode

A.1.2 Configuration File Format

The configuration currently consists of two sections. One section for the database configuration and another section to configure the metrics. The sections are defined by a section name followed by a colon and are enclosed into curly brackets. The following example shows a section definition:

```
db: {}
```

The section name may consist only of alphanumeric characters, dashes ("‘-"’), underscores ("‘_"’), and asterisks ("‘*"’), and must begin with a letter or asterisk. A section can contain additional sections or key, value pairs of configuration parameters. The configuration distinguishes different value types:

Strings are enclosed in quotes ("‘a string"’)

Integer Values are just digits (1, 2, 3, ...)

Float Values have to have o point in the number (0.1, 11.0, ...)

Arrays are enclosed in box brackets and can contain every of the values above. But the values have to be strict. (["‘a"’, "‘valid"’, "‘array"’], [1, "‘this"’, "‘is"’, "‘not"’, "‘valid"’])

A detailed example can be found in Appendix C

Database Configuration

To configure the database is called "‘db"’, the following parameters have to be provided:

server hostname to connect to

db = the database

user = the user

password = the password

Metric Configuration

The metrics are configured in the "‘metrics"’ section. The metric section contains a sub section for each module. Currently the following 4 modules are supported by the monitoring system: memory, loadavg, network, sensors.

All module sections have to contain the following keys: interval, classes, classIntervals. The "‘interval"’ value is used to specify the capture interval of the according metric. "‘classes"’ contains the number of classes and "‘classIntervals"’ the values of the borders of the different classes. The lower bound of the first and the upper bound of the last class have not to be specified.

The network module requires to specify the network device for the metric as a string. This can either be a single device or an array of different devices. As third option an "‘*"’ defines all available devices (including the loopback device)

The sensor module requires a label field to specify the metric that has to be read. The label is the same, that is used by lmsensors. To receive a list of the available sensor metrics a call to the `sensors` program, that is part of lmsensors, shows all configured sensors.

A.1.3 Topology File Format

The topology file describes the network layout of the TBŌN. It contains all hosts that are participating in the monitoring system for the normal startup (with back-end instantiation). If the back-ends should not be started by the front-end, it only contains the root node and the intermediate nodes.

The structure of the topology file is very simple. Beginning with the root of the tree (the head node) the connections to the nodes in the first level of the tree are described. A configuration line has always the following form:

```
hostname1:0 => hostname1:1 hostname1:2 ;
```

meaning a process on hostname1 with MRNet id 0 has two children, with MRNet ids 1 and 2, running on the same host. A specification line may span one or more physical lines in the topology file:

```
hostname1:0 =>
  hostname1:1
  hostname1:2
;
```

Listing A.1 shows a more complex example with .

Listing A.1: Topology file example with 4 intermediate and 16 back-end nodes.

```
root-node:0 =>
  intermediate-node1:1
  intermediate-node2:2
  intermediate-node3:3
  intermediate-node4:4
intermediate-node1 =>
  cluster-node1:5
  cluster-node2:6
  cluster-node3:7
  cluster-node4:8;
intermediate-node2 =>
  cluster-node5:9
  cluster-node6:10
  cluster-node7:11
  cluster-node8:12;
intermediate-node3 =>
  cluster-node1:13
  cluster-node2:14
  cluster-node3:15
  cluster-node4:16;
intermediate-node4 =>
  cluster-node5:17
  cluster-node6:18
```

```
cluster-node7:19
cluster-node8:20;
```

A.1.4 Topology File Generator

The MRNet library comes with a topology file generator that can create the topology file. The following description is taken from the MRNet documentation [28]:

> When the MRNet test programs are built, a topology generator program, `$MRNET_ROOT/bin/$MRNET_ARCH/mrnet_topgen`, will also be created. The usage of this program is:
> `mrnet_topgen [OPTIONS] TOPOLOGY_SPEC [INFILE] [OUTFILE]`
>
> Create a MRNet topology from the machines listed in `[INFILE]`, or standard input, and writes output to `[OUTFILE]`, or standard output.
>
> The format of the input machine list is one machine specification per line, where each specification is of the form "host[:num-processors]". Note that the first machine listed should be the host where the front-end should be run.
>
> OPTIONS:
>
> -m max-host-procs, –maxprocs=max-host-procs
>
> Specify the maximum number of processes to place on any machine, in which case the number of processes allocated to a machine will be the minimum of its processor count and "max-host-procs".
>
> TOPOLOGIES:
>
> -b topology, –balanced=topology
>
> Create a balanced tree using "topology" specification. The specification is in the format F$\hat{D}$, where F is the fan-out (or out-degree) and D is the tree depth. The number of tree leaves (or back-ends) will be F$\hat{D}$. An alternative specification is FxFxF, where the fan-out at each level is specified explicitly and can differ between levels.
>
> Example: "16$\hat{3}$" is a tree of depth 3 with fan-out 16, with 4096 leaves. Example: "2x4x8" is a tree of depth 3 with 64 leaves.
>
> -k topology, –knomial=topology
>
> Create a k-nomial tree using "topology" specification. The specification is in the format K@N, where K is the k-factor ($\geq$2) and N is the

total number of tree nodes. The number of tree leaves (or back-ends) will be (N/K)*(K-1).

