UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

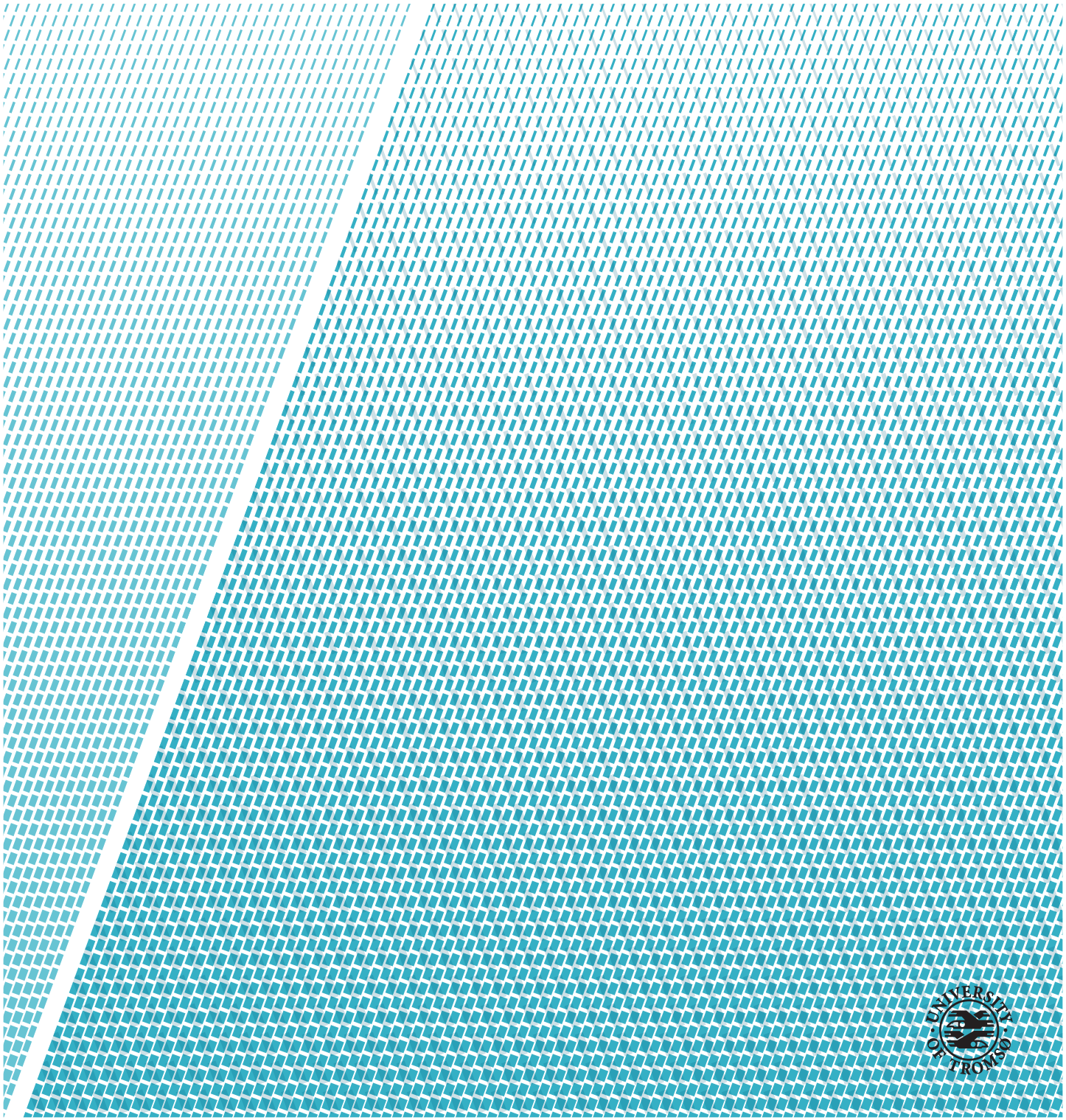
AutoMon

Automatic monitoring and problem detection for distributed systems

—

Magnus Wikstad

Master thesis in Computer Science [INF-3990] – May 2016



Dedicated to those whom I can always depend upon

"And so, does the destination matter? Or is it the path we take? I declare that no accomplishment has substance nearly as great as the road used to achieve it. We are not creatures of destinations. It is the journey that shapes us. Our callused feet, our backs strong from carrying the weight of our travels, our eyes open with the fresh delight of experiences lived."

——Brandon Sanderson, The Way of Kings

Abstract

When working with distributed systems, detecting faults can be a difficult task, as abnormalities isn't necessarily immediately evident by warnings or system crashes. This is especially true with subtle faults, such as variations in performance of a running program, it is not necessarily its own fault, but could rather be from a different source, somewhere in the cluster, using a lot of resources (CPU, IO, etc.), thereby causing other programs to perform sub-par compared to earlier executions.

These types of problems won't necessarily be detected by regular cluster monitoring tools, as these only look at cluster metrics, or by distributed debuggers, as these only monitor specific programs, and thus won't find the cause for the degraded performance if it comes from a different source.

As the usage of distributed systems is becoming more common amongst those without an intimate knowledge about these systems, being able to quickly inform the user about any faults or abnormalities, would be a great improvement on their efficient use of the system. It would additionally be a great help to developers, as they could easily get their programs performance data without implementing specific procedures for the task, thus simplifying the development of new distributed software.

This thesis is looking to discover if the system, and process, information attainable from each nodes operating system, is enough to detect abnormal operation. This is approached by creating a prototype system that collects this information from the cluster, and doing analysis on the data during runtime to check for faults.

The achieved system is capable of collecting large amounts of data from the cluster, storing it, and doing some rudimentary analysis on the data. While leaving most of the clusters resources free for its computations. This shows that it is possible to create a low resource cluster monitoring tool, that collects large amounts of system data, with high frequency, from each of the nodes, and analyze the data.

Acknowledgements

I first would like to thank my advisor, Associate Professor John Markus Bjørndalen and co-advisor, Professor Otto Anshus, for the guidance and encouragement when I needed it. And everyone at the department of computer science for making me feel so welcome throughout my studies, especially Svein Tore Jensen and Jan Fuglesteg for all the assistance with my studies, it has been good to know I can turn to you when I have had questions about anything regarding my studies. And lets not forget all my fellow students, for the discussions, ideas, and company these last years.

I would like to thank my family for being there for me, especially my parents for giving me a place to sleep, making sure I don't eat the same thing every day, and that I keep moving forward.

My thanks also go out to my D&D group for helping me unwind from real life, and give me afternoons where wonderment is commonplace, and magic a way of life.

Christian Haugen for all the motorcycle trips, may we have many more. For the food, company, and for being a good friend.

Kaffebaren $C_8H_{10}N_4O_2$ for the friendly welcome, for keeping me with a supply of $C_8H_{10}N_4O_2$ to help me concentrate on my studies, and letting me taste a lot of good coffee, and some not so good.

