A Scalable Lightweight Performance Monitoring Tool for Storage Clusters

Daniel Bauer and Metin Feridun
IBM Research Zurich Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
Email: {dnb,fer}@zurich.ibm.com

***Abstract*—Distributed infrastructures providing cloud services as well as compute- and storage clusters are notoriously difficult to administer and optimize. Management applications benefit from up-to-date data on system performance. This paper describes the Performance Monitoring and Management System (PMMS), a light-weight and versatile performance monitoring tool that collects hundreds of thousands of metrics per second and delivers this information as time-series in near real-time. At its core lies an in-memory database that scales through federation to large clusters of several hundreds of nodes. Data is collected using sensors which collect and periodically send their metric data to the PMMS collectors. Designed as a component that is embeddable into a larger system, PMMS is light-weight, with little impact on the system resources and it is easy to install and to configure.**

***Keywords*—Monitoring, clusters, distributed systems**

I. Introduction

In order to manage large-scale systems such as the cloud infrastructure, management approaches that can scale and flexibly handle changes in the environment are necessary. Tools need to be distributed and co-exist within the system, sharing resources and yet delivering management functions timely without impacting the system performance. The performance monitoring and management system (PMMS) described in this paper is designed and implemented for the IBM General Parallel File System (GPFS) with the design goals to collect real-time stream of performance metrics from an arbitrarily large-scale, distributed system; to store historical data; and to make the data available to management applications. GPFS is a scalable, high-performance data and file management solution that provides fast, reliable data access to large, distributed, homogeneous or heterogeneous clusters of storage [1].

In the design of PMMS a number of constraints and challenges were addressed:

- PMMS is embedded into the managed system and has to run alongside other system components with memory and cpu constraints, and therefore needs to have a small footprint on resources.
- PMMS handles a large stream of time-series metric data as there are many performance metrics per resource, and there are many resources.
- The collected data is of different types with different collection rates.
- The system is confined to configurable amounts of in-memory and storage resources.
- Installation of PMMS should be simple and adaptive to enable large-scale deployment.

Addressing these challenges has strongly influenced the design of PMMS. A system that has been tested up to 500000 metrics/s using a single PMMS instance was built and is used in IBM GPFS Storage Server (GSS) as well as other GPFS related products.

In this paper we describe that architecture, design considerations and experiences in developing PMMS. The paper is organized as follows. Related work is described in section II, followed by an in depth discussion of the architecture of PMMS in section III. Section IV presents performance measurements on the implemented system. Finally, section V provides our conclusions and planned future work.

II. Related Work

One of the earliest performance monitoring tools is the Multi Router Traffic Grapher (MRTG) and its accompanying Round Robin Database (RRD) [2], [3]. MRTG is widely used to monitor network traffic at router interfaces. It uses a centralized architecture based on SNMP polling and stores the data in fixed-sized archives. These data archives are preconfigured to cover time ranges from minutes to years in decreasing resolution. Data from the archives are directly rendered into time-series graphs, resulting in excellent response times. On the other hand, the lack of drill-down capabilities, and the centralized, SNMP-based polling architecture limit both flexibility and scalability of MRTG/RRD.

Ganglia addresses scalability using a hierarchical aggregation tree [4]. Raw performance metrics such as CPU load per machine are collected at each leaf of the tree. Nodes within the tree aggregate data of their children up to the root node that provides cluster-wide aggregated values. Each tree node uses the aforementioned RRD tool for data storage and visualization. Ganglia offers high scalability and excellent response times since, as in MRTG/RRD, queries return precomputed data from the RRD archives. The same feature severely limits the flexibility of data queries.

The open-source monitoring platform Sensu [5] has a strong focus on health monitoring while also supporting performance metrics. Sensu integrates status reports and events from different tools using the RabbitMQ message-bus [6].