Example: "2@128" is a binomial tree of 128 nodes, with 64 leaves. Example: "3@27" is a trinomial tree of 27 nodes, with 18 leaves.

-o topology, –other=topology

Create a generic tree using "topology" specification. The specification for this option is (the agreeably complicated) N:N,N,N:... where N is the number of children, ',' distinguishes nodes on the same level, and ':' separates the tree into levels.

Example: "2:8,4" is a tree where the root has 2 children, the 1st child has 8 children, and the 2nd child has 4 children.

A.2 Back-end

A.2.1 Command Line Parameters

The back-end is usually started by the front-end and does not need to be started directly. For the second startup mode however it is necessary to start the back-end either by hand or by utilizing a job submission system.

The back-end expects 5 parameters in the correct order to start and connect to the monitoring system. The following command line is used for the back-end startup:

```
rasmonbed <parent_hostname> <parent_port> <parent_rank> <my_hostname>
```

The `parent_hostname`, `parent_port` and `parent_rank` are needed to tell the back-end where to find its parent process in the TBŌN. The front-end creates a file that contains these informations, when it is executed in the second startup mode. The `my_hostname` is the hostname of the host the back-end is executed on and the `my_rank` is used internally by the to identify the back-end.

B MRNet Format Strings

After the % character that introduces a conversion, there may be a number of flag characters. u, h, l, and a are special modifiers meaning unsigned, short, long and array, respectivley. The full set of conversions are:

c	signed 8-bit character
uc	unsigned 8-bit character
ac	array of signed 8-bit characters
auc	array of unsigned 8-bit characters
hd	signed 16-bit decimal integer
uhd	unsigned 16-bit decimal integer
ahd	array of signed 16-bit decimal integers
auhd	array of unsigned 16-bit decimal integers
d	signed 32-bit decimal integer
ud	unsigned 32-bit decimal integer
ad	array of signed 32-bit decimal integers
aud	array of unsigned 32-bit decimal integers
ld	signed 64-bit decimal integer
uld	unsigned 64-bit decimal integer
ald	array of signed 64-bit decimal integers
auld	array of unsigned 64-bit decimal integers
f	32-bit floating-point number
af	array of 32-bit floating-point numbers
lf	64-bit floating-point number
alf	array of 64-bit floating-point numbers
s	null-terminated character string.
as	array of null-terminated character strings.