Martine Posti for your kindness, the warm smiles, and the heart filled cups of coffee. You have helped me more than you could ever know.

Rune Olsen for all those bike rides, parties, stupid things, and everything else we have done together since childhood. For helping me battle through tough times, and enjoy the good. You will always be invaluable.

My motorcycle, for bringing some adventure into my life when the weather allows it. And to all those I have inadvertently forgotten. Thank You

Contents

List of Figures

List of Listings

List of Abbreviations

API application programming interface

CCA canonical correlation analysis

CPU central processing unit

GB gigabyte

HTTP hypertext transfer protocol

IO input/output

JSON javascript object notation

KB kilobyte

LAN local area network

MB megabyte

MPI message passing interface

OS operating system

procfs proc filesystem

RAM random-access memory

RAPL running average power limit

REST representational state transfer

WAN wide area network

1

Introduction

Clusters of computers forming a distributed system is a widely used alternative for creating parallel computing systems, this is a highly scalable and cost effective way to create such a system. With these large systems comes an increased complexity for the users, this makes it necessary for any system, with maybe the exception of very small clusters, to have a way to monitor the health of this system.

A common approach for this is to have a monitoring system for the cluster, such as ganglia[1], or HP Insight Cluster Management Utility[2], that is responsible for collecting data such as total central processing unit (CPU) and memory usage for each node in the cluster, and present it to the user. This makes it easier to compare the overall usage of each of the nodes in the cluster. There are issues with these systems however; they usually work with a slow heart beat for their updates (thus it is possible that short term system abnormalities goes undetected), another problem is that they don't normally give any data on processes. Some systems do however allow the user to add additional data points to be monitored, ganglia gives the possibility for user-defined metrics that can represent arbitrary state, but here every metric characteristic must be explicitly defined. But then you have the problem of defining all the necessary metrics that could be of value, something that can be difficult to know beforehand.

There are systems that monitor specific programs, either with or without including host system metrics. They vary in whether or not they start and stop at the

same time as the process they are monitoring, or if they have a server running continuously. These normally only store the collected data, and has an interface for the user to get the current or historical data from the database[3, 4, 5].

The systems with a server running continuously can have agents collecting system metrics, in addition to getting data from the monitored program through an installed library that allows it to connect and deliver its own usage metrics to the server.

Although it is vital to a systems administrator to make sure the system as a whole is working as it should, this is not necessarily the main concern of someone using the system, they generally only care about one process, or a small group of processes, depending on their use of the system. For the task of checking on the health of a single distributed program, one runs into the same problem as with checking on the system as a whole, it gets very time consuming if it's not automated in some way.

One could for instance check every node individually to make sure a program is running as it should, or get information from tools used to run the program on the cluster, such as ansible[6]. Checking every node for errors or abnormal operation is time consuming and can be difficult, even to users familiar with the system. While relying on tools like ansible only gives you information about a run at certain intervals, and only gives information about whether or not the program is running, not the correctness of the program.

Something that must be known however, is what measurements is normal for a well behaved system where everything is working as it should. This would of course differ based on what the distributed system is doing at any given time. A high CPU usage when the system is idling, or no usage when doing a CPU intensive calculation, would f.ex. be indications of abnormalities, as this isn't an expected metric for the use case. This same goes for every aspect of the metrics monitored, high memory usage in some program that normally use only a few megabyte (MB) could be an indication of abnormal behavior.

Thus to detect strange behaviors in a system, it is necessary to know how the metrics differ from normal behavior, this is of course difficult to know. For example if we know how a system normally behaves when running nothing but one single program, lets call this process A, and we also know how it should be when running only process B. How can we know the normal behavior of the system when running both of these processes simultaneously.

If only system metrics was monitored, like in general cluster monitoring tools, this would be a very difficult question, but if we also monitor the metrics of each of these processes, we still know how each individual process should behave,

even with both running at the same time, although we don't necessarily know how the system as a whole will behave. This lack of knowledge is less of an issue when knowing how all components of the system should behave, as it is still possible to detect abnormalities in each of these processes.

1.1 Problem Statement

This thesis is looking to design and implement a distributed service that collects and analyses resource usage from all computing nodes, and their running processes, to detect abnormal operation in parts of a distributed program running on several nodes. And give a warning to the user about any abnormal process or node behavior. This challenge can be divided into two categories: cluster monitoring, and usage analysis.

As the operating system (OS) keeps data on all resource usage, and every running process, it is easy to get large amounts of data from the OS. It is however necessary to keep this data for much longer to be able to do any analysis on the data. This means that the data must be aggregated, and stored while running, and with the large amounts of data, it is necessary to use as little space as possible while still keeping all the important data from the cluster.

The analysis of both the system as a whole, and of every process on the system, relies on the continuous detection of any deviation from the normal resource usage pattern of the specific process or system node. To achieve such an analysis it is necessary to find out which of the collected data fields are relevant for detecting abnormal operation from the normal, as well as to find out if there is additional data that would improve this detection.

1.2 Contributions

The contributions of this thesis are:

- The architecture, design and implementation of a Prototype for monitoring a distributed system, with a rudimentary ability to detect faults within this system.
- Analysis of data needed for the detection of abnormal behavior.
- Experiments showing that it is possible to collect large amounts of cluster

data with a low resource usage.

1.3 Limitations

There are some aspects not taken into consideration in this thesis

- There was no focus on security, this was done to limit the scope of the thesis. As access to the cluster network, on which AutoMon is running, normally is limited, there is still some security to the system.
- Every connecting monitoring-agent is assumed to be part of the same cluster.
- The usability of a system over a wide area network (WAN) has not been tested.
- There is no automatic discovery of the AutoMon server, thus the monitoring-agents have to know the IP address of the server at startup.

2

Background and Related Work

This chapter gives an introduction to cluster monitoring, and presents existing cluster monitoring systems, before looking at ways of analyzing large amounts of collected data; in order to detect any deviation from expected metrics.

2.1 Cluster Monitoring

Traditional cluster monitoring systems generally follows a client-server architecture showed in figure 2.1, where there is some form of agent on each node that obtains some system information. The information is then sent to a server part for monitoring. The server then aggregates, stores, and visualizes the cluster data to the user. The data can be collected from the OS, especially on an Unix-type system where it can be easily gathered from the proc filesystem (procfs), it might also be possible to obtain this information from middleware systems and/or applications. This data is collected at certain intervals, that normally span several seconds to keep the overhead as low as possible, so it doesn't interfere with anything running on the cluster[7, 8, 9].