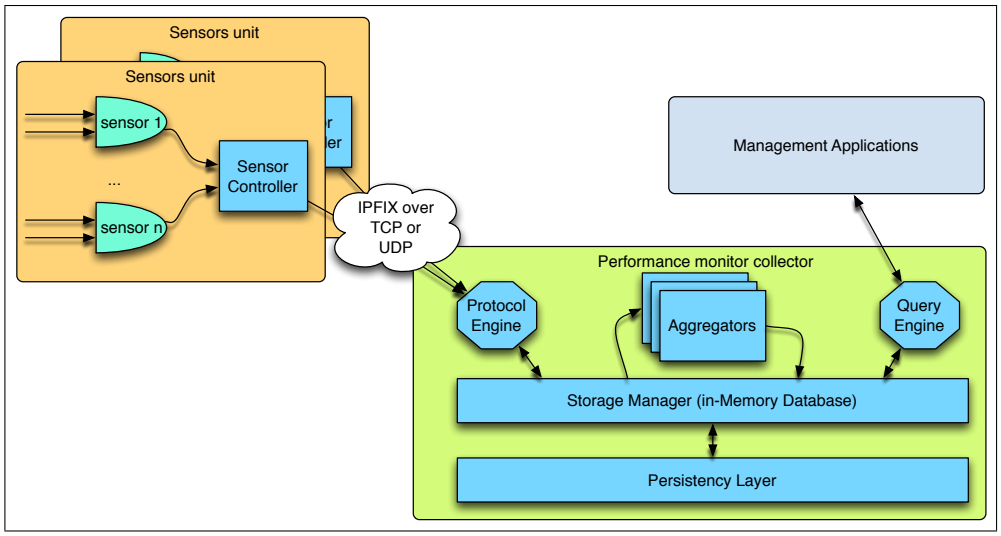

Fig. 1: PMMS component overview.

It uses the Redis in-memory database to store performance data on a central server. This centralized approach limits the scalability of Sensu to medium sized systems. In addition, the rather heavy-weight multi-component setup of the Sensu server necessitates dedicated hardware.

III. Architecture

The PMMS architecture distinguishes two basic components, as shown in Figure 1. Sensor units are responsible for gathering performance data and for sending this data to one or more collector components. The collectors collect and store the data and make this data available for further processing by management applications such as a dashboard. Multiple collectors may be used simultaneously, either for storing the data redundantly or for scaling purposes as described in Section III-E.

Sensor units are memoryless. All performance data is collected and stored in the PMMS collector using an in-memory database in combination with a persistency layer. This design enables the PMMS query engine to provide requested data in near real-time; most queries are executed well below one second, allowing for interactive data exploration. Access to data from the in-memory database is fast enough to allow for versatile and flexible retrieval of the data. In particular, all incoming data is stored in its raw form. Data is processed during query execution which includes converting into the requested time-resolution or aggregating of multiple metrics of the same type etc.

The persistency layer writes the incoming data to files and also populates the in-memory database after a restart. For efficiency reasons, the system is not transactional. In order to reduce the load on the I/O subsystem, data is written in blocks that match the underlying I/O transfer units. This is achieved by buffering the data and writing out entire blocks only.

A. Time-series and Meta-Data

The PMMS collector organizes incoming data into time-series objects. Each time-series object is uniquely identified by a key. The key is hierarchical with as many layers as needed. For example, `zimon1|CPU|cpu_idle` refers to the `cpu_idle` metric from the CPU sensor of node `zimon1`, whereas `zimon1|Network|eth0|net_bytes_recv` denotes the number of bytes received on network interface `eth0` on node `zimon1`. Each part of the key has an associated label such as `net_iface` for the network interface. These labels are used by the SQL-like query language for data selection. Labels together with information about data type, data format and semantic types are meta-data information that is sent by each sensor for each metric. Data type and data format are necessities for interpreting the binary data stream. Semantic information differentiates *absolute values*, *counters* and *delta-counters* and is used during query processing to compute the correct measurement value. An example of an absolute value is the free memory metric and the corresponding aggregation operation is to average free memory. For counters such as the number of bytes transferred over a network interface, the difference between two time points is taken.

B. Memory conservation strategies

As an embedded component, PMMS takes care to efficiently use the available main memory for its in-memory database. Two strategies are employed to conserve main memory. The first one exploits the fact that recent performance data is more valuable and useful than older performance information. The other observation is that performance metrics can be efficiently compressed using loss-free compression methods.