C Test Configurations

```
# Example ras monitor configuration file
# All names are case-sensitive. They may consist only of
    alphanumeric
# characters, dashes ('-'), underscores ('_'), and asterisks
    ('*'), and must
# begin with a letter or asterisk. No other characters are
    allowed.
# example:
#   name = value ;
# or:
#   name : value ;

# database configuration
db:
{
        server = "localhost";
        db = "Rasmon";
        user = "monitor";
        password = "";
};

# Definitions for the metrics to gather
metrics:
{
  # system metrics
  # this part of the configuration is devided into a
      configuration section for
  # each module ( memory, loadavg, net, sensors, ...)

  # memory related metrics
  memory:
  {
    MemFree:
    {
```

```
      interval = 30;
      classes = 4;
      classIntervals = [ 1024, 2048, 3192 ];
    };
  };

  # load related metrics
  loadavg:
  {
    LoadAvg1:
    {
      interval = 30;
      classes = 4;
      classIntervals = [ 0.5, 1.0, 2.0 ];
    };
    LoadAvg5:
    {
      interval = 30;
      classes = 4;
      classIntervals = [ 0.5, 1.0, 2.0 ];
    };
    LoadAvg15:
    {
      interval = 30;
      classes = 4;
      classIntervals = [ 0.5, 1.0, 2.0 ];
    };
  };

  # network related metrics
  network:
  {
    BytesTransmit:
    {
      interval = 30;
      classes = 4;
      classIntervals = [ 1024, 2048, 3072 ];
```

```
  # specifies a submetric for the devices for wich a
     metric will be gathered
  # * for all devices, a device name or comma seperated
     list of names
  devices = [ "eth0" ];
};
      BytesReceive:
{
  interval = 30;
  classes = 4;
  classIntervalss = [ 1024, 2048, 3072 ];
  devices = [ "eth0" ];
};
      PacketsTransmit:
{
  interval = 30;
  classes = 4;
  classIntervals = [ 1024, 2048, 3072 ];
  devices = [ "eth0" ];
};
      PacketsReceive:
{
  interval = 30;
  classes = 4;
  classIntervals = [ 1024, 2048, 3072 ];
  devices = [ "eth0" ];
};
      ErrorsTransmit:
{
  interval = 30;
  classes = 4;
  classIntervals = [ 10, 50, 100 ];
  devices = [ "eth0" ];
};
      ErrorsReceive:
{
  interval = 30;
  classes = 4;
```

```
    classIntervals = [ 10, 50, 100 ];
    devices = [ "eth0" ];
  };
};

# sensor metrics
# a working libsensors installation is nesassary for these
    metrics
# the metric name can be anny value, the sensors will be
   identified by
# the label field.
sensors:
{
      fan: {
              interval = 30;
              classes = 8;
              classIntervals = [ 100, 250, 500, 1000,
                 1500, 2000, 2500 ];
              label = "fan2";
      };
      V2_5: { # +2.5V:   (min =  +2.25 V, max =  +2.75 V)
              interval = 30;
              classes = 8;
              classIntervals = [ 100, 250, 500, 1000,
                 1500, 2000, 2500 ];
              label = "+2.5V";
      };
      VCore: { # VCore:   (min =  +1.66 V, max =  +1.84 V)
              interval = 30;
              classes = 8;
              classIntervals = [ 100, 250, 500, 1000,
                 1500, 2000, 2500 ];
              label = "VCore";
      };
      V3_3: { # +3.3V:   (min =  +2.97 V, max =  +3.63 V)
              interval = 30;
              classes = 8;
```

```
                classIntervals = [ 100, 250, 500, 1000,
                   1500, 2000, 2500 ];
                label = "+3.3V";
        };
        V5: { # +5V:      (min =  +4.50 V, max =  +5.50 V)
                interval = 30;
                classes = 8;
                classIntervals = [ 100, 250, 500, 1000,
                   1500, 2000, 2500 ];
                label = "+5V";
        };
        VCC: { # VCC:     (min =  +2.97 V, max =  +3.63 V)
                interval = 30;
                classes = 8;
                classIntervals = [ 100, 250, 500, 1000,
                   1500, 2000, 2500 ];
                label = "VCC";
        };
        CoreTemp: {
                interval = 30;
                classes = 8;
                classIntervals = [ 10.0, 20.0, 30.0, 40.0,
                   50.0, 60.0, 70.0 ];
                label = "CPU Temp";
        };
        MBTemp: {
                interval = 30;
                classes = 8;
                classIntervals = [ 10.0, 20.0, 30.0, 40.0,
                   50.0, 60.0, 70.0 ];
                label = "M/B Temp";
        };
  };
};
```

[caption=Metric Configuration File for the evaluation tests on xtorc]

```
node0:0 =>
        node1:0
```

```
            node9:0
            node20:0
            node28:0;
node1:0  =>
            node1:1
            node2:0
            node3:0
            node4:0
            node5:0
            node6:0
            node7:0
            node8:0;
node9:0  =>
            node9:1
            node10:0
            node11:0
            node12:0
            node14:0
            node17:0
            node18:0
            node19:0;
node20:0  =>
            node20:1
            node21:0
            node22:0
            node23:0
            node24:0
            node25:0
            node26:0
            node27:0;
node28:0  =>
            node28:1
            node29:0
            node32:0
            node36:0
            node40:0
            node41:0
            node42:0
```

```
node44:0;
```

[caption=Topology File for the evaluation tests on xtorc]