The data collected is generally only systems wide data, like average CPU usage over the last 5, 10, and 15 minutes, memory usage, and net usability as standard,

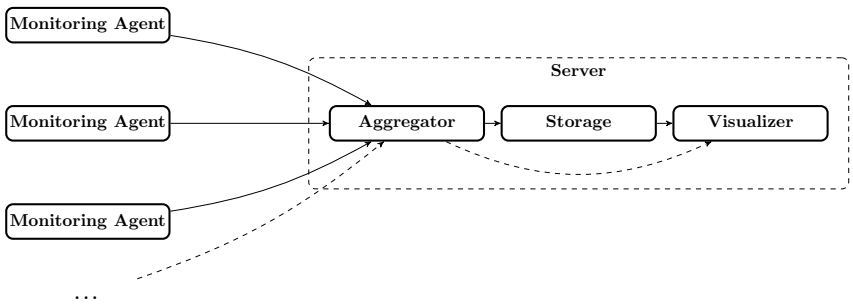


Figure 2.1: General architecture for traditional cluster monitoring systems

sometimes with the possibility of having the user add specific measurement points to the collections. By only collecting these few points of data, the network usage of the system is kept at a minimum, while at the same time giving the user enough data for them to see if the cluster is healthy.

Normally no processing of the data takes place on the nodes, the agents only collect information. This behavior is usually explained by the need to reduce the impact the agents has on the distributed system. There is typically also a predefined fixed configuration, set at start, for where the data is to be sent. For large clusters the monitoring system is set up as a tree-like structure, where each agent sends information to aggregates, that then sends the data in a more compact form upwards in the structure. To cope with failure, the data from any point in the structure can be sent to multiple places, thereby getting redundancy in the system.

Ganglia uses a similar architecture that is comprised of two parts, their difference is that the daemons running on each node of the cluster don't send their collected data to a server node, they instead use multicast, and send it to every node of the cluster so that every node knows the status of the entire cluster. Then the server can poll for the data from any one node, thus by polling several nodes, they achieve redundancy[1]. This can also be used for clusters located on a WAN, where they build up a hierarchical monitoring tree using several of their aggregating server parts[10].

The trend of the development of these types of systems seems to be focused mostly on scalability[11, 12, 13, 14] (Both Van Renesse et al.[12] and Clough et al.[13] use a hierarchical system, and with it claim to be able to handle tens of thousands of nodes efficiently. Zhan et al.[11] also presses this as an important piece of their system). This is of course a good thing, distributed systems are rapidly growing in size, and thus the task of monitoring the system and keeping it healthy, increases in size and complexity. And being able to use the same system for this task whether the cluster is comprised of 5 or 5000 nodes, while

still maintaining usability, and low overhead, is an advantage. However, as much as this is very helpful for the system administrators, it doesn't necessarily help the users of the system, as they don't get any information about how their specific tasks are running.

There are some systems that target application monitoring. That however differs from system monitoring in that the collection rate has to be much higher, resulting in the volume of collected data to be greatly increased. A common feature of application specific monitoring systems, is that they are intended for monitoring specific jobs, thus the execution time is the same as the job it is monitoring, and much of the analysis is done afterwards [15].

2.2 Analysis

There are some that tries to automate the process of fault detection on a distributed system[16, 17]. They employ a combination of statistics and supervised detection to automate the process of detecting faults on a distributed system (Chen et al. introduces a concept of using canonical correlation analysis (CCA) to decide whether variables should be monitored supervised or unsupervised[16]).

Automating this process is of course a logical step, as the size of the monitored systems increase, the frequency of the collections, and the amount of collected data increases, the analysis of this data becomes a big data problem in and of itself. As such is impossible for any one person to do alone, especially for real time analysis.

Tang et al.[17] combines the use of an analysis of historical alerts and incidents, with a rule-based learning algorithm with rule complexity criteria to generate a set of predictive rules. Potential monitoring conditions are then built upon these rules, if any degradation of the system's vital signs (defined by acceptable thresholds or monitoring conditions), the issue is flagged and sent to supporting personnel. The resolution of this issue is then used to update the system's conditions. Thus creating a system that becomes more accurate, regarding which issues are flagged and sent to supporting personnel, as time passes.

2.3 Related Work

Ganglia's[1] multicast communication between monitoring-agents withing a cluster, gives every agent the metrics for the entire cluster, thus allowing the

server to get all the metrics by polling one agent. AutoMon instead has each monitoring-agent push data directly to the server. Another difference is in the server-part, where ganglia visualizes the data to a user. AutoMon does analysis on the data instead, and gives a warning to the user if some abnormality is detected.

The monitoring framework by Zhan et al.[11] uses collectors, called end hosts, that gather metrics and send feedback messages to a coordinator, that then forwards the data to a feedback server. This is similar to AutoMon except for the additional step of the coordinator. A monitoring platform communicates with these feedback servers periodically to retrieve available aggregated measurement data, for analysis and visualizing in real-time. This is something the server takes care of in AutoMon. Both systems uses hypertext transfer protocol (HTTP) for all communication. Messages in the framework are encoded by administrator needs, while AutoMon uses javascript object notation (JSON).

2.3.1 Hierarchies

Astrolabe[12] use a peer-to-peer protocol, with a *zone* hierarchy structure. A *zone* is either a host or a set of non-overlapping *zones*, thus creating a tree-like structure. Each node has an agent that collects information, and also act as a web server, these agents learn about other *zones* using a gossip protocol. Using this structure, summaries of the metrics are created using SQL queries to get on-the-fly aggregation. This way it is possible to gather, disseminate, and aggregate information about the *zones*.

Panopticon[13] like Astrolabe[12] use a hierarchy of agents, that run as daemons on each node, in a tree-like structure. These agents observe and record host system metrics, and provide them to other nodes on request. This way the information is propagated up the tree, recording the route within the data. Thus the root node receives all the data, while only having to communicate with a few child nodes. When running experiments of Panopticon, they used a similar small set of metrics as ganglia[1]. Between the agents a client-server binary protocol is used. While a web service interface with REST and JSON is used between the root node and the storage and retrieval system. They use a 5 minute time granularity, which is the same as many other systems.

Both of these differ greatly from AutoMon, especially in that the use a hierarchical tree-like structure, and aggregate data upwards to a root node, instead of direct communication between client and server. And they also have slow collection, and only collect host metrics, so that messages between nodes are very small in size. Because of their dedication to scalability, both claiming to be

able to handle tens of thousands (possibly up to millions) of nodes simultaneously, thus they have to give up on fine time granularity and complete system metric collection that AutoMon has.

3

Architecture and Design

AutoMon is being developed to run under Linux OS, so whenever 'operating system' is mentioned, this refers to a Linux-kernel OS. AutoMon is a distributed monitoring system, composed of collector agents running on each node of a cluster, these send messages to the analysis-engine server that processes the information, see figure 3.1.

