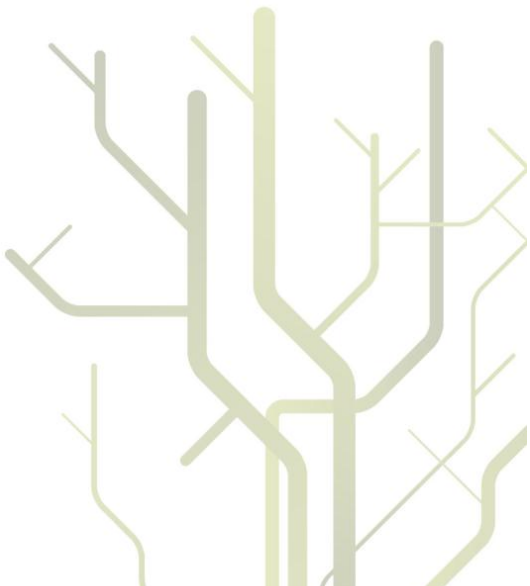
UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

WallMon: Interactive distributed monitoring of process-level resource usage on display and compute clusters

Arild Nilsen

INF-3990
Master's Thesis in Computer Science
November, 2011



Abstract

To achieve low overhead, traditional cluster monitoring systems sample data at low frequencies and with coarse granularity. However, interactive monitoring requires frequent (up to 60 Hz) sampling of fine-grained data and visualization tools that can explore and display data in near real-time. This makes traditional cluster monitoring systems unsuited for interactive monitoring of distributed cluster applications, as they fail to capture short-duration events, making understanding the performance relationship between processes on the same or different nodes difficult. To address this issue, WallMon was developed, a tool for interactive visual exploration of performance behaviors in distributed systems. For gathering of data, WallMon is centered around an abstraction of collectors and handlers; collectors gathers data of interest, such as CPU and memory usage, and forwards it to handlers in a push-based fashion, while handlers take action upon the data. WallMon captures and visualizes data for every process on every node, as well as overall node statistics. Data is visualized using a technique inspired by the concept of information flocking. WallMon's design is based on the client-server model, and it is extensible through a module system that encapsulates functionality specific to monitoring (collectors) and visualization (handlers). A set of experiments have been carried out on a cluster of 29 nodes with 180 processes per node. Performance results show 7% (of 100) CPU usage at 64 Hz sampling rate when performing process-level monitoring with WallMon. Using WallMon's interactive visualization, we have observed interesting patterns in different parallel and distributed systems, such as unexpected ratio of user- and kernel-level execution among processes in a particular distributed system.

Acknowledgments

I would like to thank my advisor Professor Otto J. Anshus for his guidance and support during this thesis. Otto has motivated and inspired me to keep on researching, developing and refining the systems presented in this thesis.

I would like to thank my co-advisors Daniel Stødle and Tor-Magne Stien Hagen for their support and valuable feedback during the thesis. I would also like to thank Bård Fjukstad and Lars Ailo Bongo for discussions and support.

I thank the technical administrative staff at the Computer Science department at the University of Tromsø. Your support made my life easier during both the thesis and the rest of my study.

I thank the Norwegian Research council for their funding for the projects No. 187828, No. 159936/V30 and No. 155550/420.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Clusters of computers are one of the most used alternatives for parallel systems construction primarily due to its high scalability and excellent cost-performance ratio. However, understanding overall performance behavior of a cluster and the individual computer systems it executes, is a difficult and complex task. A common approach to address this task is to have a monitoring system for the cluster. Such a system is responsible for collecting data, such as total CPU and memory usage, from the individual computers in the cluster, and storing and/or presenting it to the user. The presentation of data is often based visualization, such as charts and lists. A particular type of computer systems where cluster monitoring can be useful, is distributed systems. These systems are executed on multiple computers, making it hard to correlate and discover performance patterns.

Traditional cluster monitoring systems, such as Ganglia [1] and HP Cluster Management Utility [2], gather data about resource usage in cluster of computers and presents it visually to users. To achieve low overhead, these systems sample data at low frequencies and with coarse granularity. For example, [1] reports a default sampling frequency of 15 seconds and a data granularity on node level, such as total CPU and memory usage. However, interactive monitoring requires frequent (up to 60 Hz) sampling of fine-grained data and visualization tools that can explore and display data in near real-time. This makes traditional cluster monitoring systems unsuited for interactive monitoring of distributed cluster applications, as they fail to capture short-duration events, making understanding the performance relationship between processes on the same or different nodes difficult. Suitable sampling frequency for interactive monitoring varies between application domains and applications. In situations where it is important to capture and visualize short-duration events, the sampling rate should match the frame rate of the visualization, which can be as high as 60 frames per second. However, sampling at 60 Hz might capture too much data and make the visualization difficult to follow and understand. Fine-grained data is important when related distributed systems execute simultaneously. For example, without process-level granularity, it would be difficult to observe how behavior in one system impacts behavior of other systems. Interactivity is useful to navigate and explore the large amount of data produced by fine-grained sampling.

WallMon is a tool for interactive visual exploration of performance behaviors in distributed systems. The WallMon system captures and visualizes data for every process on every node,

as well as overall node statistics. For gathering of data, WallMon is centered around an abstraction of collectors and handlers; collectors gathers data of interest, such as CPU and memory usage, and forwards it to handlers in a push-based fashion, while handlers take action upon the data. At the architectural level, collectors and handlers are glued together and managed by a light-weight runtime. WallMon's design is based on the client-server model, and it is extensible through a module system that encapsulates functionality specific to monitoring (collectors) and visualization (handlers). Although WallMon provides process-level granularity, it does not, nor does it intend to, provide capabilities of profilers. While profilers are able to gather micro-level data, such as detailed stack traces of processes, WallMon's process-level data is at a macro-level, such as total CPU, memory and network I/O consumption of single processes.

The visualization of data gathered by WallMon is based on an approach inspired by the concept of information flocking [3], originally introduced by Proctor & Winter: "*[...] Information flocking presents data in a form which is particularly suited to the human brain's evolved ability to process information. Human beings are very good at seeing patterns in colour and motion, and Information Flocking is a way of leveraging this ability*". As figure 1.1 shows, flocking behavior is applied to processes. The idea behind this concept in WallMon, is that distributed systems, which consists of related processes, show, or are expected to show flocking behavior. Such behavior might also be interesting to observe and explore when multiple distributed systems execute simultaneously.

Figure 1.1: WallMon running on the Tromsø display wall. WallMon's visualization runs on 3x2 tiles, while an image of Tromsø is rendered in the background. Moving symbols within axes represent different performance metrics of processes running on the display wall cluster.

A set of experiments have been carried out on a cluster of 29 nodes with 180 processes per node. Performance results show 7% (of 100) CPU usage at 64 Hz sampling rate when performing process-level monitoring with WallMon. Its maximum sampling rate on this cluster is about 128 Hz. On this cluster, the primary overhead is caused by obtaining raw process-level data from the operating system.

The visualization inspired by information flocking has been applied to a several distributed systems, and certain interesting patterns have been observed. For example, when exploring data sets in a particular distributed system, certain visually similar data sets caused some of the participating processes to execute more on kernel-level (and less on user-level). This particular example can be seen in figure 1.1; the majority of square shaped symbols, which represents CPU utilization horizontally and ratio between user- and kernel-level execution

vertically, have about the same CPU utilization, however, they are unexpectedly scattered vertically. Compared to traditional visualization approaches, such as using charts and/or graphs, our experience is that WallMon's visualization makes it easier to discover patterns in distributed systems. We are currently experimenting with the visualization technique on a diverse range of distributed systems, and extending and improving WallMon.

1.1 Problem Statement

Performing interactive visualization of process-level data on high-resolution tiled display walls is a challenge. This challenge can be divided into two categories: cluster monitoring, and data visualization and interactivity. Challenges for cluster monitoring in this thesis are:

1. **Process-level monitoring**. The thesis' goal is to monitor processes externally with low overhead to catch data about the behavior processes, including user- and kernel-level CPU usage, code, memory consumption, file I/O and network I/O. The resolution time-wise should be fine (at least millisecond accuracy), and happen frequent enough to, with some probability, cover significant events.

2. **Data aggregation**. Cluster of computers often run a combined number of processes in the thousands, or hundred-thousands. This presents a challenge when carrying out process-level monitoring, which can produce large amount of data that must be aggregated in order to take action upon it. An architecture for cluster monitoring must be able to efficiently aggregate data in order to adhere to WallMon's goal of providing near real-time data visualization. Moreover, display walls often have a distributed and parallel architecture [4, 5]. An architecture must take such an organization into account.

When the infrastructure for aggregating process-level data is in place, the goal of the thesis is to explore ways of visualizing and interacting with this data. Challenges for data visualization and interactivity in this thesis are:

1. **Visualization techniques**. Traditional approaches for visualization in cluster monitoring include charts and lists. However, in this thesis there are several factors that make such approaches less useful. Firstly, many of the systems that use charts and lists sample data at coarse granularity, such as overall node statistics, while WallMon samples at process-level granularity. Charts and lists might not be that useful to visualize data at fine granularity. Secondly, the presence of a display wall is likely to influence the visualization. For example, its large display surface allows for more details, and its integrated event system, which will form the basis for data exploration within the visualization, might influence the visualization technique(s).

2. **Real-time visualization**. The presence of a display wall is an important motivating factor for real-time visualization in WallMon. A useful scenario for real-time visualization is where WallMon's visualization executes on some part of the display wall, while at the same time another system executes on another part of the display wall. In such

an scenario, also shown in figure 4.2, WallMon should be able to monitor this system and reflect performance changes in real-time through its visualization. This way, if the monitored system has visual output and/or allows for interactive input, these mechanisms will immediately be reflected in WallMon's visualization, and might reveal interesting behavior and patterns that otherwise would be difficult to observe. To achieve real-time visualization, both the latency for aggregating data, and processing and visualization of the data must be sufficiently low.

3. **Distributed visualization and interactivity**. Distributed visualization and interactivity might require synchronization of state in order to appear as a coherent visualization. For visualization, display-output might need to be synchronized, while for interactivity, event processing might need synchronization. The degree of synchronization will depend on the visualization techniques explored in the thesis.

1.2 Scientific Contributions

This section presents the scientific contributions claimed by this thesis. The contributions have been established using a systems approach where architectures, designs and implementations are developed, before experiments are carried out to document the resulting systems' characteristics.

1.2.1 Models and Architectures

This section presents the models and architectures developed in this thesis.

Collector-handler model

The collector-handler model provides an abstraction for gathering and taking action upon gathered data; collectors gather data of interest and handlers take action upon gathered data, while external mechanisms glue together related collectors and handlers. The model is not limited to cluster monitoring, however, it was designed for it. A characteristic of the model is that it adheres to the end-to-end argument [6]; only the collectors and handlers have knowledge of the semantic meaning of gathered data, while the external mechanisms gluing together collectors and handlers are only concerned with efficient data aggregation.

Information flocking model for cluster monitoring

The information flocking model is a visualization technique inspired by [3]. The model is not limited to cluster monitoring, however, it was designed for it. Two components make up the model: a chart with moving entities representing different performance metrics and lists which allows for hierarchical exploration of data. A particular useful experience with this model for visualization, is that it often provides better means to spot and discover patterns compared to traditional models, such as traditional graphs/charts and lists.

Cluster monitoring architecture

The architecture for cluster monitoring developed in this thesis provides real-time aggregation of data, and extensibility through a module system. Real-time data aggregation is based on push-based data transmission, and extensibility allows for integration of the collector-handler model.

1.2.2 The WallMon system

The WallMon system for cluster monitoring realizes and implements the collector-handler model, information flocking model and the cluster monitoring architecture presented in section 1.2.1. In addition to these models and architecture, the system also contributes by exploring and implementing interactivity mechanisms for visual data exploration.