The in-memory database is divided into storage domains. Each domain is a ring-buffer that holds performance metrics for a configurable amount of time using a configurable resolution. This architecture is shown in Figure 2.

The process of time-based aggregation is continuously executed. New data from sensors is added to the ”raw”, i.e. unaggregated domain. Typically, sensors report metrics once per second and that defines the time resolution of the raw

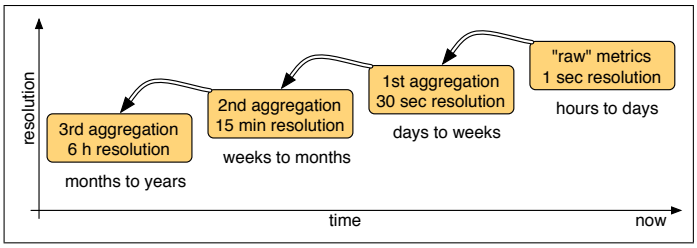

Fig. 2: Time-based Aggregation.

domain. If the domain reaches its configured capacity, the oldest data is pushed out and aggregated into the next aggregation domain. Aggregating metrics of type counter can be done very efficiently through downsampling, i.e. by throwing away measurements. For absolute values, the aggregated value is computed by taking the average of the values over the aggregation time, and for delta-counters, the sum of values over the aggregation time is used.

Both the number of storage domains as well as the aggregation factors are configurable. In the example of Figure 2, one raw domain and 3 aggregation domains with resolution 30 seconds, 15 minutes and 6 hours are defined. Such a setup with four domains is typically used to store performance data over a period of one year. Depending on the application, fewer domains might be used that cover a shorter overall time period and use less memory.

Performance metrics are numerical values that often are confined to a limited range of possible values. In addition, abrupt changes in values are rare. These two properties make delta-compression [7] a good choice for encoding metric values. PMMS implements delta-compression by taking the delta of consecutive metric values and storing the difference using the smallest amount of bits necessary. If values don't change over time, delta-compression effectively turns into run-length encoding. Using this approach, PMMS compresses typical metrics by a factor of 50 or more compared to storing the values in their native binary format. A consequence of using compression is that floating point numbers are not supported. Instead, PMMS supports two different fixed point formats that have a binary representation that is amenable to delta compression.

C. Sensors

The design of the PMMS sensor unit has been driven by the requirement for easy management and installation as well as efficiency in terms of system resources. In order to simplify management and deployment, the PMMS sensor unit is implemented as a single process that executes sensors for various resources. A single configuration file controls the sensors that get started and also the polling period for each sensor. An example configuration file is shown in Listing 1.

The architecture of the sensor unit is shown in Figure 3. At the core is the sensor controller that loads, instantiates and executes the individual sensors as specified in the configuration file. Each sensor runs in its own thread and is periodically activated by the controller. Once activated, a sensor retrieves the performance metrics from its resource. For example, the

Listing 1: Sensor configuration

```
sensors =
{
  name = "CPU"
  period = 1
}, {
  name = "Load"
  period = 1
}, {
  name = "Memory"
  period = 1
}, {
  name = "Network"
  period = 1
  filter = "eth*"
}, {
  name = "Disk"
  period = 5
}
```

CPU sensor on a Linux system reads `/proc/stat` in order to obtain the current CPU time counters. Resources typically provide more than one set of metrics. These metrics are returned to the sensor controller that hands over the data to the IPFIX protocol stack [8] for sending it to the collectors.

The architecture has proven to be robust, flexible and easily extensible with additional sensors. Each sensor runs in its own thread. If a data source is blocked, only the corresponding sensor is affected rather than all sensors in the system. Also, the complexity of timer management and protocol implementation are delegated to the sensor controller. New sensors are easily added to the system as they only have to implement a simple data- and meta-data interface. The latter significantly simplifies the PMMS configuration: each sensor reports the meta-data describing its data, obviating the need for a central configuration mechanism at the collector.