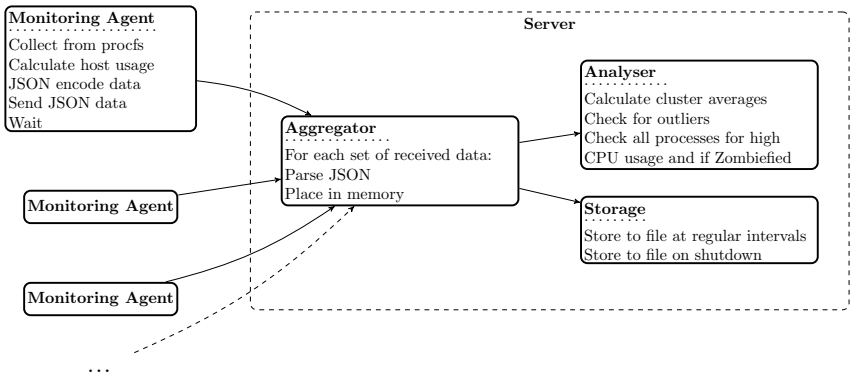


Figure 3.1: AutoMon architecture

3.1 Collector

The AutoMon collector is the component that runs on every node of the cluster. The collector runs on an interval timer, at the start of each of these intervals it collects metrics about the OS, and every process running on the OS. All the collected data is then converted to JSON before being sent to the analysis-engine, in a push-based fashion. The collector then waits until the start of the next timer interval.

3.1.1 Metric Collection

All the data AutoMon collects is from procfs. More specifically the OS metrics is gathered from */proc/stat*, */proc/net/dev*, and */proc/meminfo*, and for each process(PID) currently executing, it is gathered from */proc/PID/stat*, */proc/PID/statm*, and */proc/PID/io*.

As small calculations are done on the data, not all CPU fields are sent. Instead, for each CPU entry in */proc/stat* (this will be whole CPU plus every CPU thread, see listing 3.1), the fields are added to make the total ticks, and using this the usage percentage since last collection is calculated. So for each CPU field the usage percentage is sent, in addition to the idle and total CPU ticks.

Listing 3.1: Example */proc/stat* file excerpt

```
cpu  25220 1 3887 3316908 2450 0 5 0 0 0
cpu0 5057 0 1250 410265 1319 0 1 0 0 0
cpu1 2236 0 258 415867 315 0 1 0 0 0
cpu2 6150 0 938 411482 252 0 2 0 0 0
cpu3 916 0 161 417502 184 0 1 0 0 0
cpu4 3962 0 516 413836 220 0 0 0 0 0
cpu5 1193 0 152 417260 43 0 0 0 0 0
cpu6 4982 0 422 413052 69 0 0 0 0 0
cpu7 722 0 188 417639 45 0 0 0 0 0
```

From the */proc/net/dev* file, 1st and 9th number fields are collected, see listing 3.2, these are for total bytes received and sent respectively. These values are stored in the collector, and are used for calculating the total network traffic in kilobyte (KB) since the last collection time. This total network traffic is what is sent to the analysis-engine.

Listing 3.2: Example */proc/net/dev* file excerpt

```
eth0:       0     0    0    0    0    0    0    0        0     0    0    0    0
0    0    0
   lo: 43412   460    0    0    0    0    0    0    43412   460    0    0    0
0    0    0
```

The memory metrics collected is the *MemTotal*, *MemFree*, *Buffers*, and *Cached* fields from the */proc/meminfo* file, see listing 3.3. The *MemTotal* field is of course the total memory available to the system, the three other fields combined are what is free memory (possible to allocate). Using these metrics the memory usage percentage is calculated, and it is only this calculated metric that is sent to the analysis-engine.

Listing 3.3: Example */proc/meminfo* file excerpt

```
MemTotal:       16348432 kB
MemFree:        13830324 kB
MemAvailable:   14623324 kB
Buffers:           69932 kB
Cached:          1155036 kB
SwapCached:            0 kB
Active:          1490204 kB
Inactive:         752820 kB
```

3.1.2 Data Transfer

AutoMon uses a push-based data transfer for sending the collection metrics from the collectors to the server, this is useful because it ensures that collected data is always ready when the transfer happens. The metric data sent to the analysis-engine is encoded as JSON.

3.2 Analysis-Engine

The AutoMon analysis-engine acts as the system server, in that this is what aggregates, stores, and analyses the data received from the clients. Any connections to this server is done over HTTP, through a representational state transfer (REST)ful application programming interface (API), with metric data being encoded as JSON.

3.2.1 Interface

The collectors use a RESTful HTTP API to send the data to the analysis-engine. This includes the name of the host that sent it, see listing 3.4.

Listing 3.4: HTTP header for data transfer

```
POST /raw/HOSTNAME HTTP/1.1
Content-Length: X

MESSAGE-BODY
```

By using HTTP as the connections between the collectors and the analysis-engine, there is no disruption to the rest of the system if one or more collectors were to fail.

This also makes adding and removing nodes from the system during runtime very easy, as it only depends on a client sending data to the server, there is no dependency on collectors sending any data.

3.2.2 Aggregation and Storage

The received JSON data is first parsed, then it is filtered and stored in a memory structure, using a hierarchical formation split on nodes and processes, see figure 3.2.

This whole structure is at regular intervals, and at shutdown, stored to a file on disk. Regularly saving the structure to disk makes sure that not all data is lost, if for example the server crashes, and thus is unable to write the contents to disk.

The filtering is done for each data point, the received data is compared to the last recorded value from the same node. The newly received data is only stored if there is no previous data, or if it is different from the preceding data point, all data is stored with a time stamp, that way it is possible to see how long a value remained unchanged.

The time stamp used when storing the data is generated by the server, so it corresponds with the time the data was received, not the time it was collected, this prevents server error caused by differing times on the cluster nodes.

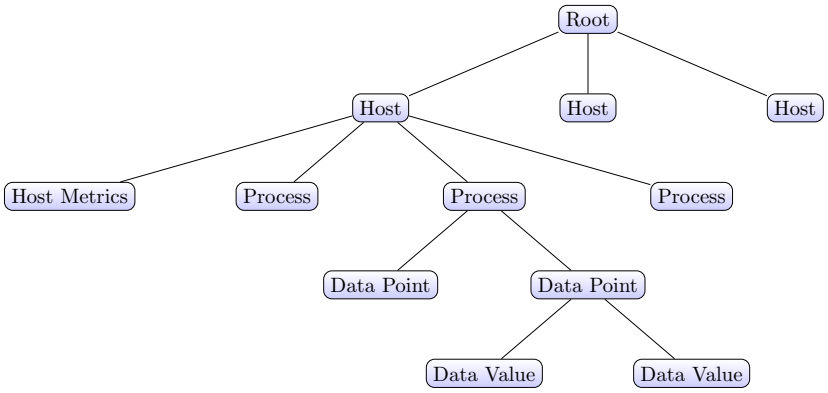


Figure 3.2: Memory structure

3.2.3 Analysis

When analyzing data, the first thing done is to calculate the cluster averages for each node, using this information along with each nodes metrics, it is determined if any nodes have values that differ more than a specific amount from the average. The nodes that fall into this category are reported to the user.