D Listings

Listing D.1: Ganglia Monitor Output for localhost with Ganglias standard metrics. Additional metrics can be added trough configuration and will generate a new line in the HOST section in the XML output.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE GANGLIA_XML [
   <!ELEMENT GANGLIA_XML (GRID)*>
      <!ATTLIST GANGLIA_XML VERSION CDATA #REQUIRED>
      <!ATTLIST GANGLIA_XML SOURCE CDATA #REQUIRED>
   <!ELEMENT GRID (CLUSTER | GRID | HOSTS | METRICS)*>
      <!ATTLIST GRID NAME CDATA #REQUIRED>
      <!ATTLIST GRID AUTHORITY CDATA #REQUIRED>
      <!ATTLIST GRID LOCALTIME CDATA #IMPLIED>
   <!ELEMENT CLUSTER (HOST | HOSTS | METRICS)*>
      <!ATTLIST CLUSTER NAME CDATA #REQUIRED>
      <!ATTLIST CLUSTER OWNER CDATA #IMPLIED>
      <!ATTLIST CLUSTER LATLONG CDATA #IMPLIED>
      <!ATTLIST CLUSTER URL CDATA #IMPLIED>
      <!ATTLIST CLUSTER LOCALTIME CDATA #REQUIRED>
   <!ELEMENT HOST (METRIC)*>
      <!ATTLIST HOST NAME CDATA #REQUIRED>
      <!ATTLIST HOST IP CDATA #REQUIRED>
      <!ATTLIST HOST LOCATION CDATA #IMPLIED>
      <!ATTLIST HOST REPORTED CDATA #REQUIRED>
      <!ATTLIST HOST TN CDATA #IMPLIED>
      <!ATTLIST HOST TMAX CDATA #IMPLIED>
      <!ATTLIST HOST DMAX CDATA #IMPLIED>
      <!ATTLIST HOST GMOND_STARTED CDATA #IMPLIED>
   <!ELEMENT METRIC EMPTY>
      <!ATTLIST METRIC NAME CDATA #REQUIRED>
      <!ATTLIST METRIC VAL CDATA #REQUIRED>
      <!ATTLIST METRIC TYPE (string | int8 | uint8 | int16 | uint16 | int32 | uint32 | float | double |
          timestamp) #REQUIRED>
      <!ATTLIST METRIC UNITS CDATA #IMPLIED>
      <!ATTLIST METRIC TN CDATA #IMPLIED>
      <!ATTLIST METRIC TMAX CDATA #IMPLIED>
      <!ATTLIST METRIC DMAX CDATA #IMPLIED>
      <!ATTLIST METRIC SLOPE (zero | positive | negative | both | unspecified) #IMPLIED>
      <!ATTLIST METRIC SOURCE (gmond | gmetric) #REQUIRED>
   <!ELEMENT HOSTS EMPTY>
      <!ATTLIST HOSTS UP CDATA #REQUIRED>
      <!ATTLIST HOSTS DOWN CDATA #REQUIRED>
      <!ATTLIST HOSTS SOURCE (gmond | gmetric | gmetad) #REQUIRED>
   <!ELEMENT METRICS EMPTY>
      <!ATTLIST METRICS NAME CDATA #REQUIRED>
      <!ATTLIST METRICS SUM CDATA #REQUIRED>
      <!ATTLIST METRICS NUM CDATA #REQUIRED>
      <!ATTLIST METRICS TYPE (string | int8 | uint8 | int16 | uint16 | int32 | uint32 | float | double |
          timestamp) #REQUIRED>
      <!ATTLIST METRICS UNITS CDATA #IMPLIED>
      <!ATTLIST METRICS SLOPE (zero | positive | negative | both | unspecified) #IMPLIED>
      <!ATTLIST METRICS SOURCE (gmond | gmetric) #REQUIRED>
]>
<GANGLIA_XML VERSION="2.5.7" SOURCE="gmond">
<CLUSTER NAME="my cluster" LOCALTIME="1254766201" OWNER="unspecified" LATLONG="unspecified" URL="unspecified">
<HOST NAME="localhost" IP="127.0.0.1" REPORTED="1254766188" TN="13" TMAX="20" DMAX="0" LOCATION="unspecified"
    GMOND_STARTED="1254766188">
<METRIC NAME="cpu_nice" VAL="0.0" TYPE="float" UNITS="%" TN="13" TMAX="90" DMAX="0" SLOPE="both" SOURCE="gmond
    "/>
<METRIC NAME="cpu_user" VAL="0.7" TYPE="float" UNITS="%" TN="13" TMAX="90" DMAX="0" SLOPE="both" SOURCE="gmond
    "/>
<METRIC NAME="proc_total" VAL="237" TYPE="uint32" UNITS="" TN="13" TMAX="950" DMAX="0" SLOPE="both" SOURCE="
    gmond"/>
```