1.2.3 Impact

The impact of this thesis includes a paper published in NIK (Norsk Informatikk Konferanse) 2011. Moreover, the WallMon system has been used to analyze several systems: Gigapix [7] and an implementation for the Weather Research & Forecasting (WRF) Model [8], as well as student assignments in the distributed systems course at the Computer Science department at University of Tromsø.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the methodology for this thesis. Chapter 3 presents and discusses background and related work. Chapter 4 details the work on building WallMon's underlying infrastructure for cluster monitoring, before chapter 5 presents and evaluates the visualization techniques explored and employed in WallMon. Chapter 6 presents the interactivity mechanisms present in WallMon, and the challenges related to it. Chapter 7 discusses the thesis, and Chapter 8 draws conclusions based on the research presented. Chapter 9 outlines future work for WallMon.

Appendix A contains a complete copy of the single paper published during the work on WallMon. Appendix B gives an overview of the contents of the CD-ROM accompanying this thesis.

Chapter 2

Methodology

The research carried out in this thesis follows a systems approach. This approach (figure 2.1) starts with an idea, which describes the research intended at an overall level and its goals. To evaluate an idea, a system is created to gain knowledge of its implications on real hardware. The first step towards realizing the system is an architecture describing the interactions between the major components of the system. With the architecture as a basis, a specific design is devised for the system. The design gives a description of the components in the architecture. The implementation realizes the system on actual hardware.

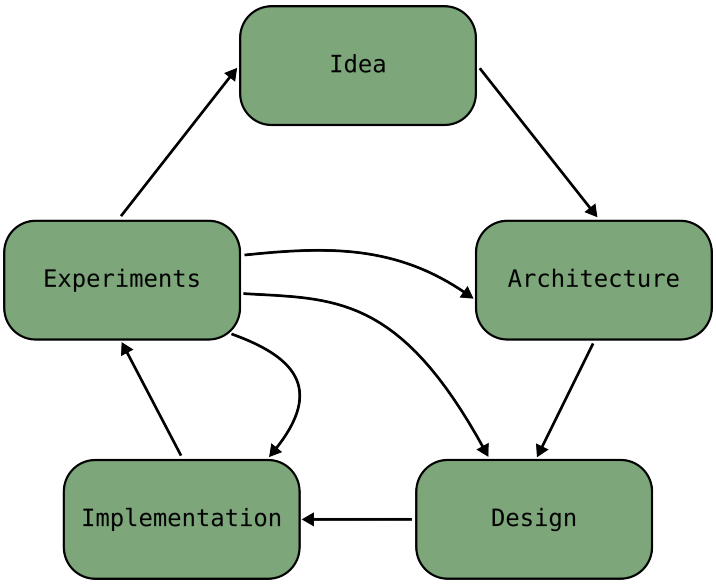


Figure 2.1: Systems research methodology.

The last step in the cycle (figure 2.1) is experiments. In this step, performance measurements are carried out in a controlled manner in order to evaluate and and demonstrate the implications of the initial idea. With the results from the experiments as a basis, analyses and conclusions are drawn. The conclusions might reveal that the idea, architecture, design or implementation need to be revisited and refined, which implies that the cycle continues.

The reason for carrying out an implementation on real hardware is due to the complexity of

the interplay between software and hardware. Without an implementation, it is difficult to predict the outcome of an idea without creating a system that realizes it.

2.1 Metrics

The metrics presented in this section are used in both the experiments carried out on the WallMon system, and in WallMon's visualization of performance behaviours. Unless otherwise stated, measurements of the metrics presented in this section have been obtained from the procfs file system. procfs is a virtual file system that is present in most Linux environments. Among others, procfs keeps a file for each process on the system. These files are continuously updated with different types of data and performance metrics, and they are located under `/proc/<pid>/`, where `<pid>` is the process id of a particular process.

2.1.1 CPU load

CPU load is calculated by the time-period a process has been executing on the CPU divided by the total time of the measurement, before multiplying by 100 which gives a percent representation. This metric is relative to the number of cores and hardware-threads present on the CPU. For example, given a single-core processor with HyperThreading enabled, a process that has been allocated 500 milliseconds execution time for a period of 1 second will have a CPU load of 100 percent. In this example, maximum CPU load is 200 percent.

CPU load can be split into user-level load and kernel-level load. User-level load represents the time spent in user space executing non-privileged instructions, while kernel-level load is the time spent executing inside of the kernel on behalf of the process. Kernel-level load is sometimes referred to as system-level load. The CPU load is obtained from the `/proc/<pid>/stat` file within procfs.

2.1.2 Memory usage

Memory usage represents the total amount of memory allocated by a process. Both stack and heap usage are included in this metric. The measurements for memory usage originate from the `/proc/<pid>/statm` file within procfs.

2.1.3 Network bandwidth usage

Network bandwidth usage is the number of bytes sent and received by a process within a given time period. In this thesis, it is reported in (mega/kilo) bytes per second. Since the procfs file system is used for this metric, the measurements represent all network bandwidth usage of the process in the sense of number of bytes passed to system calls such as `read()` and `write()`. Network bandwidth usage is obtained from the `/proc/<pid>/io` file within procfs.

2.1.4 Storage bandwidth usage

The storage bandwidth usage is similar to the network bandwidth usage, except that it measures the bandwidth from and towards persistent storage, such as a hard disk drive or solid-state drive. Measurements of the metric is also obtained from `/proc/<pid>/io`.

2.1.5 Utilization

Utilization is not a specific metric itself, but a term that describes the aforementioned metrics in an alternative way. It unifies the metrics by describing them along a single axis ranging from 0 percent utilization to 100 percent utilization. For example, a memory usage of 1 GB on a computer with a total of 2 GB memory, results in 50 percent memory utilization for a particular process. An example for CPU utilization is where a process has exclusive usage of a single core out of four available cores during the measurement, which would result in a CPU *utilization* of 25 percent, while the CPU *load* would be 100 percent. For network and storage bandwidth usage it is harder to define utilization due to the difficulty of measuring total bandwidth usage. For WallMon's visualization, the total bandwidth usage for both network and storage is set to **50 MB/s** in order to calculate utilization.

2.2 Platform for experiments

Unless otherwise stated, the experiments presented in this thesis have been carried out on the Tromsø display wall cluster. This cluster consistes of 29 nodes connected via a full-duplex gigabit Ethernet. The hardware of each node consists of a quad-core Intel Xeon W355 CPU at 3.07 GHz with hyper-threading (a total of 8 threads per node) , 12 GB of RAM and a MSI GeForce GTX 560Ti graphics card with 1 GB GDDR5 video memory and PhysX PCI-Express 2.0. Each node runs a 64-bit version of the Rocks Cluster Distribution with Linux kernel 2.6.18. 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.

Since WallMon sample data at process-level granularity, the number of processes a node execute is important performance-wise. On the display wall cluster, the root node executes about 280 processes, while the remaining display nodes execute 180 processes each.

2.3 Wallgrabber

Throughout this thesis there are several images showing WallMon's visualization on the Tromsø display wall. Some of these images are screen shots taken with Wallgrabber [9], which is a tool for taking screen shots of the Tromsø display wall. The screen shots taken show WallMon running on 6 of the total 28 tiles of the display wall. In these screen shots, Wallgrabber was configured to only include these 6 tiles (except from figure 2.2).

As shown in figure 2.2, taking screen shots with WallGrabber consumes resources on the nodes involved, and might interfere with already executing processes and alter their performance pattern in WallMon's visualization. However, the performance patterns in the screen shots presented should, to a large extent, be unaffected by Wallgrabber since at the time the screen shot is taken, that is, the framebuffer is read, the performance impact of Wallgrabber has not yet been registered by WallMon. The performance impact of WallGrabber in figure 2.2 is a result of several consecutive screen shots taken by Wallgrabber, and hence the performance impact of an earlier screen shot is present.

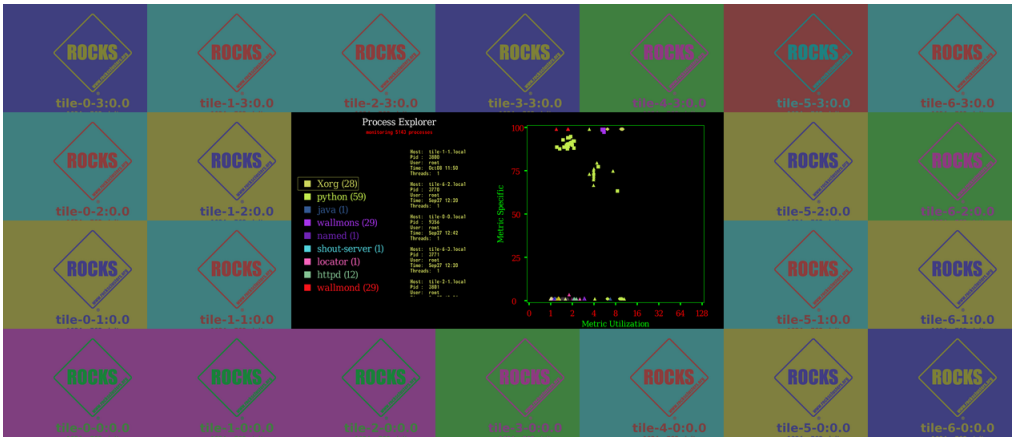


Figure 2.2: Screen shot of Wallgrabber covering the entire display wall. The yellow symbols in WallMon's visualization represent the temporary resource usage by Wallgrabber.

2.4 Sampling frequency

The data sampling frequency of WallMon can be configured. Unless otherwise stated, the sampling frequency used to conduct the experiments in this thesis was configured to 750 milliseconds, meaning that WallMon would only sample data every 750 millisecond. This is a useful interval for most systems monitored by WallMon, and it results in acceptable resource consumption by WallMon. Also, unless otherwise stated, the term *sampling* in this thesis implies collecting data about all present processes.

Chapter 3

Background and Related Work

This section presents three categories of background and related work. The first section gives a brief introduction to cluster monitoring, and presents existing cluster monitoring systems. The focus of the next section is visualization of monitored data. The section gives a brief introduction to visualization, and presents approaches to visualization in existing monitoring systems. The last section briefly presents and discusses display walls in general, and the Tromsø display wall which has been the primary platform for development and testing of WallMon.

3.1 Cluster monitoring

3.1.1 Introduction

The majority of cluster monitoring systems follow the general architecture showed in figure 3.1: collector, aggregator, storage, and visualizer. A *collector* is typically a background process running on each cluster node. It periodically reads performance metric values, such as CPU load and memory usage. These values are often obtained from the OS. For example, in an UNIX environment this often is achieved by reading the procfs file system. However, the same collector, or other collectors at the same node, might obtain them from middleware systems and/or applications. Such middleware systems include high performance systems, such as Hadoop Map Reduce. These systems provide external components with performance values through well-defined interfaces.

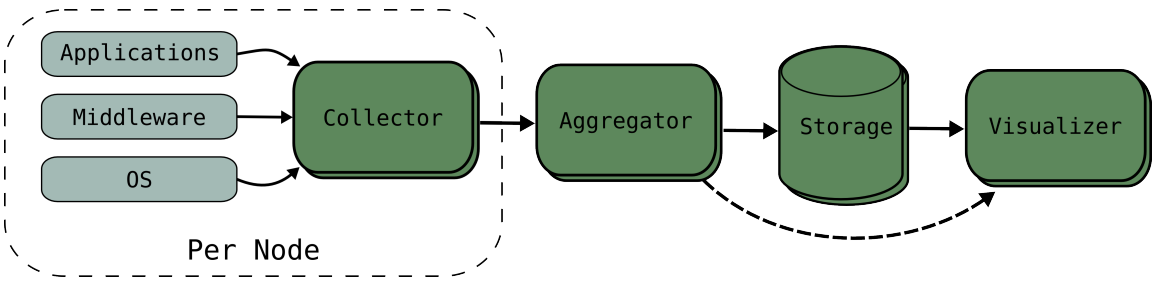


Figure 3.1: General architecture for cluster monitoring.