For each process, the state is checked, to see if any has become a zombie. The CPU usage is also checked, and logged if it exceeds a specific amount.

4

Implementation

AutoMon is implemented in C to run under Linux.
Each node in the test cluster runs a 64-bit version of the Ubuntu 14.04 LTS with Linux kernel 3.13.

The implementation of the prototype was done in a circular fashion. Based on the idea the architecture was decided, and then each part of the system were designed and implemented, then experiments were run to test that it was working, and performing as expected.

Each time a problem was detected, f.ex. not working as it should, or having very bad performance, the process was reiterated, improving the design and implementation, before redoing the experiments, see figure 4.1.

4.1 Collector

When gathering data from the procfs, there is always the chance that user permission for the program is not set correctly, this is especially the case for the process input/output (IO) file for system processes. To prevent this giving an error every time a collection occurs, it stores whether or not it has permission to access the file, and opens the file accordingly. This prevents the error log from overflowing with the same messages, making it difficult to see other, more serious errors. This permission problem mostly occurs with system processes,

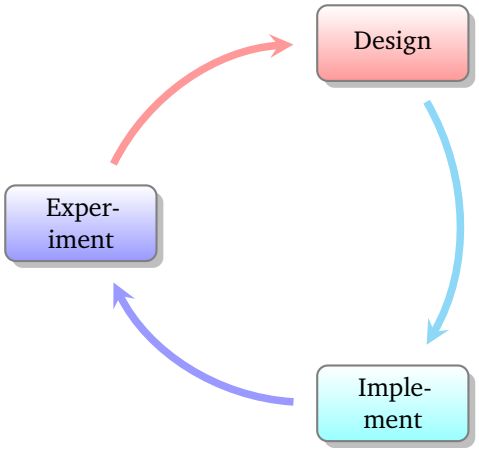


Figure 4.1: Implementation Routine

as the required permissions for reading the */proc/PID/io* file, is higher for these processes.

To decrease amount of data sent each time, the complete data for each process is only sent once, for all following collected metrics, the fields known to be unchanged, such as *command line arguments* are not sent to the server.

4.2 Analysis-Engine

4.2.1 Data Transfer

For the data transfer, the first attempt was made using binary, and not JSON to encode the data. The binary encoded messages was a lot smaller in size than the JSON equivalent, the size advantage was however lost when a bug caused random data to become corrupt, and being binary it was very difficult to debug. Altering this to using JSON made debugging easier, and the bug was quickly corrected. Because of being easier to debug if something were to happen, this encoding method was used onwards in the project.

For the JSON parsing server side, a third party library called JSMN[18] was used. JSMN proved to be very lightweight, both in the size of the code base, and in its very low resource consumption.

4.2.2 Storage

As databases are generally optimized for reads, the performance for consecutive writes may suffer, especially with large amounts of data, therefore it was decided to keep all the data needed for the analysis in memory.

5

Experiments

5.1 Experimental Setup

5.1.1 Platform

Experiments were run on the Tromsø Display Wall Cluster. This cluster consists of 29 nodes connected via a full-duplex gigabit Ethernet. The hardware of each node consists of:

- Intel Xeon W3550 quad core CPU at 3.07 GHz with hyper-threading (total of 8 threads).
- 12 GB of random-access memory (RAM).
- MSI GeForce GTX 560Ti graphics card with 1 GB GDDR5 video memory and PhysX PCI-Express 2.0.

The cluster is organized in such a way that one of the nodes functions as a remote entry point for the cluster. This node, referred to as the root node, has no display connected to it, while the remaining 28 nodes, referred to as tiles or display nodes, each has a projector connected. The projectors each have a resolution of 1024x768 pixels.

One other test machine was used, located on the same local area network (LAN) over a 5GHz wireless connection, the hardware consists of:

- Intel Core i7-4720HQ quad core CPU at 2.6 GHz with hyper-threading (a total of 8 threads).

- 16 GB or RAM.

Running a 64-bit version of Linux Mint 17.2 Cinnamon with Linux kernel 3.16.

Experiments were run on the Tromsø Display Wall Cluster unless anything else is specified.

5.2 Collector

The collectors CPU usage is determined from the raw data they collected from several executions.

The CPU usage percentage was found by dividing the number of CPU ticks used by each collector, by the total number of CPU ticks from their respective hosts during the same time-frame, and multiplying this value by one hundred.

The memory usage is from the *vsize* field from the */proc/PID/stat* file, this field contains the total memory usage of the process in bytes.

5.2.1 Network

The network usage is found by collecting the number of bytes sent from the collector each time it transfers data. This data is then used to find the average message size that the collector sends to the server each time.

5.3 Analysis-Engine

5.3.1 Benchmarking

The clock function from the C library *time.h* was used to benchmark how the resources was spent internally in the analysis-engine.

By subtracting the start time from the end time for each function call or code segment, the amount of time used (in CPU ticks), is found for each of these functions or segment.

Using this data, it is discovered which code segment uses the highest percentage of time, compared to the total time for each loop.

5.3.2 Usage

The usages for the server is calculated in the same fashion as for the collectors, using raw data gathered by a collector on the same cluster node.

Unlike with the collectors, the usages of the analysis-engine is expected to rise with increasing amounts of collectors connected, and as the runtime duration increases. Thus the measurements are done for several executions, with differing amounts of connected collectors over a limited time (5 minutes). And for with the whole cluster with a longer duration (330 minutes).

5.3.3 Storage

To find the storage space needed, the time and size of each of the stored files are collected for the long duration execution. Then the total storage usage over time is calculated by adding the sizes of the files for each of the storage times.

5.4 Analysis

To find out which data fields were useful, every gathered field was checked to see if they changed, and the frequency of their change, on a longer cluster execution.

5.5 Detection

To test the detection, several collectors and an analysis-engine was run on the single test machine. Using a testing option on one of the collectors, increasing the cpu load to 60 percent for that process, in addition to injecting false host data to the server (99 percent CPU usage, and 70 percent memory usage).

6 Results

6.1 Collector

The results from every execution showed that the collector usage remained virtually unchanged over time, therefore the average was calculated for the collector metrics.

The CPU usage was on average 0.785 percent, this was very stable across the collectors, none showing above 1 percent CPU usage. Average memory usage during these experiments was 2.8 MB, seemingly remaining stable at that usage.

There were only negligible variations in the measurements from the collectors, most differences being a few bytes of memory, or a few cpu ticks between.

Given the design of the collector the stable, unchanging usage over time is expected, as the workload, and needed memory allocation, remains the same, only changing if the number of processes on the host change.

6.1.1 Network

The average message size from the collector to the server is 126.2 KB, this remains stable as expected, because as with the memory usage, the message size only really changes if the number of processes change.