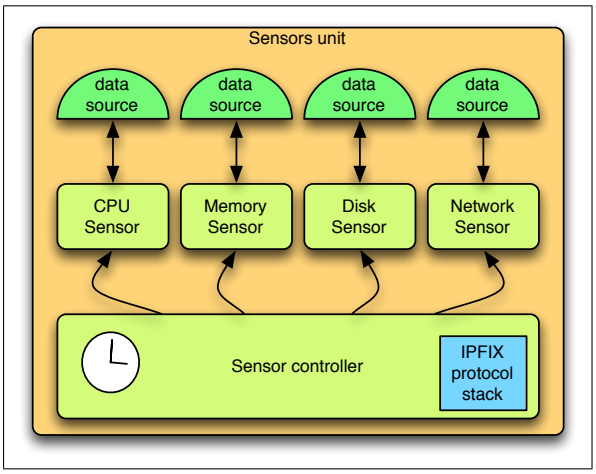

Fig. 3: Sensor Unit Architecture.

D. Data interfaces

PMMS supports a simple, SQL-like query language to enable flexible access to data. Taking advantage of the raw

Listing 2: Query example

```
get metrics cpu_user,mem_memfree from node=zimon[1-2] last 10 bucket_size 60
1:      zimon1|CPU|cpu_user
2:      zimon2|CPU|cpu_user
3:      zimon1|Memory|mem_memfree
4:      zimon2|Memory|mem_memfree
Row     Timestamp             cpu_user   cpu_user    mem_memfree mem_memfree
1       2014-09-01 12:51:00   0.121167   0.309000    553596      751125
2       2014-09-01 12:52:00   0.091667   0.095833    553520      750665
3       2014-09-01 12:53:00   0.298167   0.121167    553482      751082
4       2014-09-01 12:54:00   0.112667   0.286167    553401      750582
5       2014-09-01 12:55:00   0.095833   0.087500    553490      751015
6       2014-09-01 12:56:00   0.292333   0.117000    552874      751254
7       2014-09-01 12:57:00   0.133667   0.281833    552780      751427
8       2014-09-01 12:58:00   0.091667   0.091667    553720      750787
9       2014-09-01 12:59:00   0.297833   0.133833    552786      750682
10      2014-09-01 13:00:00   0.115926   0.301296    553352      750708
```

data and the column-based storage, it enables queries that span metrics across resources and sensor types. Custom sampling intervals for returned data is computed on-the-fly to match application needs, e.g., a graph may initially request courser intervals (e.g., 30 secs), and switches to finer intervals (e.g., 1 second) for a focused view. The query language supports multiple types of time-span expressions; filtering; and collective operations on metrics, e.g., sum. Responses are assembled as a formatted string, or using JSON or CSV formats. The example query shown in Listing 2 returns the metrics cpu_user and mem_memfree from all resources where the node attribute of the key is zimon1 or zimon2, 10 values (rows), each sampled for 60 seconds, for a total time span of 600 seconds, ending at the time the query is received.

To process this query, the query engine determines the key indices for the requested data using the metric names, and filters using the constraint ”from node=zimon[1-2]” to determine the set of matching metric instances. The instance identifiers are then used to send requests for data to the storage manager, which returns data asynchronously for each requested metric, covering the specified time span. The query engine processes the incoming blocks of data and assembles rows of columns, aggregating available data into the specified bucket size, 60 seconds in this example. As rows of response are complete, they are sent back to the requester in chunks until the query is completed.

The queries are sent to a defined socket in PMMS. A set of query APIs are provided in Java, Python and C/C++, to enable programmatic access to data from client applications.

E. Federation

To handle performance monitoring in very large systems, PMMS collectors can be federated, i.e., connected as peers, sharing collection and storage of metrics in the system among them. In a federation, each sensor reports to only one PMMS collector. Meta-data is distributed across the peers and not centralized, which means that the collectors cooperate to respond to a query.

The federation is configurable: given a system with two collectors A and B, A and B are ”full peers” if A is configured to connect to B, and B is a connected to A. A query sent to collector A or B will return the same response. Alternatively, A and B are ”half peers” if only A is connected to B or vice versa. If A is connected to B only, then a query to A will return values from both A and B, whereas a query to B will return values from B only. A full peer federation set-up is shown in Figure 4.