Data gathered by collectors are sent to one or more *aggregators*. In a client-server architecture there might only be a single aggregator, however, in a hierarchical architecture there can be several. In some systems, an aggregator will filter and/or transform the data before forwarding it to a *storage back-end*, which persistently stores the data and allows for post-analyses of data. A storage-backend usually provides a query interface for data access. On the other hand, some systems do not have any storage back-ends, and the aggregator immediately forwards data to a *visualizer*. A visualizer provides a graphical interface for users to analyze the metrics gathered by the system. Although visualization usually is the single final step in a cluster monitoring system, there are other approaches, such applying statistics to data [10].

3.1.2 Systems

Ganglia [1] and HP Cluster Management Utility [2] are two traditional cluster monitoring systems that are designed for overall cluster management. They obtain low overhead and scalability through data sampling at low frequencies and with coarse granularity. For example, [1] reports a default sampling frequency of 15 seconds and a data granularity on node level, such as total CPU and memory usage. Supermon [11] is a system similar to Ganglia. Both systems employ a pull-based approach for transfer of data, and a hierarchical architecture for scalable aggregation of data. Supermon focuses on and provides frequent sampling of data. It uses a kernel module for Linux in order to avoid context switch overhead when gathering data from the procfs file system.

The majority of cluster monitoring systems provide to some extent extensibility through a module system. RVision [12] is such an example, and the system provides a more elaborate module system, which is similar to WallMon's module system. Both RVisison and WallMon allow user-defined modules to be dynamically loaded at runtime. Differences between the module systems include that RVision's is procedure oriented, while WallMon's is object oriented, and that WallMon offers more control over the usage of gathered data.

dproc (distributed /proc) is a flexible system that by itself is not a complete monitoring system. However, dproc provides external applications with selective monitoring via publish/subcribe channels. A channel in dproc is an entry in the procfs file system on some specified cluster node. When an subscription has been made to a channel, dproc will publish performance data from procfs to the given application at a specified rate. While WallMon does not provide external applications with hooks that allows for selective monitoring, the module system in WallMon has similarities to dproc's model. For example, if an application could be contained in a WallMon module, WallMon would be able to provide capabilities similar to dproc. However, it is likely that for most applications, using dproc would be simpler and more convenient. Another approach for WallMon to provide capabilities of dproc, could be to implement a module that provided hooks for external applications.

Otus [13] is a recent cluster monitoring system that, as with WallMon, samples data at process-level. Otus also have support for sampling data from specific applications, such as Hadoop MapReduce. [13] reports that in order to achieve the initial requirements of Otus,

data should be collected at the interval 1-10 seconds. The experiments conducted in [13] collected data at a 5 seconds interval, while transmitting data at a 10 seconds interval. Although Otus has the potential to collect data at a relative rapid rate, its goal is to provide detailed post-analyses via charts, and not real-time analyses. On the other hand, WallMon's goal is real-time data visualization.

3.2 Data visualization

3.2.1 Introduction

Visualization can be defined according to McCormick et al. [14]: "*Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science.*". This definition highlights some factors to why visualization is common and useful within the field of cluster monitoring. Firstly, computer systems, especially distributed systems, are complex. In order to understand them, several performance metrics must constantly be monitored, which produces large amount of data. This data, in its symbolic representation (.e.g log files), quickly becomes overwhelming, and makes it difficult to recognize and observe patterns. By transforming symbolic values into geometric representations, such as charts and animations, the human's brain visual capabilities are better exploited [15, 3].

Another important factor that McCormick's definition highlights, is that when monitoring a cluster/distributed system, one is not always sure what to expect and look for. A visualization can then reveal interesting patterns that would not otherwise been discovered.

3.2.2 Charts and lists

The primary visualization techniques in many cluster monitoring systems are based on charts, such as graphs and bar charts. For example, figure 3.2 shows a graph in Ganglia that conveys a cluster's bandwidth usage over time, while figure 3.3 shows a chart used in Otus [13]. Although there are certain types of charts that more frequently used in cluster monitoring, there exists many alternatives type of charts [16]. Charts provide a familiar and intuitive interface for conveying information, however, they might not be a suitable approach for visualizing large amount of low-level data. For example, given process-level data, if each present process would be represented in its own chart, there might end up being too many charts, making it hard to get a coherent overview. On the other hand, having all processes represented in one chart could make the single chart too complex to efficiently convey information.

Figure 3.2: Web front-end in Ganglia showing various charts [1, p. 827]

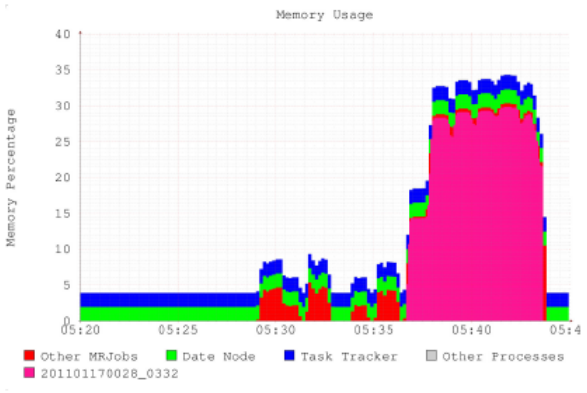


Figure 3.3: Chart in Otus showing whole cluster memory usage [13, p. 4]

Lists are another common visualization technique. Compared to charts, they are often applied when there are a lot of detailed and/or diverse data present. Process managers , such as the UNIX `top` application shown in figure 3.4, often visualize process-level data with lists. `top` simply lists the (locally) running processes, and as implied by its name, displays the "top" CPU intensive processes at the top of its list, which is continuously updated. The process list in `top` is limited to the number of lines on the display (typically around 30 to 40), however, there exists numerous alternative process managers similar to `top`, which might handle this limitation. Another example of lists is shown in figure 3.5. This list originates from HP Cluster Management Utility, and lists detailed and diverse information about a particular cluster node.