This is quite a lot larger than the 8 KB messages used in some systems, but they collect a very limited set of metrics, in addition to having a slower heartbeat for the collection.

With AutoMon doing collections every second this would for the test cluster amount up to 454.32 MB per hour from each collector. thus the network traffic for the whole cluster would be 13.175 gigabyte (GB) per hour.

6.2 Analysis-Engine

6.2.1 Benchmarks

The code benchmarking for the analysis-engine showed that about 99 percent of the time was spent in two different segments, the usage between these two was split quite evenly.

The first of these was the JSMN[18] JSON parsing function, as it was reported from benchmarks[**?**] that it was supposed to have a high parsing speed, it was discovered that for large files this was only the case with **PARENT_LINKS** enabled. After enabling this feature the time used by this function dropped by roughly 93 percent.

The other time consuming segment, was the code that stores the parsed JSON data into the memory structure. A recursive algorithm that caused a lot of overhead was used for this task, changing it to a non-recursive algorithm instead reduced the time-frame by roughly 75 percent.

The combined effort of these two optimizations reduced the total server usage by about 82 percent.

6.2.2 Scalability

The resource usage of the analysis-engine is intrinsically connected with the systems scalability. Collectors only communicate with the server, and they themselves have a steady resource usage, resulting in the server being the only system part affected by the increase in monitoring-agents. Therefore the analysis-engines resource usage is considered as scalability results.

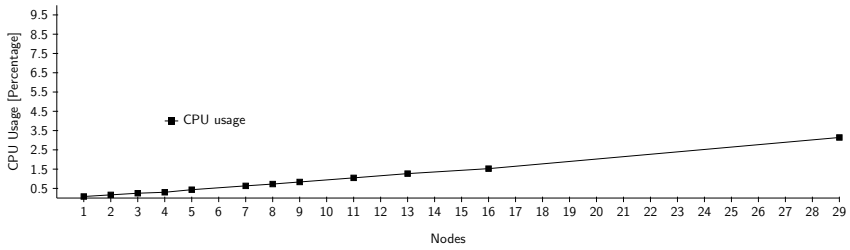


Figure 6.1: CPU usage with differing amount of nodes.

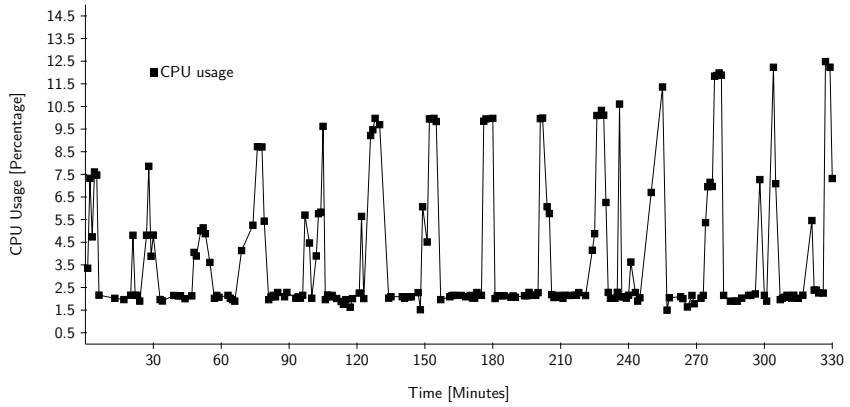


Figure 6.2: CPU usage, long term with 29 connected nodes.

CPU

The CPU usage for a limited execution time (2 minutes), using different amount of nodes on each execution, see figure 6.1, shows that the increase in CPU usage is quite linear based on the number of connected nodes.

When looking at the usage over time, for 29 connected monitoring-agents, the usage is fairly low most of the time, with some peaks appearing, see figure 6.2, with the average for this run being 3.728 percent CPU usage.

Memory

The memory usage for the 2 minute execution time, for differing amounts of nodes, the usage shows a near linear increase in usage based on the amount of connected nodes, see figure 6.3.

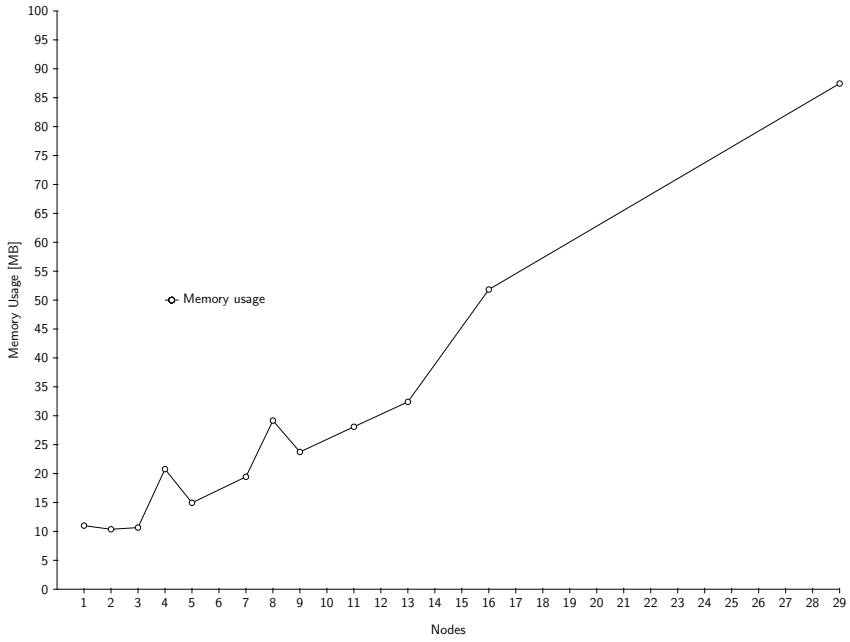


Figure 6.3: Memory usage with differing amount of nodes.

Storage

The storage usage over the long term execution for the whole cluster, this includes both memory and disk storage space used, see figure 6.4.

This shows that there is a linear increase in memory usage over time, with the disk usage increasing exponentially over time. This is because the complete memory structure is saved to disk each time, thereby increasing the used disk space with the size of the memory usage at each save time.

6.3 Analysis

The data analysis showed that the data fields for each process that changed most frequently was connected to:

- Cpu related fields, ticks used, processor used last, etc.
- Memory related fields, total memory size, stack pointer, etc.
- IO fields, especially bytes read, and bytes written fields.