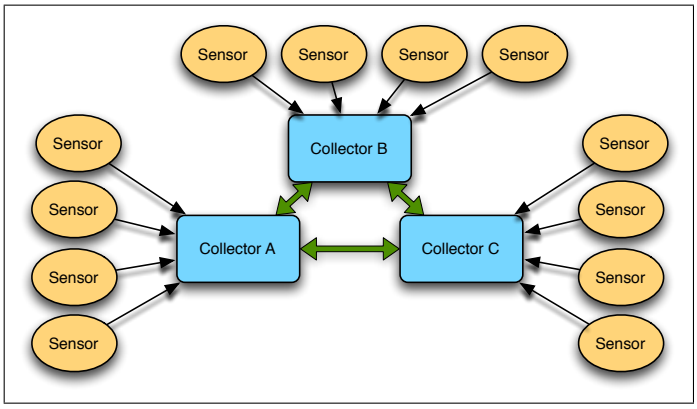

Fig. 4: Federation Setup.

When a collector receives a query, it temporarily serves as the ”master” during the processing of the query, and creates a meta-data request which it sends to its peers. Each peer responds with a list of metrics that it can provide data for. When all the peers have responded, the ”master” collector creates a column-based template for the data it is going to receive. It sends a ”start query” request to all peers which have data for the query. Received data from the peers as well as the ”master” collector are processed using the prepared template for the query and the response sent.

Although not currently implemented, it is also possible to use the federation mechanism to create a hierarchy of collectors, where each level aggregates or consolidates results from its children.

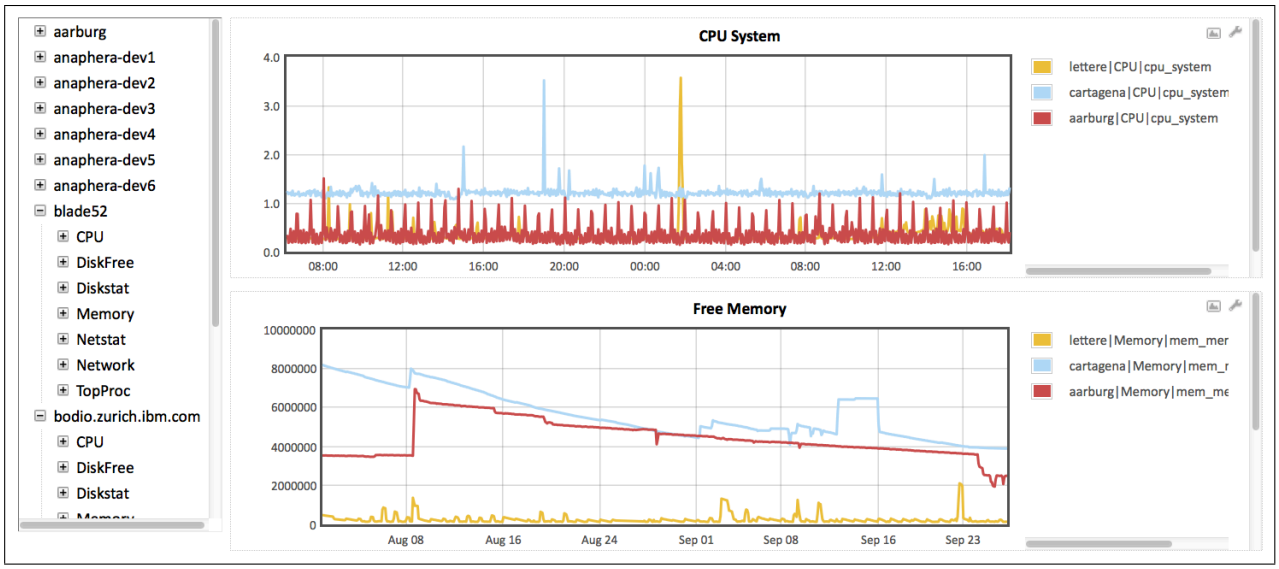

Fig. 5: ZIMon Performance Monitor.

F. Applications

PMMS is designed as an embeddable component that provides performance monitoring data to management applications. To demonstrate the capabilities of PMMS and how it can be integrated using its query facility, a number of management applications have been prototyped.