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
23520 arild     20   0  599m 453m  24m S   15 11.3  21:55.46 opera
 2271 arild     20   0  187m  48m  25m S    4  1.2  17:27.38 spotify
 1142 gkrellmd  20   0 13660 1340 1076 S    1  0.0  13:04.03 gkrellmd
 1153 root      20   0  178m  29m  11m S    1  0.7   6:08.93 Xorg
 1653 arild      9 -11  166m 8684 7112 S    1  0.2   0:51.89 pulseaudio
23542 arild     20   0 94216  19m  12m S    1  0.5   0:43.22 operapluginwrap
27865 arild     20   0  2620 1180  840 R    1  0.0   0:00.14 top
    7 root      20   0     0    0    0 S    0  0.0   0:00.78 ksoftirqd/1
   69 root      20   0     0    0    0 S    0  0.0   0:17.20 kondemand/0
 3147 arild     20   0 97008  15m  10m S    0  0.4   0:03.40 gnome-terminal
27370 arild     20   0  2620 1192  840 S    0  0.0   0:01.59 top
    1 root      20   0  2868 1696 1220 S    0  0.0   0:00.71 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.82 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
    9 root      RT   0     0    0    0 S    0  0.0   0:00.01 migration/2
   10 root      20   0     0    0    0 S    0  0.0   0:00.21 ksoftirqd/2
   11 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/2
   12 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/3
   13 root      20   0     0    0    0 S    0  0.0   0:00.13 ksoftirqd/3
   14 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/3
```

Figure 3.4: Example of the top process manager. Each entry in the list represtens a process.

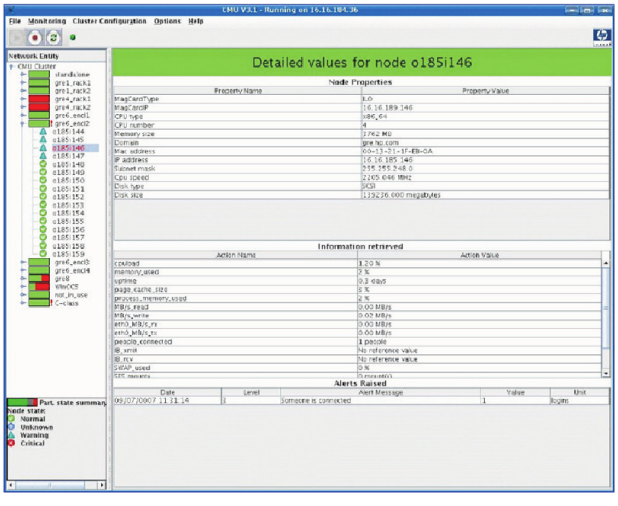


Figure 3.5: HP Cluster Management Utility displaying detailed per-node information [17, p. 13].

3.2.3 Abstract concepts

PSDoom [18] and [19] provide administration through game-like interfaces. In PSDoom, local processes are visualized by a modified version of Doom; a well-known computer game. This approach relates to WallMon's visualization in several ways. For example, as shown in figure 3.6, a user can quickly get an overview of the load on the system by inspecting how crowded by monsters a room is. Another similarity, also shown in figure 3.6, is that a user can kill (terminate) any process via a unified interface, and then observe the effects of the termination.

Figure 3.6: Process visualization in PSDoom [18, p. 2]. Each monster represents a process.

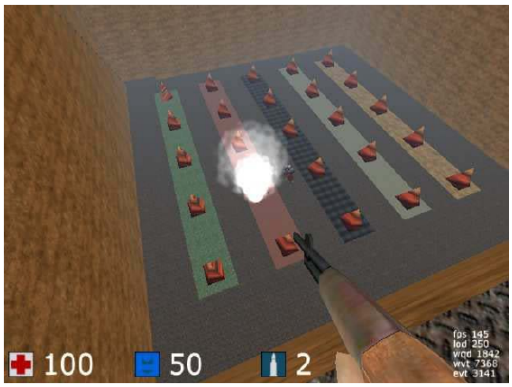


Figure 3.7: Computer game inspired network management interface in [19, p. 8]. The tiles on the ground each represent a host.

LavaPS is another local process visualizer with similarities to WallMon's visualization. The interface of LavaPS consists of a lamp showing blobs of different sizes and colors, where each blob represents a process. Blob size is proportional to memory usage, while movement is proportional to CPU usage. Color is a combination of program name and time since the program last ran. WallMon's visualization uses similar techniques: the color of an object is based on the name of a process, movement and position represent performance usage of different types of metrics, such as CPU and memory, and the shape of an object is used to distinguish between the different performance metrics. Both LavaPS and WallMon's visualization adheres to principles of information flocking [3].

The research in [20] is also based on information flocking. [20] applies their interpretation of information flocking to several areas, such as time-varying data simulation, live database querying, and continuous data streaming. Although the system has not been applied to computer monitoring, the system's capability for supporting continuous data streaming might allow for it. The system's interface, as shown in figure 3.9, might not be suitable for process monitoring; with the current interface, it might be difficult to distinguish between different processes on a cluster running thousands of processes. Moreover, [20] mentions that the visualization only scales to around 500 objects being animated in real-time.

Figure 3.8: The LavaPS interface to process visualization. Each blob represents a process.

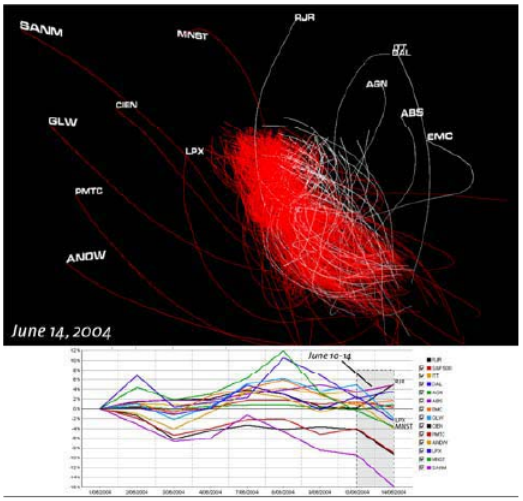


Figure 3.9: Information flocking inspired interface in [20, p. 101]. Majority of objects shows flocking behaviour, while some individual objects separate from the flock

3.3 Display walls

A display wall is a scalable high-resolution tiled display (figure 3.10). Tiled means that the display wall is made up of multiple displays arranged in a grid. Due to multiple displays, a display wall provides higher-resolution than normal desktop displays, which is typically between one to four megapixels. For example, the Tromsø display has a resolution of 22 megapixels, and the display wall system in [5] provides tens-to-hundreds of megapixels. Scalable implies that higher resolution can be achieved by adding more displays to the grid.

Figure 3.10: Illustration of the display wall at the Department of Computer Science, University of Tromsø.

There are several key characteristics of a display wall. Firstly, its physical size enables several people to use it simultaneously. Secondly, its high resolution allows large amounts of information to be presented. Lastly, the combination of size and resolution enable users to get overviews, while at the same time being able to walk up close to inspect details.

The Tromsø display wall consists of 28 tiles, which each tile consists of a projector and a computer driving the projector. Each tile has a resolution of 1024x768, giving a total resolution of 22 megapixels. The display wall allows for interactive input from users through an event system named *Shout*. As figure 3.10 shows, user input can be provided by hand-movement, which is registered by cameras. Moreover, Shout also supports user input through sound, which is registered by microphones. Any event detected by Shout is made available to applications either executing within the display wall cluster, or outside of the cluster. Shout gives applications an entrance for integrating and making use of user input on the display wall.

Chapter 4

Cluster Monitoring

This chapter presents the underlying infrastructure of WallMon. The infrastructure handles data aggregation, and forms the basis for the visualization techniques and interactivity mechanisms employed in WallMon.

4.1 Idea

4.1.1 Collector-handler model

An initial idea in WallMon was to provide an extensible architecture, a feature that is available in several cluster monitoring systems [12, 13]. The primary reason for this requirement was due to the diversity of cluster monitoring. For example, a cluster monitoring system can gather data at different granularity from a variety of sources, such as the OS, middleware systems and applications. Taking action upon gathered data is perhaps even more diverse, and includes visualization, storage and applying statistics. Another requirement for this idea was to provide extensibility at a high-degree, implying that a user could extend the system in almost any way. This idea resulted in the collector-handler model shown in figure 4.1.

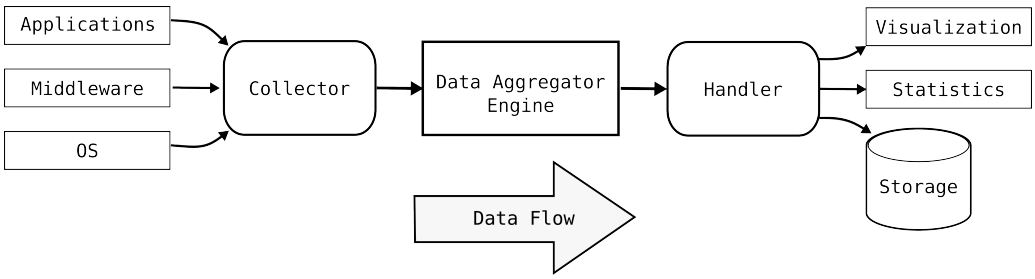


Figure 4.1: Collector-handler model in WallMon. The collector and handler are implemented by the user, while a data aggregator engine glues the collector and handler together. A collector gather data of interest, while a handler takes action upon this data.

The responsibility of a collector is to gather data of interest. Since a user of the system implements the collector, the data can be obtained from any source. On the other hand, the responsibility of a handler is to take action upon the data gathered by the collector.

The handler is also implemented by the user, hence the actions can by anything. A data aggregator engine is responsible for connecting the collector and handler.

4.1.2 Real-time data aggregation

The initial idea for WallMon's architecture was aggregation of data in real-time. The motivating factor for this idea was the presence of a display wall. Figure 4.2 shows a useful scenario for real-time data aggregation: WallMon's visualization executes on some part of the display wall, while at the same time another system executes on another part of the display wall. In such an scenario, WallMon should be able to monitor this system and reflect performance changes in real-time through its visualization. This way, if the monitored system has visual output and/or allows for interactive input, the performance consequences of these mechanisms will immediately be reflected in WallMon's visualization, and might reveal interesting behavior and patterns that otherwise would be difficult to observe.

Figure 4.2: Illustration of the usefulness of real-time data aggregation. WallMon's visualization runs on parts of the Tromsø display wall, while another interactive system runs on the whole display wall. WallMon is able to reflect performance changes due to interactive input, in real-time.

As figure 4.2 shows, *real-time is defined as relative to human perception*, where anywhere from 50 milliseconds up to a second might be sufficient time intervals for visual updates.

4.2 Architecture and design

Figure 4.3 shows the overall architecture of WallMon: the core runtime which glues together collectors and handlers through a push-based approach to data transfer. The collectors gathers data of interest, and the handlers take action upon the gathered data by for instance storing or visualizing it. This organization gives control over functionality specific to monitoring and end-to-end usage of data [6], while leaving data distribution to the core runtime. The core runtime notifies collectors when to collect data, and when data has been collected, the runtime will route the data to the appropriate handler. For instance, figure 4.3 shows a typical data flow: the core runtime routes data from multiple collectors, each running on a node in a cluster of computers, to a single handler running on a remote workstation.

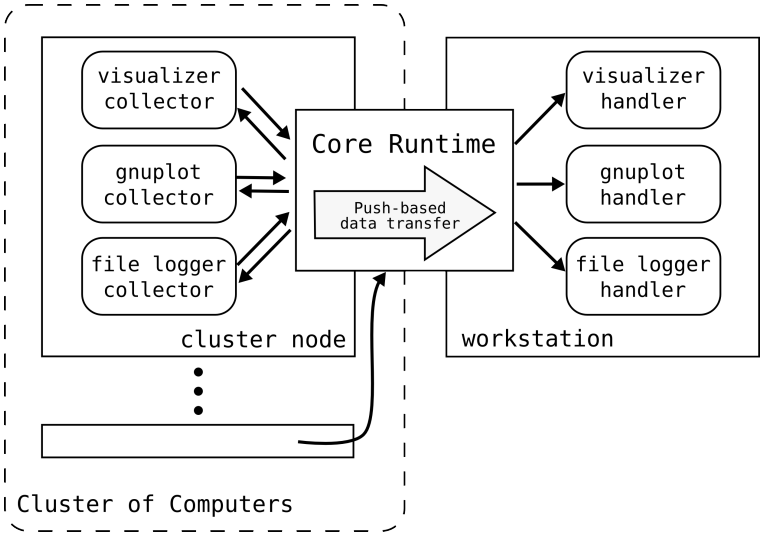


Figure 4.3: WallMon architecture and current collectors and handlers available: *visualizer* collects process-level data and visualize it using an information flocking inspired approach, *gnuplot* collects data about specified processes and generates different charts based on the data, and *file logger* also collects data about specified processes and simply writes the data to persistent storage.