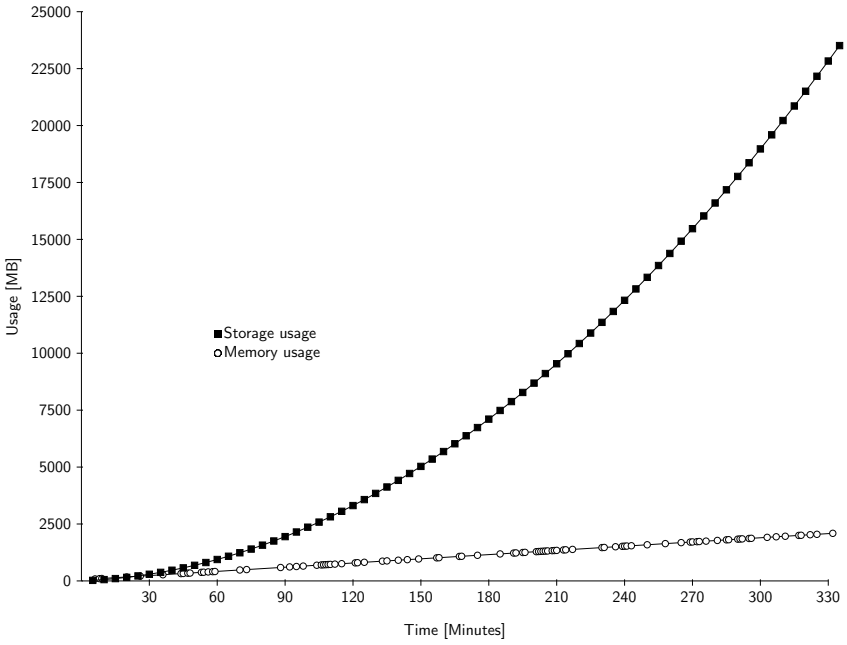


Figure 6.4: Storage Usage, long term with 29 connected nodes.

- Fault fields.

Fields connected with command line argument, start values, etc. never changed. This is expected as they only say something about starting states for the process, and these can't change at later times.

6.4 Detection

By running the detection test, the injected CPU and memory values are detected as abnormal from the cluster norm as expected, as these values are much higher than the cluster averages.

The increase in collector CPU usage was however not detected as it should be. This seems to be caused by a system bug, causing the CPU usage to be incorrectly calculated.

7 Discussion

7.1 Collector

The results from the experiments shows that creating collectors for gathering large amounts of system data from each node can be done without incurring any significant resource usage.

There is however, as can be seen from the network usage, reasons to consider doing more processing on the monitoring-agents,

One example would be to only send periodic reports to the server when the node is considered stable, then increase the frequency of these reports when the node is unstable in some way. This kind of frequency changing could however have the disadvantage of stressing the system at times when it is already under some stress (the reason the frequency changed), thus enhancing the existing problem.

There are good reasons to increase the workload on the monitoring-agents, one being a decrease in network traffic, this is especially important if the system is expected to work on some WAN, but as some tasks use a lot of network traffic in a cluster, having the monitoring system use a high degree of network traffic can cause problems it was meant to detect.

Another reason is that sending pre-processed data can lessen the cpu and memory usage of the server, thus it can handle more clients with the same

resources, giving the system a better scalability.

7.2 Design Choices

Using push-based communication between the collectors and server also comes with a disadvantage, and that is that the server becomes the system bottleneck. Seeing as the increases in resource usage is fairly linear, the amount of nodes a server can handle could, at least to some extent, be estimated. And if it is likely that the amount of nodes will exceed the capacity of a server, the system could be expanded with the possibility of adding additional servers to the system. Doing such an expansion would also give the possibility of redundancy by sending data to multiple server each time.

The decision to use HTTP was because of the ease of use, combined with the ability to add and remove nodes easily. message passing interface (MPI) for example have the problem that it may crash the entire system if one collector were to fail, and adding and removing nodes is difficult during runtime.

The lack of cleanup in the system memory structure was because of the intent to add a database system for storage. This way the most resent metrics could stay in memory for the calculations, and then written to disk when it is less likely to be of value to the necessary calculations. Thus maintaining a lower memory footprint, and having the option to write to the database in bulk instead of many smaller writes. This addition was not possible to achieve because of time constraints. The usage of the current memory and file setup was deemed good enough for this prototype, as the intent was to see if such a large amount could be gathered, and analyzed. This storage choice did however make early data analysis more difficult to achieve.

7.3 Development

There was also an idea to collect system power usage as part of the metrics using running average power limit (RAPL), this proved impossible for the prototype, as it is a hardware option, that only became available on processor models newer than those in the test cluster. With this option being mainly intended for servers, it is not available on the separate test machine either, as this is a consumer grade CPU.

The decision was first made to use binary encoding for the messages, this uses less space and therefore less network bandwidth. The problem with this

approach came when because of a system bug, the data became corrupted. This was difficult to detect and debug, and when not finding the problem the encoding was switched to JSON as this is human readable, with the hope that it would help in the debugging process.

Using JSON did help get enough debug data to be able to find and fix the specific bug that caused the error, being a helpful debug tool for the implementation of the prototype it was used for the rest of the project.

AutoMon first collected and kept all data from the monitoring-agents, this used an enormous amount of storage space, a different design with only storing changed data was implemented and tested, and this approach reduced stored data to approximately one hundredth of the previous usage.

7.4 Metric Analysis

The analysis of which metrics are needed to reliably determine if a process or system is executing normally is complex, there are a lot of variables that come into play, such as the task the processes are expected to perform.

It becomes clear that the metrics with high frequency updates, are important ways to monitor the process, but these are not necessarily the only important metrics. For metric analysis, the usage of statistics, or machine learning to define the metrics that can be associated with abnormal behavior seems like a very good place to start, this would over time also improve the detection rate of the system.

To improve the reliability of such a system, the addition of agents looking more closely at some processes (f.ex. those known to have a high resource usage), could improve the reliability of the detection, as it is more likely that a resource heavy process is the cause, than some system daemon checking on the system.

7.5 Observations

The observed CPU peaks for the experiment with the long execution time, could be connected with the storing of the memory structure to disk, as each of these peaks seems to be happening at times where such storage is scheduled.

Memory usage is as seen something that will always increase with the current

system design, and storing to disk being as it is, the used disk space will always be exponential, this is of course something that would be different if a database was used in conjunction with some form of memory cleanup instead.

The observed failure of the system to detect high CPU usage in a process seems to be because of a bug in the calculation of the CPU usage, caused by the system CPU ticks being reported as negative, this was unfortunately not fixed because of time constraints.

7.6 System

AutoMon design choice of having a single server, and thereby a single point of failure does mean a limit to its inherent scalability. There are however possibilities of adding more servers to the system architecture, and by doing so creating a more robust system.

Having a HTTP API makes it very easy to connect new clients to the system, and by having a server the collectors connect directly to, the amount of data transmitted between nodes, remains low. If using a hierarchy model with this large amount of collected metrics, it would quite possibly generate huge amounts of network traffic in the cluster network.

The current AutoMon storage system is as already mentioned, this was a solution that was implemented based on time constraints for the prototype. The way it stores data now is ineffective, as timestamps are stored with each metric value, wasting storage space, and is string based, costing resources to have it parsed if one wanted to find data for a specific time. Some more efficient manner to store this data, both space and resource wise, would be very helpful for improving the metric analysis.