The ZIMon Performance Monitor is a browser-based dashboard for displaying graphs based on PMMS metric data, obtained using queries sent directly to the socket interface (see Figure 5). The topology view on the left displays all monitored resources and associated metrics using a tree structure: a double-click on a metric creates a new graph, or a metric can be dragged into an existing one, allowing multiple metrics in the same graph. Parameters such as the sampling interval and time span can be specified using a pop-up editor. Graphs can be grouped into views, e.g., a view with graphs showing cpu, memory, network etc. metrics for a given server.

To demonstrate the integration with external tools, a prototype gateway has been developed using the PMMS Python API to enable the Grafana [9] dashboard to collect and display performance metrics.

To experiment with health monitoring concepts, a prototype tool called the PMMS Monitor has been implemented in Python that generates alerts based on rule scripts. For example, it can periodically monitor the disk capacity of one or more servers and issue an alert (email, SMS) when the free disk space falls below a configured threshold. A rule script consists of a *computation*, *filter*, and *listener* sections, and the specification of how often the rule should be executed. A computation defines the metrics to be collected and the operation to be performed on them, e.g., average of the x^{th} percentile. The filter is a threshold on the computed value, consisting of an upper bound and/or lower bound. If the threshold is crossed, the listener sends an email and/or an SMS to specified receivers. New types of computations, filters and listeners can be added to the PMMS Monitor, e.g., a computation that computes the trend over time, taking advantage of the object-based design.

IV. Performance Measurements

The PMMS tool is a component in a larger system that provides a monitoring service on nodes that run (mainly) other services. As such, the footprint of the tool in terms of CPU, memory and I/O consumption must be low enough so as not to interfere with these other services.

Measurements of the deployed system show that PMMS is very well suited to co-exist with other service.

The PMMS sensor unit collects metrics without storing them. Its resident memory size is very modest, less than 2 Mbytes, while running 20 sensor types and collecting 500 individual metric values per second. The CPU requirements for the sensor unit is equally modest, less than 2% of a single CPU core when running on an Intel® Core2® Q9400 with 2.66 GHz. Finally, the average network I/O required by the sensor is less than 100 kbit/s. This network traffic shows small burst of less than 1 second duration that require 280 kbit/s.

We have evaluated the performance of the PMMS collector in a test setup. The test environment consists of 100 nodes, where each node generates and sends 5120 metrics per second to the collector. This setup emulates a medium sized storage cluster. The PMMS collector was running on a 8 core 2.4 GHz node that was equipped with two Intel® Xeon® E5530 processors with 36 GB of memory. It is configured with 3 storage domains, each with 500 MB of memory. As the test results show, a single collector is sufficient to process and store the accumulated performance metrics of the cluster.

The first evaluation investigates the CPU load, with the results depicted in Figure 6.

The graph shows the outcome of an experiment that was running for 3 minutes. At the beginning of the experiment, the cluster nodes are starting to send performance data to the collector. The blue, straight line shows the number of metrics

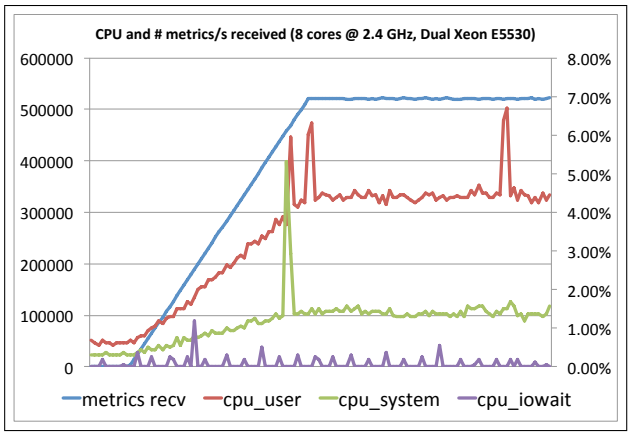

Fig. 6: CPU usage by received volume.

that the collector receives per second. It reaches a plateau with over 500'000 metrics per second after all the nodes have been started. The graph also shows the CPU time on the collector node. Initially, the system was idling with the CPUs spending less than 1% o in system- and user-mode and almost no time for doing I/O. This time increases to about 4.5% for user-mode activities and 1.5% system-mode activities, whereas the I/O activity doesn't change. This demonstrates the frugal use of CPU by the PMMS collector, allowing it to be co-located with more CPU intensive applications.