Typically there is a one-to-one mapping between collectors and handlers, as shown in figure 4.3. For example, the data collected by the gnuplot collector might not be interpretable by the file logger handler. However, the system supports a M:N mapping of collectors and handlers. Figure 4.4 shows the flexibility of a M:N mapping of collectors and handlers in WallMon. In this example, the file logger collector is not used, and both remaining collectors each supply two handlers with data.

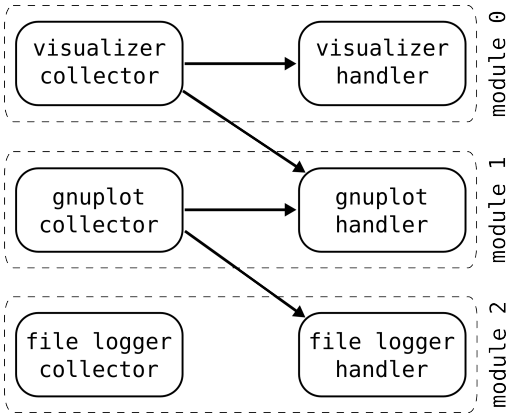


Figure 4.4: Flexible mapping of collectors and handlers in WallMon.

4.2.1 Core runtime

The design of the core runtime in WallMon is based on the client-server model, as shown in figure 4.5. The core runtime consists of an arbitrary number of clients and servers. Servers are independent of each other and a client might not be aware of all servers. The motivation for clients to be connected to multiple servers, is flexibility. For certain application domains, this model keeps the core runtime general, and places specific functionality in collectors and handlers. Figure 4.5 shows such a case: to make WallMon run and provide visualization on wall-sized, high-resolution tiled displays, collected data is sent to all nodes/tiles driving the high-resolution display. The client-server model results in a simpler system and lower latency than alternative approaches, such as hierarchical structures employed in [1] and [11]. On the other hand, a hierarchical structure scales to larger number of nodes.

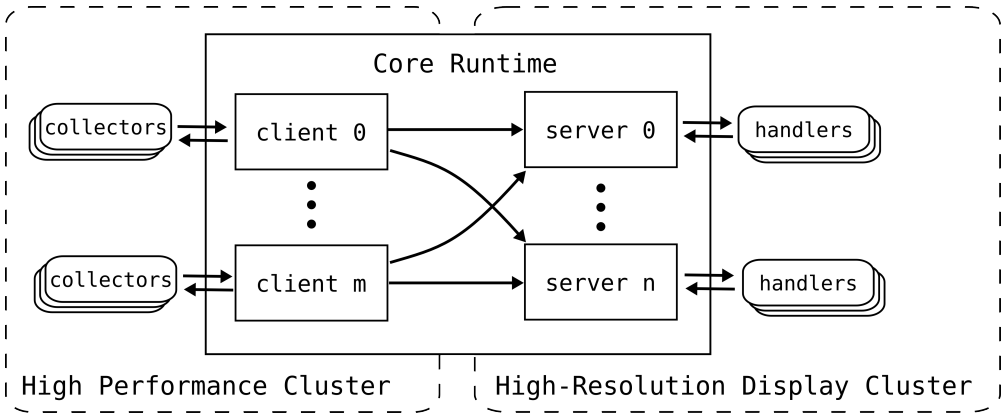


Figure 4.5: Design of core runtime in WallMon

4.2.2 Module system - collectors and handlers

Figure 4.6 exposes design details of WallMon's module system and its life-cycle. As the figure shows, a module consists of a collector and a handler. The life-cycle of a module starts when

a `module loader` loads it from persistent storage during runtime. When loaded into memory and installed in the core runtime, a `collector engine` interacts and manages the module's collector, while a `handler engine` carries out similar actions for the module's handler. All interactions are carried out through pre-defined interfaces. The specific responsibilities of the `collector engine` are scheduling and collecting data from collectors at user-defined intervals, and forward collected data to end-points (servers) specified by the collector. On the other hand, the responsibilities of the `handler engine` are routing incoming data to correct handlers, and invoking these handlers. The collector and handler engine are able to manage arbitrary many collectors and handlers, respectively.

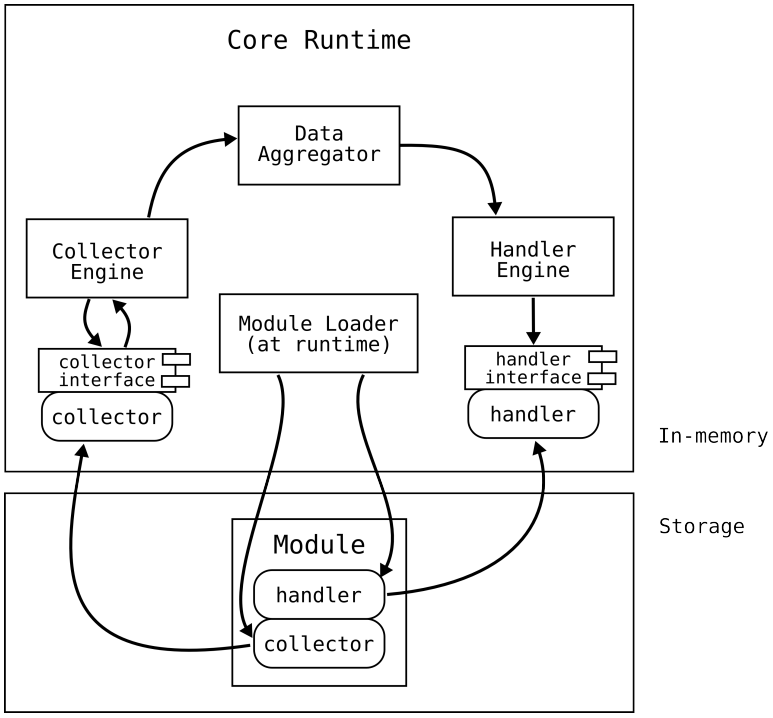


Figure 4.6: Design of module system and core runtime in WallMon

The main motivation for a module system in WallMon is a way of extending the system. This is important since monitoring is diverse, both when it comes to collecting data and taking action upon data. Among others, visualization, applying statistics to monitored data, higher level of abstractions for accessing monitored data, which metrics to monitor and how to monitor them, are ideas and questions that have been considered in WallMon. The module system allows for quick exploration and prototyping of different approaches.

Collectors and handlers are based on the inversion of control pattern [21]: they implement an interface shared with the core runtime, and do not manage a unit of execution, but wait for the core runtime to invoke them. When a collector or handler is invoked, the actions taken are defined by their implementation, and can be anything. Figure 4.3 shows the handlers and collectors currently available in WallMon.

Another characteristic of collectors and handlers is a guarantee of sequential execution: col-

lectors and handlers are never invoked concurrently. This guarantee removes the need for synchronization primitives inside collectors and handlers, such as mutexes and monitors. However, there might be necessary to spawn unit of execution within modules. In such scenarios, the user is responsible for handling concurrency.

4.2.3 Push-Based data transfer

In WallMon, clients push monitored data to server(s). This contrasts to many traditional monitoring systems in which servers pull data. For a system that provides near real-time data, a push-based approach has potential to alleviate load on the server-side. Instead of book-keeping when to pull data and carrying out pulling of data, the server is only focused on receiving data. On the other hand, servers have less control over rate and amount of incoming data in a push-based approach. This lack of control could result in clients overwhelming servers.

On the server-side, WallMon uses asynchronous I/O (AIO) for handling multiple persistent connections, one for each client. The primary reason for settling on AIO over an alternative approach where each client-connection is handled in a separate thread, is minimal overhead and simplicity. In AIO there is only a single thread managing all the connections, which eliminates the need for spawning and allocating resources for a new thread whenever a client connects to the server. It is unclear whether AIO or other alternative approaches would contribute the most to scalability in WallMon.

4.3 Implementation

WallMon is implemented in C++ and runs on Linux.

4.3.1 Core runtime and module system

Figure 4.7 exposes internal components of WallMon's core runtime and module system. The `scheduler` and `source` make up the client-side of the core runtime, and together with the collectors they make up the WallMon client, which is implemented as an UNIX daemon. The `scheduler` manages the collectors and puts collected data in a queue shared with the `source`, whose job is to forward data to the WallMon server(s). The server-side of the core runtime is similar to its client-side: a `sink` handles all the incoming connections and puts received data in a shared queue. A `router` takes data out of this queue and invokes appropriate handlers. Each of the shown components of the core runtime is implemented by a single thread. This could be a problem with regard to the `scheduler` and `router`. For example, a handler might block when invoked by the `router`. This would keep the `router` from fetching incoming data and invoking other handlers. A solution for this could be to implement the `router` with a thread-pool. However, this particular scenario has not caused problems in WallMon, and therefore has not been prioritized.

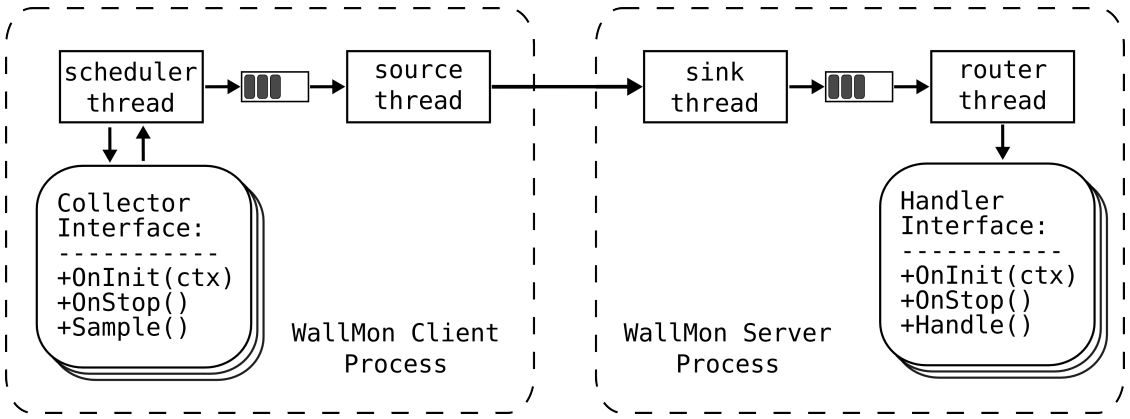


Figure 4.7: Implementation of core runtime and modules in WallMon

Modules in WallMon are represented as UNIX shared libraries. Figure 4.7 shows the interface required to be implemented by collectors and handlers. In common they have to provide the `OnInit` and `OnStop` functions that are invoked at the beginning and end of their life-cycle, respectively. while collectors have to provide `Sample` for collecting data, handlers have to provide `Handle` for handling data. The argument passed in `OnInit`, which is shared with the core runtime, is used to control certain aspects of the life-cycle, such as how often a collector is invoked.

4.3.2 Data transfer

Asynchronous I/O (AIO) in WallMon is implemented by using the `libev`[1] framework. `libev` is a cross-platform framework that automatically makes use of available AIO primitives on supported platforms. For example, if available, the `epoll` system call is used on Linux, otherwise `select` or `poll` is used as an alternative.

WallMon's network communication protocol relies on Protocol Buffers[2], a serialization format developed by Google. Most of the modules in WallMon also use Protocol Buffers internally. The Protocol Buffers library was chosen due to its simplicity and binary serialization. For example, compared to XML representation, binary serialization in Protocol Buffers is more compact. In addition, Protocol Buffers implement an encoding technique known as `varint`. `varint` will represent primitive types, such as integers, with fewer bytes if possible. For example, instead of using four bytes to represent a value that strictly need only one byte, `varint` will use only a single byte at the cost of one bit per byte. This extra bit per byte is used to determine when the value ends.

WallMon uses TCP as the underlying protocol for transfer of collected data. When a client connects to a server for the first time, the connection remains open in order to avoid further overhead of handshakes in the TCP protocol. TCP is used since WallMon does not assume that clients and servers are located on the same local area network. Hence, the network

[1] http://software.schmorp.de/pkg/libev.html
[2] http://code.google.com/p/protobuf/

between clients and servers in WallMon could be unreliable and the well-known guarantees provided by TCP are important.

4.3.3 Process-level data collection

In order to gather data about every process on a cluster of computers, WallMon uses the procfs file system present in most Linux environments. procfs is a virtual file system that, among others, keeps a file for each process on the system. These files are continuously updated with different types of metrics. The collector responsible for gathering process-level data, will open all the process specific files in procfs and read them during sampling of data. Listing 4.1 shows the sequential process of sampling process-level data in WallMon, combined with Protocol Buffers operations. Functionality not showed in the listing includes discovery of new processes and terminated processes.

Process monitoring of data in WallMon is done at the user-level. This has the advantage of not having to modify the kernel. However, kernel modifications can serve as an optimization, as reported in [11]. In the case of WallMon, where relatively large amount of resources are monitored, a kernel modification could have been justifiable.

Currently there is limited amount of filtering of data on the collector side. Data obtained from the procfs file system is parsed and transformed from ASCII to binary representation, such as integers, before being serialized by Protocol Buffers. One optimization could have been to filter out, for example data about processes whose resource consumption is below a certain threshold, inside the collector. On the other hand, the current approach is flexible for handlers, and simple.