Having the client collect all data and then sending it in bulk to the server gives the system much data to work through, and then nothing, this evens out a little bit with multiple clients, but it is still a problem. It would quite possibly be better if each client, collected and sent the data one process at a time as a stream, to even the load of both the client and the server.

7.7 Lessons Learned

One of the things I have learned is the complexity of working on distributed systems, even if an idea seems simple to design and implement there are a

lot of potential pitfalls that should be considered, one example is the system communication, this is somewhere a lot of potential problems can occur, and is somewhere extra care should be given during the design.

The fact that all objects that one would write to or read from in unix-based systems are considered to be a *file descriptor* is very handy, they are however very important to keep track of, as a problem was encountered when the system crashed because of trying to open a file, because the maximum number of file descriptors was reached from not closing the network sockets correctly.

8

Conclusion

A cluster monitoring system that frequently collects large amounts of data, and does rudimentary analysis on this data, that in addition has a low system resource requirement was developed.

Monitoring agents with a low resource usage, that collect data and sends it to a server is especially easy to achieve. The more difficult part of the system design is how to achieve storage and analysis of the data while keeping the usage at a reasonable level. Especially the storage space needed for the collected data for a cluster quickly increase with a high amount of collected metrics, thus there is a need to compress this data without losing anything vital, and storing it both for runtime analysis, but also for the possibility of later analysis.

The systems ability to detect faults is not completed and is currently only rudimentary, the analysis of which metrics is needed to detect faults are complex, and has a high degree of uncertainty. This is due to the very large amount of data collected, and the high degree of variation in metrics between processes. It thus becomes a big data problem, and it is necessary to automate this to some degree, for long term usage metrics to be able to define the required data to detect faults in the system.

For developing distributed systems it is invaluable to see what design choices others have made, and why they made their decision. Using this knowledge on how a system could be, is a good starting point for designing a new system based on the needs. As some design decisions are difficult to change after

starting implementing, it is a good idea to define the possible problems with the current design before starting implementation.

For systems that collect or process large amounts of data, it is vital to begin storing this data, and do analysis on it as early as possible, therefore it should immediately by stored in a fashion were it can easily be worked upon.

9

Future Work

As of now the analysis is not able to detect system faults, an improvement in this regard is needed. The completion of this task relies on a more defined normality of system processes, and to achieve this the storage should be finished so that long term data can be collected and analyzed.

There is also the need for a way to efficiently giving a notice to the user when something is detected, a web service for the system could be one way to do it, as this would be available as long as the user has access to the network.

Other things of interest would be to see if other methods of collecting metrics is better suited for gathering these large amounts of system data, and if there is any additional metrics that can be gathered.

In addition to doing more experimentation on increasing the amount of computations the collectors are doing, and decreasing the amount of data sent to the server.

Bibliography

[1] 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.

[2] "Hp insight cluster management utility." `http://www8.hp.com/us/en/products/server-software/product-detail.html?oid=3296361`. Accessed: 2016-05-05.

[3] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic monitoring of high-performance distributed applications," in *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pp. 163–170, 2002.

[4] R. Mooney, K. P. Schmidt, and R. S. Studham, "Nwperf: a system wide performance monitoring tool for large linux clusters," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 379–389, Sept 2004.

[5] M. Kluge, D. Hackenberg, and W. E. Nagel, "Collecting distributed performance data with dataheap: Generating and exploiting a holistic system view," *Procedia Computer Science*, vol. 9, pp. 1969 – 1978, 2012. Proceedings of the International Conference on Computational Science, {ICCS} 2012.

[6] "Ansible." `https://www.ansible.com/`. Accessed: 2016-05-05.

[7] "Munin." `http://munin-monitoring.org/`. Accessed: 2016-05-06.

[8] "M/monit." `https://mmonit.com/`. Accessed: 2016-05-06.

[9] "Collectd." `https://collectd.org/`. Accessed: 2016-05-06.

[10] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler, "Wide area cluster monitoring with ganglia," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 289 – 298, Dec 2003.

[11] L. Zhan, T. Z. Fu, D. M. Chiu, and Z. Lei, "A framework for monitoring and measuring a large-scale distributed system in real time," in *Proceedings of the 5th ACM Workshop on HotPlanet*, HotPlanet '13, (New York, NY, USA), pp. 21–26, ACM, 2013.

[12] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, pp. 164–206, May 2003.

[13] D. Clough, S. Rivera, M. Kuttel, V. Geddes, and P. Marais, "Panopticon: A scalable monitoring system," in *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, (New York, NY, USA), pp. 39–47, ACM, 2010.

[14] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Trans. Comput. Syst.*, vol. 5, pp. 121–150, Mar. 1987.

[15] K. Stefanov, V. Voevodin, S. Zhumatiy, and V. Voevodin, "Dynamically reconfigurable distributed modular monitoring system for supercomputers (dimmon)," *Procedia Computer Science*, vol. 66, pp. 625 – 634, 2015. 4th International Young Scientist Conference on Computational Science.

[16] H. Chen, G. Jiang, C. Ungureanu, and K. Yoshihira, "Combining supervised and unsupervised monitoring for fault detection in distributed computing systems," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, (New York, NY, USA), pp. 705–709, ACM, 2006.

[17] L. Tang, T. Li, L. Shwartz, F. Pinel, and G. Y. Grabarnik, "An integrated framework for optimizing automatic monitoring systems in large it infrastructures," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, (New York, NY, USA), pp. 1249–1257, ACM, 2013.

[18] "Jsmn." `http://zserge.com/jsmn.html`. Accessed: 2016-05-07.

A

System Usage

To start the system the files must first be extracted before running make.

This will create the system binaries in the bin folder.

Run automon_ad [-dpD] to start the server
-d to set debug
-p to set server portnumber
-D to change storage directory

Then run automon_cd [-adlpD] to start a client
-a to set server address
-d to set debug
-p to set server port number
-D to set storage directory

running 'run_automon tests' starts a series of scalability tests on the Tromsø Display Wall cluster.

B

Glossary

middleware is computer software that provides services to software applications beyond those available from the operating system.

multicast is group communication where information is addressed to a group of destination computers simultaneously (one-to-many or many-to-many distribution).

node is a single computer in a distributed systems cluster.

proc filesystem is a special filesystem in Unix-like operating systems that presents information about processes, and other system information in a hierarchical file-like structure, providing a more convenient and standardized method for dynamically accessing process data held in the kernel than traditional tracing methods or direct access to kernel memory.

runtime is the time during which a program is running (executing).

ticks measurement for the amount of time for which a CPU was used for processing instructions.

zombie is a process that has completed execution but still has an entry in the process table, this occurs for child processes when they wait for the parent to read their exit status.