```
<METRIC NAME="proc_run" VAL="2" TYPE="uint32" UNITS="" TN="13" TMAX="950" DMAX="0" SLOPE="both" SOURCE="gmond"
      />
<METRIC NAME="load_fifteen" VAL="0.04" TYPE="float" UNITS="" TN="13" TMAX="950" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="pkts_in" VAL="0.00" TYPE="float" UNITS="packets/sec" TN="13" TMAX="300" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="swap_total" VAL="9936160" TYPE="uint32" UNITS="KB" TN="13" TMAX="1200" DMAX="0" SLOPE="zero"
      SOURCE="gmond"/>
<METRIC NAME="load_five" VAL="0.11" TYPE="float" UNITS="" TN="13" TMAX="325" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="machine_type" VAL="x86_64" TYPE="string" UNITS="" TN="13" TMAX="1200" DMAX="0" SLOPE="zero"
      SOURCE="gmond"/>
<METRIC NAME="disk_total" VAL="236.061" TYPE="double" UNITS="GB" TN="13" TMAX="1200" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="mem_buffers" VAL="206532" TYPE="uint32" UNITS="KB" TN="13" TMAX="180" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="mem_total" VAL="3556980" TYPE="uint32" UNITS="KB" TN="13" TMAX="1200" DMAX="0" SLOPE="zero"
      SOURCE="gmond"/>
<METRIC NAME="bytes_in" VAL="0.24" TYPE="float" UNITS="bytes/sec" TN="13" TMAX="300" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="load_one" VAL="0.05" TYPE="float" UNITS="" TN="13" TMAX="70" DMAX="0" SLOPE="both" SOURCE="gmond
      "/>
<METRIC NAME="sys_clock" VAL="1254766188" TYPE="timestamp" UNITS="s" TN="13" TMAX="1200" DMAX="0" SLOPE="zero"
       SOURCE="gmond"/>
<METRIC NAME="mem_free" VAL="1225576" TYPE="uint32" UNITS="KB" TN="13" TMAX="180" DMAX="0" SLOPE="both" SOURCE
      ="gmond"/>
<METRIC NAME="mtu" VAL="1500" TYPE="uint32" UNITS="B" TN="13" TMAX="1200" DMAX="0" SLOPE="zero" SOURCE="gmond"
      />
<METRIC NAME="mem_shared" VAL="0" TYPE="uint32" UNITS="KB" TN="13" TMAX="180" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="cpu_aidle" VAL="99.0" TYPE="float" UNITS="%" TN="13" TMAX="3800" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="cpu_idle" VAL="99.0" TYPE="float" UNITS="%" TN="13" TMAX="90" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="cpu_speed" VAL="3391" TYPE="uint32" UNITS="MHz" TN="13" TMAX="1200" DMAX="0" SLOPE="zero" SOURCE
      ="gmond"/>
<METRIC NAME="mem_cached" VAL="1181740" TYPE="uint32" UNITS="KB" TN="13" TMAX="180" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="cpu_num" VAL="2" TYPE="uint16" UNITS="" TN="13" TMAX="1200" DMAX="0" SLOPE="zero" SOURCE="gmond"
      />
<METRIC NAME="part_max_used" VAL="7.1" TYPE="float" UNITS="%" TN="13" TMAX="180" DMAX="0" SLOPE="both" SOURCE=
      "gmond"/>
<METRIC NAME="bytes_out" VAL="0.03" TYPE="float" UNITS="bytes/sec" TN="13" TMAX="300" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="os_release" VAL="2.6.28-15-generic" TYPE="string" UNITS="" TN="13" TMAX="1200" DMAX="0" SLOPE="
      zero" SOURCE="gmond"/>
<METRIC NAME="gexec" VAL="OFF" TYPE="string" UNITS="" TN="13" TMAX="300" DMAX="0" SLOPE="zero" SOURCE="gmond"/
      >
<METRIC NAME="disk_free" VAL="219.277" TYPE="double" UNITS="GB" TN="13" TMAX="180" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="cpu_system" VAL="0.3" TYPE="float" UNITS="%" TN="13" TMAX="90" DMAX="0" SLOPE="both" SOURCE="
      gmond"/>
<METRIC NAME="boottime" VAL="1254230677" TYPE="timestamp" UNITS="s" TN="13" TMAX="1200" DMAX="0" SLOPE="zero"
      SOURCE="gmond"/>
<METRIC NAME="swap_free" VAL="9936160" TYPE="uint32" UNITS="KB" TN="13" TMAX="180" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
<METRIC NAME="os_name" VAL="Linux" TYPE="string" UNITS="" TN="13" TMAX="1200" DMAX="0" SLOPE="zero" SOURCE="
      gmond"/>
<METRIC NAME="pkts_out" VAL="0.00" TYPE="float" UNITS="packets/sec" TN="13" TMAX="300" DMAX="0" SLOPE="both"
      SOURCE="gmond"/>
</HOST>
</CLUSTER>
</GANGLIA_XML>
```