```
def sample():
        foreach monitor in process_monitors:
                monitor.update()
                cpu_util = monitor.get_cpu_utilization()
                .. retrieve other performance metrics ..

                process_message.set_cpu_utilization(cpu_util)
                .. populate other Protocol Buffers fields ..

                outer_message.add_process_message(process_message)
        return outer_message.serialize()
```

Listing 4.1: Pseudocode demonstrating process-level sampling in WallMon. Each process is associated with a monitor, which contains and manages hooks to the procfs file system. The `update()` call on monitors make getters retrieve performance data related to the time-interval between this `update()` call and the previous.

4.4 Experimental design and setup

The platform for the experiments is described in section 2.2.

4.4.1 Microbenchmarks

The goal of the microbenchmarks is to provide isolated performance results of data sampling in WallMon, which is a critical component in WallMon that is continuously executed. This component has the potential interfere with other applications, and its performance must be investigated.

The microbenchmarks have been carried out on the root node of the Tromsø display wall cluster, which, as mentioned in section 2.2, executes 280 processes. In this section, process-level sampling refers to gathering of data about all present processes. For example, at the front-end, one process-level sample gathers data about 280 processes. Gathering data about all present processes might not generally be an ideal approach. However, for benchmarking it is suitable since it gives an indication of the systems' maximum potential in the area being benchmarked.

4.4.2 Macrobenchmarks

The goal of the macrobenchmarks is to provide performance results of WallMon's infrastructure as a whole.

WallMon setup and methodology

The setup for the benchmarks was as follows: the WallMon client and the WallMon server executed on all 29 cluster nodes. Each WallMon client was connected and continuously sent data to all 29 WallMon servers. During the benchmark, only one WallMon module was loaded and executed, the gnuplot module. This module's collector gathers process-level data on the client-side, while its handler stores the collected data in memory until all collectors are done, before it uses the data to generate charts (including the ones presented in section 4.5). As with the microbenchmarks, process-level sampling refers to gathering of data about all present processes. The reason for having each client connected to all servers is to test the limits of the system. Moreover, the microbenchmarks indicated that process-level sampling is a bottleneck in the system, hence increasing the load of other parts of the system is interesting.

The gnuplot collector was configured to sample the same amount of data required by the collector for the information flocking inspired visualization, meaning that the collector collected about twice as much data that was necessary. The reason for this choice was to make the results more comparable to WallMon's performance behaviors when running the information flocking inspired visualization.

The results of the macrobenchmarks present different performance metrics measured at increasing sampling rates. The results originate from the same monitoring session; all relevant performance metrics were measured during a single execution of the WallMon system. This was achieved by having collectors start sampling at one Hz, before steadily doubling the sampling rate up until and including 512 Hz. The number of samples for each sampling rate were as follows: for 1-32 Hz, 60 samples were carried out, for 32-128, 120 samples were carried out, and for 256 and 512 Hz, 300 samples were carried out. The reason for increasing

the number of samples with increasing sampling rate was to test the system to its limits. For each sampling rate in each of the different measurements, the results for the WallMon client are the average of all data samples from all 29 cluster nodes, while for the server, only some of the results take all servers into account (while the others take only one server into account). The reason for this is that only one server (gnuplot handler) generated the charts, although all servers executed and received equal amount of data during the monitoring sesssion.

Limitations

Much internal instrumentation were required to produce the measurements presented in this section. Moreover, since the measurements were obtained in one monitoring session, all instrumentation were active during this session. It is clear that the instrumentation comes with some overhead that could affect the performance behaviour of WallMon.

The configuration of the macrobenchmarks focus on sampling data at increasing rates. Another interesting configuration, which has not been carried out, could have been to maintain a constant sampling rate where the number of processes on the cluster were increased instead.

4.5 Results and discussion

This section presents and discusses the results from the microbenchmarks and macrobenchmarks described in section 4.4.

4.5.1 Microbenchmarks

Table 4.1 shows the time consumption of the different phases of sampling process-level data in WallMon. The results are the average of 10 000 consecutive samples carried out as fast as possible. *Reading from procfs* is the time it takes to gather data from procfs for all processes, while *parsing procfs data* filters out data not relevant and puts relevant data into pre-defined variables. *Protocol Buffers data insertion* is the act of copying the parsed procfs data into the data structures defined by Protocol Buffers. *Protocol Buffers data serialization* takes the Protocol Buffers data structures and transform them into a compact binary representation. *Other* includes everything that the aforementioned functionalities do not cover, such as overhead related to instrumentation that makes the microbenchmark possible. *Protocol Buffers data* ***de-****serialization* unencodes Protocol Buffers serialization output. This measurement was carried out after completing all the other aforementioned measurements. In this particular benchmark, the average sampling rate was 221 Hz.