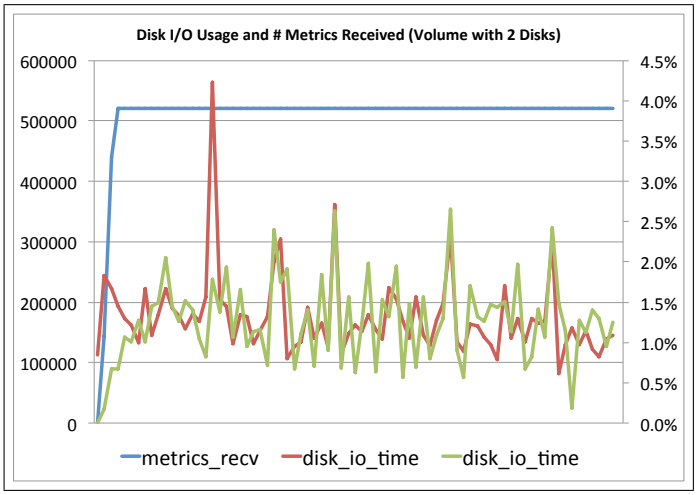

Fig. 7: Disk I/O usage by received volume.

CPU is not the only resource that can become the bottleneck. We have also investigated the disk I/O usage of the collector. Figure 7 shows the results of an experiment that was running for 30 minutes, using the same setup as the CPU experiment. The blue, straight line shows again the number of metrics per second received by the collector. This number increases to over 500'000 at the beginning of the experiment. The other two lines show the amount of time that the disk I/O channels are busy. The PMMS collector is using a filesystem that is stored on two physical disks. Both disks are initially only very lightly loaded. The time that the channels are kept busy increases slightly, with a peak of 4% for one of the disks and an average of around 1.5%. Clearly, the disk I/O performance required by a PMMS collector is also very modest, making it suitable to be executed along other services.

In addition to disk I/O performance, the load on the network also has to be considered. The load on the network is linear with the number of metrics sent per second. The 500'000 metrics/s require 68 Mbit/s network capacity or 6.8% of a 1 Gbit/s link. For todays systems that have multi-GBit/s links, this again translates into a very modest use of the network resources.

V. Conclusion

We have designed and implemented PMMS to meet the requirements set for it in terms of performance and resource footprint. Modular component design, decoupling of the PMMS collector from the specifics of the sensors and other similar design decisions have led to a versatile system that is easy to expand.

We are presently extending the PMMS to address a new set of challenges:

- scalability: PMMS needs to operate in environments with more than 1000 nodes. At this scale, the challenge is not only the sheer handling of the massive amount of data, but also the interpretation of the performance data in an efficient and meaningful way. For example, embedding management analytics into the PMMS may not only reduce the amount of data, but also help detect problems early.
- resilience: In a large deployment, it will be necessary to automate the configuration and control of the monitoring system and to effectively handle failures in PMMS components.

References

[1] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, 2002, p. 19.

[2] T. Oetiker and D. Rand, "Mrtg: The multi router traffic grapher." in *LISA*, vol. 98, 1998, pp. 141–148.

[3] T. Oetiker. (2005) Rrdtool. [Online]. Available: http://oss.oetiker.ch/rrdtool

[4] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[5] (2014) sensu - the open source monitoring framework. [Online]. Available: http://sensuapp.org

[6] (2014) Rabbitmq - messaging that just works. [Online]. Available: http://www.rabbitmq.com/

[7] (2014) Delta encoding. [Online]. Available: http://en.wikipedia.org/wiki/Delta_encoding

[8] B. Claise, B. Trammell, and P. Aitken, "Rfc 7011: Specification of the ipfix protocol for the exchange of flow information," Sep 2013.

[9] (2014) Grafana - an open source, feature rich metrics dashboard and graph editor for graphite, influxdb and opentsdb. [Online]. Available: http://grafana.org