The data representation of the Protocol Buffers library has also been microbenchmarked. Serializing one sample of process-level data at the cluster's root node results in 18210 bytes, which is an average of 65 bytes per process.

Functionality	Average time	Std dev	Min	Max
Reading from procfs	3.53 ms	0.002 ms	3.48 ms	9.27 ms
Parsing procfs data	0.80 ms	0.001 ms	0.77 ms	1.71 ms
Protocol Buffers data insertion	0.11 ms	0.001 ms	0.08 ms	0.45 ms
Protocol Buffers data serialization	0.04 ms	0.0 ms	0.035 ms	0.19 ms
Other	0.05 ms	0.0 ms	0.02 ms	0.67 ms
Total	4.53 ms ($\approx$ 221 Hz)	0.004 ms	4.44 ms	12.29 ms
Protocol Buffers data **de-**serialization (not part of process-level sampling)	0.08 ms	0.0 ms	0.074 ms	0.1 ms

Table 4.1: Microbenchmark of process-level sampling in WallMon. One sample includes sampling data about all present processes, which at the time was 280. Data serialization and de-serialization includes all 280 processes. The results are the average of 10 000 consecutive samples.

The microbenchmarks show that reading data from the procfs file system is the most expensive operation of sampling process-level data. Reading data from procfs causes a context switch from user-level to kernel-level. Since each process has its own entry in procfs, and WallMon opens three virtual files for each process, three context switches will occur for every process included in a sample. The microbenchmarks also show that parsing the raw procfs data has some cost. The parsing is done by the `sscanf` function provided by a library in the C programming language. It might have been faster to use other approaches, such as a parser tailored for the format of procfs.

The Protocol Buffers library causes minimal overhead in the microbenchmarks. Its fast and has a compact data representation. On average in the microbenchmarks, serializing a n byte value with Protocol Buffers takes up only n bytes. Compared to XML's human readable format, Protocol Buffers provide a more compact representation in WallMon.

4.5.2 Macrobenchmarks

The results and discussion for the macrobenchmarks are divided into three subsections: traditional and general measurements such as CPU, memory and network bandwidth, measurements specific to collectors and handlers in WallMon, and measurements related to data aggregation in WallMon.

CPU, memory and network bandwidth

Figure 4.8 shows that up until and including 64 Hz, the client's CPU load doubles whenever the sampling rate doubles. However, when moving from 64 to 128 Hz this pattern stops. What happened during this benchmark, was that the single-threaded collector sampled data too slow (on most samples) for the WallMon client to re-schedule it at non-overlapping intervals. This is supported by results discussed later, and also the large amount kernel-level execution time compared to user-level execution; its primary cause is reading from the procfs file system. While sending data over the network also contributes to kernel-level execution,

reading from procfs contributes the most: data read from procfs is represented as ASCII strings compared to compact serialized representation which is sent over the network, and process-level sampling requires three context switches for each process in the sample, while sending data over the network requires one or a handful of context switches. Due to the procfs overhead, it might have been justifiable to implement a kernel module, as is done in [11]. Such a module has the potential to eliminate the cost of context switching.

Figure 4.9 shows that the CPU utilization of the WallMon server has a larger amount of user-level execution compared to kernel-level execution. The user-level execution is caused primarily by the gnuplot handler which uses several in-memory data structures to store data, while the kernel-level execution is primarily caused by receiving data from the network.

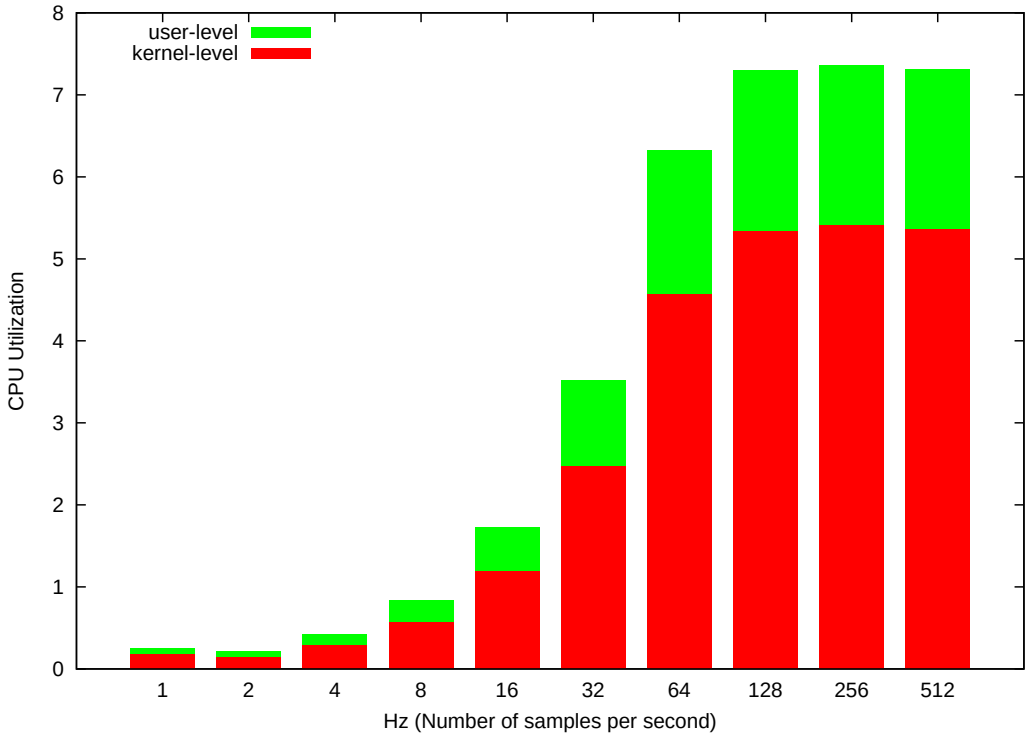


Figure 4.8: CPU utilization of WallMon client. Maximum CPU utilization (100 percent) implies constant usage of all available CPU cores.

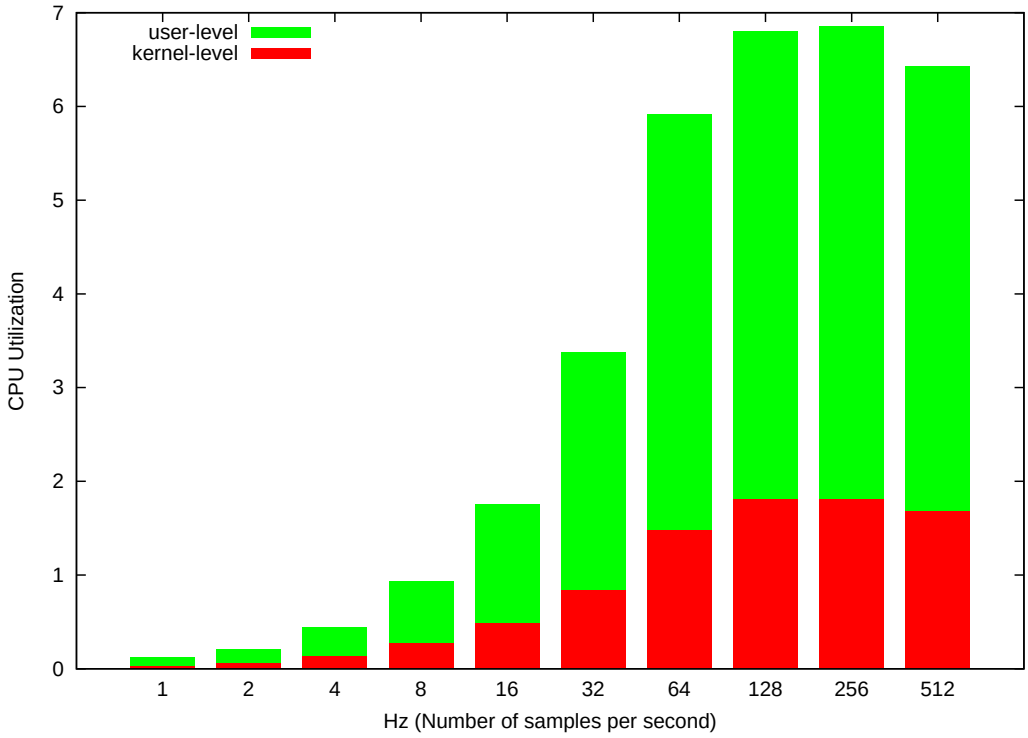


Figure 4.9: CPU utilization of WallMon server. Maximum CPU utilization (100 percent) implies constant usage of all available CPU cores.

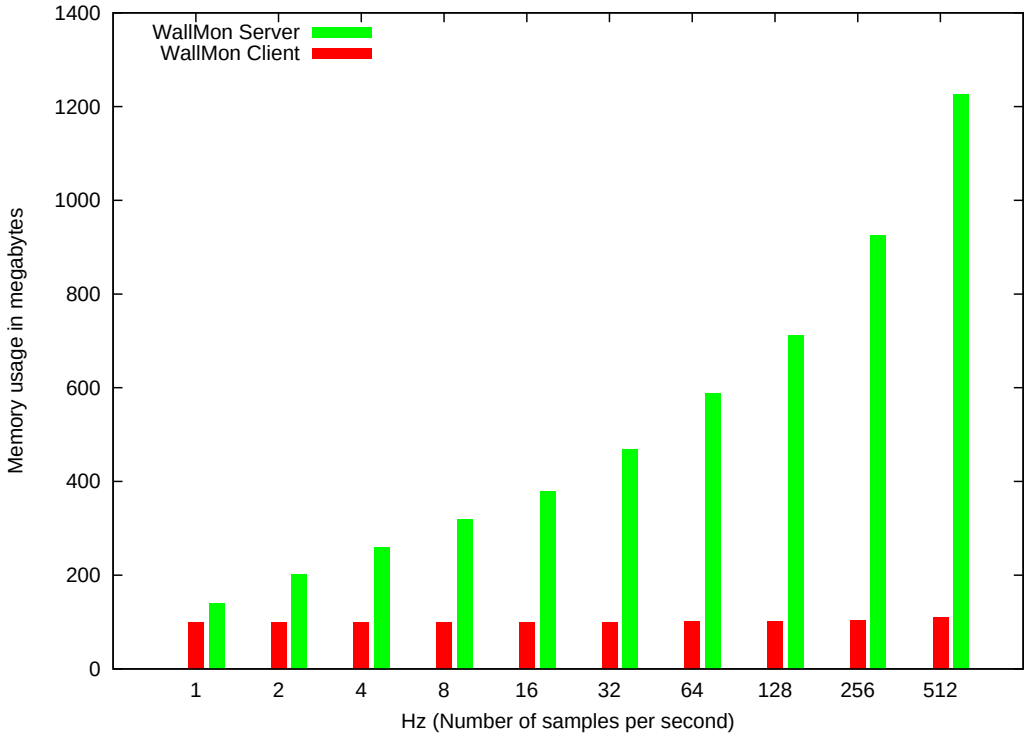


Figure 4.10: Memory usage of WallMon client and server. The measurements include the memory usage of collectors and handlers.

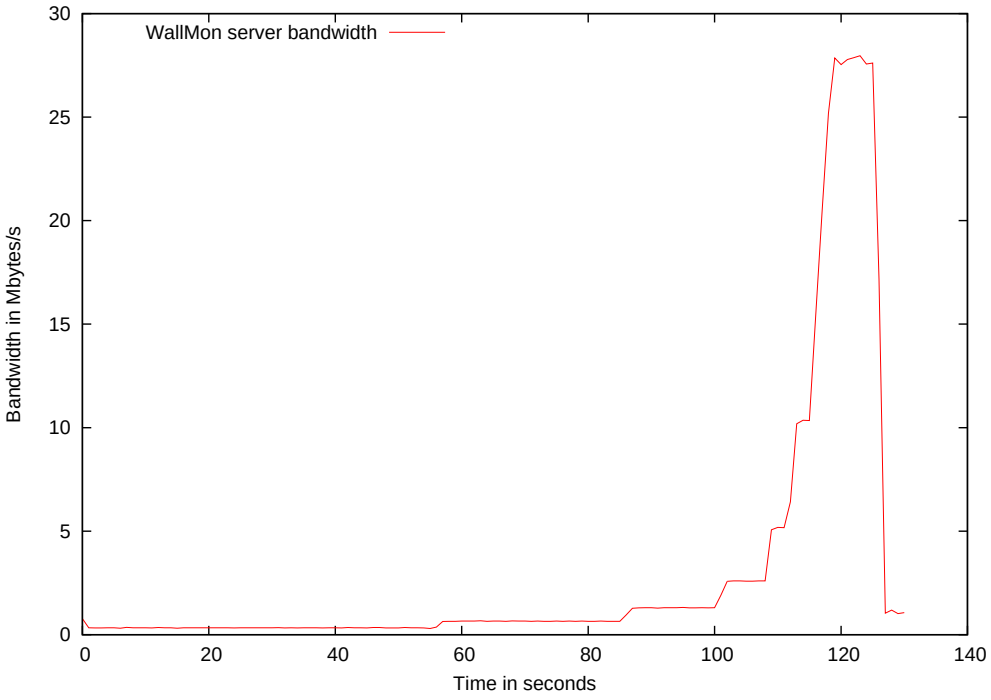


Figure 4.11: Network bandwidth of WallMon server. The bandwidth was measured internally on one WallMon server, and the data is based on the number of bytes received by the WallMon server. The time along the horizontal axis represents the time frame for the single macrobenchmark.

The memory usage in figure 4.10 shows that the WallMon client has a constant memory usage during the monitoring session, except for a slight increase at the end. On the other hand, the server's memory usage almost grows exponentially due to the increased sampling rate combined with inefficient in-memory storage of data at the server side. Compared to other cluster monitoring systems, such as [1] and [11], WallMon has a relatively large memory footprint.

The bandwidth of the server shown in figure 4.11 correlates with the CPU utilization and memory usage: as the sampling rate increases, the memory rate increases as expected. The bandwidth of the WallMon client has not been measured directly, however, given the setup of the macrobenchmark where each client would send data to all servers, its bandwidth will be about equal to the server's bandwidth. The sudden drop in bandwidth towards the end of the benchmark indicates that collectors do not finish at exact points in time.

Collector and handler measurements

Figure 4.12 shows the actual sampling rate of collectors, compared to the ideal and specified rate. This result supports that data sampling is the bottleneck in WallMon when sampling data at high rates; up until and including 64 Hz the collectors are able to sample at its specified rate, however, at higher rates they are not able to keep up. On average, the collectors maximum sampling rate is around 90 Hz. This result does not match the

microbenchmarks, where an average sampling rate of 221 Hz was achieved. Although the gnuplot collector is more complex and computational heavy compared to the simulated collector in the microbenchmarks, one would expect a smaller difference in sampling rate between the microbenchmarks and macrobenchmarks.

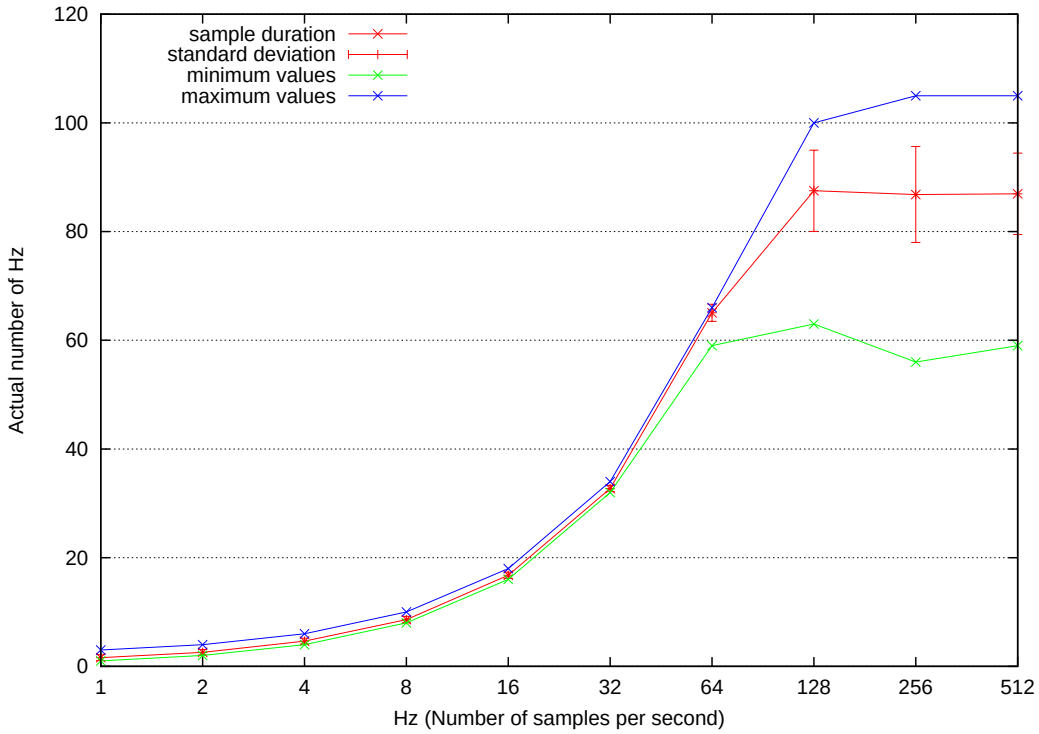


Figure 4.12: Actual sampling rate of collectors. Measured as the number of times per second a collector has been invoked, and hence sampled data.

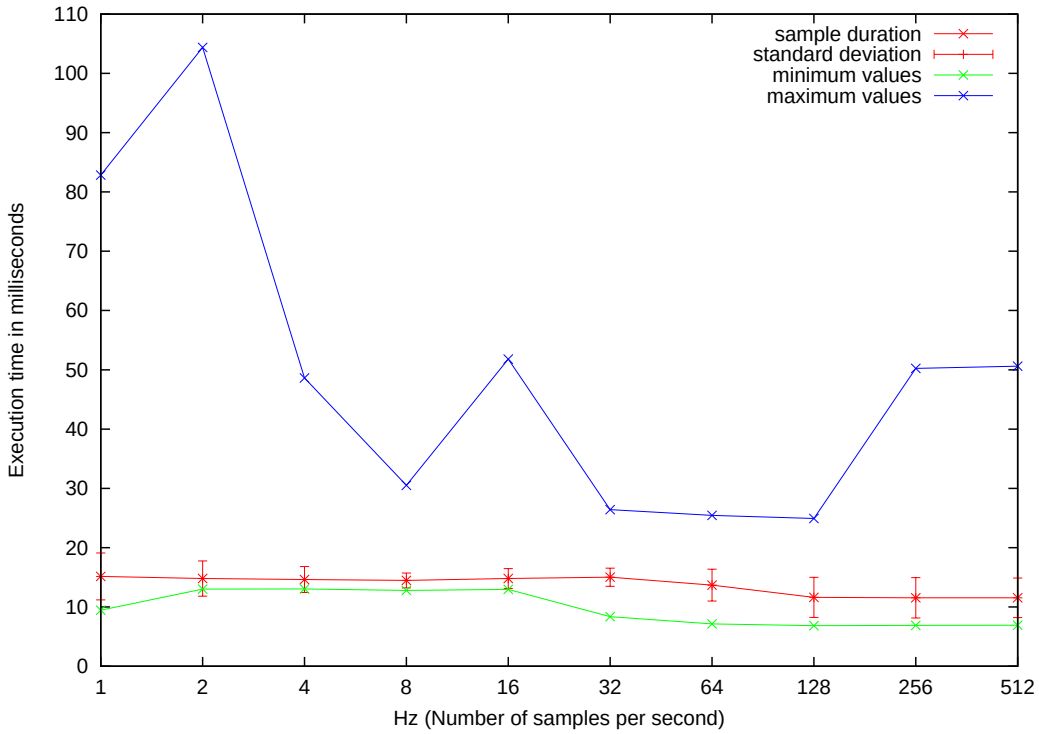


Figure 4.13: Execution time of gnuplot collector. The measurements were obtained by comparing timestamps taken right before invoking the collector and right after the invocation.

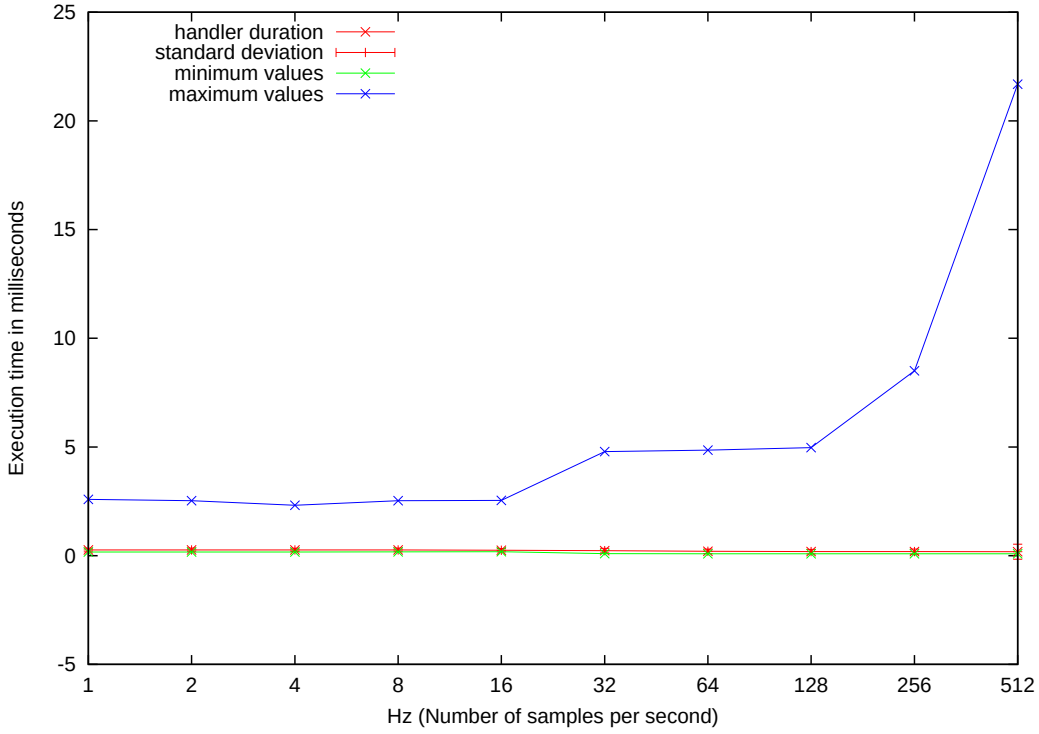


Figure 4.14: Execution time of gnuplot handler. The measurements were obtained by comparing timestamps taken right before invoking the handler and right after the invocation.

Figure 4.13 and 4.14 show the execution time of the gnuplot collector and handler, respectively. The execution time in both results remain relatively stable, at 15 milliseconds for the collector and less than a millisecond for the handler, except from the maximum values. Interestingly, for the collector the execution time slightly decreases at higher sampling rates. This could be caused by increased cache hit ratio when scheduled more frequently.

Data aggregation

Results presented in figure 4.15 measure the latency of data aggregation in WallMon. The latency includes encoding and decoding of data, and it remains stable at the different sampling rates, indicating that data is not queued up on either the client or server side, and that the network is not congested. The average aggregation latency starts out on 15 milliseconds and gradually decreases to 10 milliseconds. Taking into account previously discussed results, table 4.2 estimates an average total data propagation latency in WallMon.

Operation	Average time
Collector execution	20 ms
Data aggregation	15 ms
Handler execution	1 ms
Total propagation latency	36 ms

Table 4.2: Estimated total average data propagation latency in WallMon.

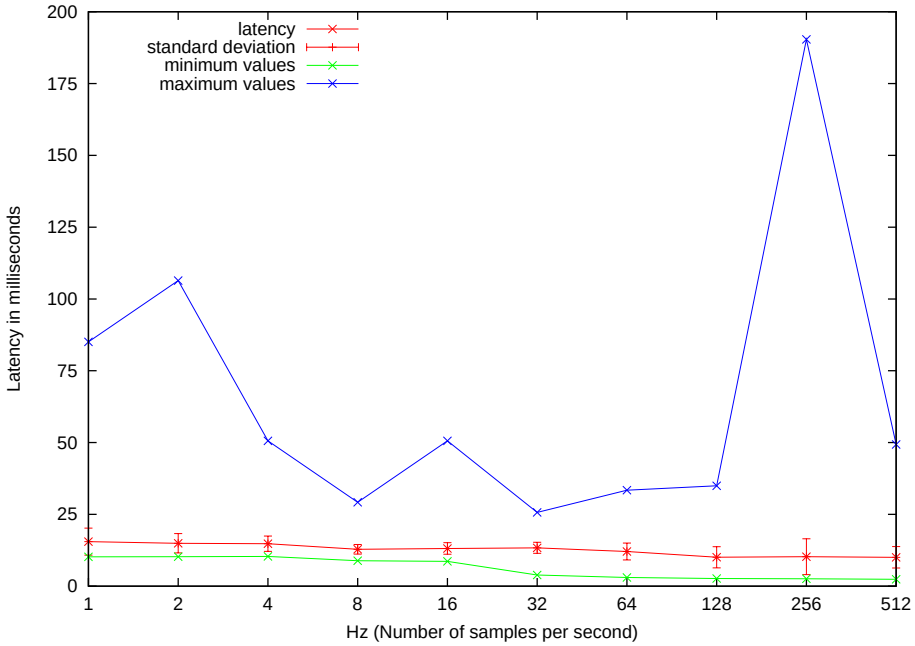


Figure 4.15: Network latency of data aggregation. The measurements are based on physical clocks on different computers: one timestamp right after the invocation of a data sampling routine of a handler, and the other on the WallMon server side (right after decoding the encoded Protocol Buffers message).

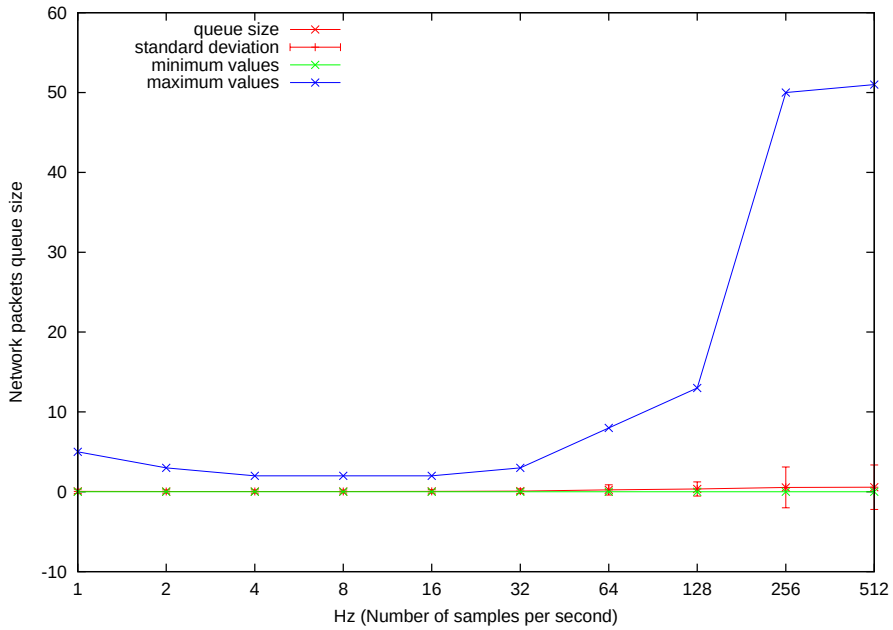


Figure 4.16: Size of WallMon server's network packet queue. This queue holds received network packets that have not been processed by the server. The measurements for the network packet queue for the WallMon client is not showed since it did not accumulate any items; for all samples, all items put in the queue were processed before the next measurement.

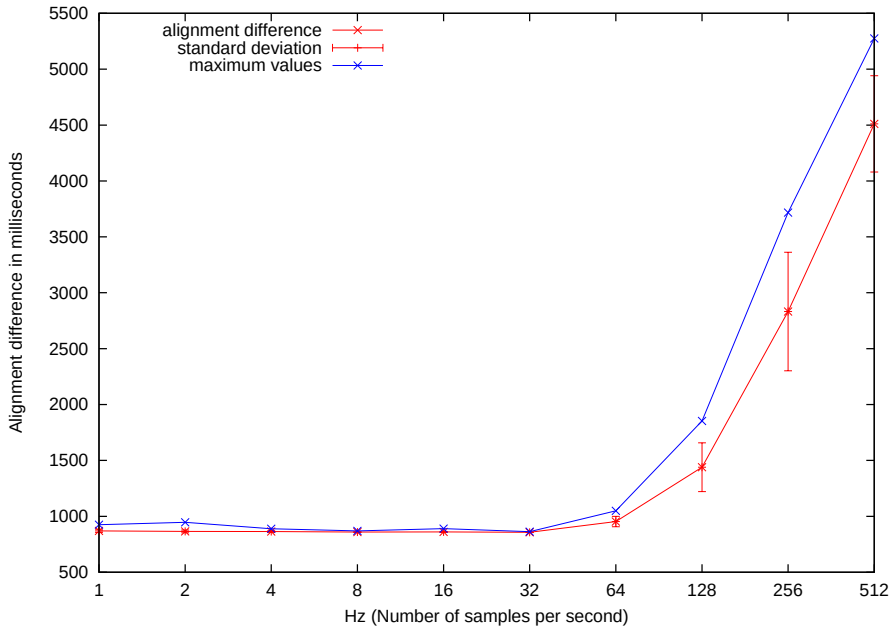


Figure 4.17: Time alignment of collectors based on server timestamps. The measurements were obtained by comparing when WallMon client messages with equal sequence number arrived at the WallMon server. The *alignment difference* graph represents the difference between messages of equal sequence number arriving first and last of the WallMon server, which therefor represents the worst-case (not taking into account messages in between).

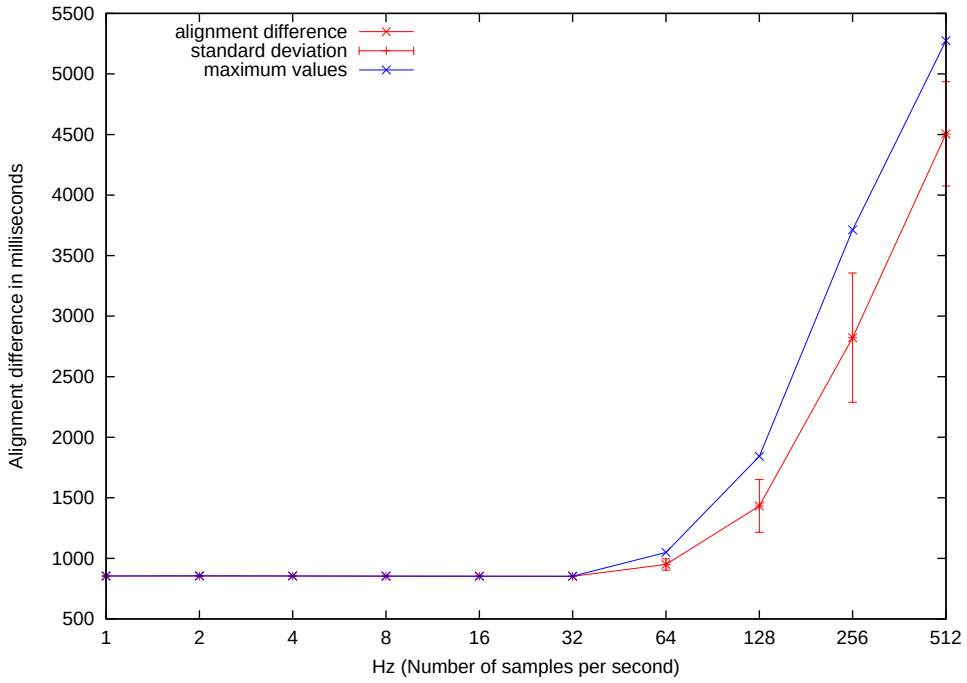


Figure 4.18: Time alignment of collectors based on client timestamps. The measurements were obtained in a similar way as the results in figure 4.17, except that timestamps from the different clients were used.

Results in figure 4.16 show and support that data did not queue up on either the client or server side during the benchmark. The network packet queue on the server slightly grew towards the end of the benchmark, while the client's queue remained empty at every sample during benchmark.

The results in figure 4.17 and 4.18 attempt to establish to which degree the 29 collectors were aligned/synchronized in time during the benchmark. Ideally, one would have the collectors to sample data at the exact moment in time at every sample. These results represents the worst case: for each set of collector messages that were expected to arrive simultaneously, the time difference between the one that arrived first and the one that arrived last was taken into account. The results in figure 4.17 measures alignment by relying on timestamps taken at the server side, while results in figure 4.18 relies on timestamps originating from the clients. Both scenarios present consistent results, the time difference starts out at about 850 milliseconds, and remains stable up and until 32 Hz. For sampling rates above 32 Hz the collectors rapidly get out of synchronization in time. This occurs due to a combination of collectors not being able to sample data fast enough, and that WallMon does not in any way attempt to synchronize collectors in time. Although these results measure the worst case, it shows that sampling data at high rates in WallMon might be unsuitable for application domains that require events on different nodes to be caught and observed with millisecond granularity.

4.5.3 Collector-handler model and module system

In WallMon, the collector-handler model is realized through the module system. The module system's support for arbitrary many modules executing simultaneously and a guarantee of sequential execution within a module's handler and collector, is an abstraction that simplifies. These guarantees and the programing model offered have similarities to the model implemented by Google's MapReduce framework [22]. Although MapReduce targets a different domain, collectors in WallMon and `map` in MapReduce both transform and forward data to a new transformation, and handlers and `reduce` usually aggregates large amount of data into something smaller and/or comprehensible. The inversion of control pattern implemented by modules and the guarantee of sequential execution also matches MapReduce.

One limitation with the module system is that users have to implement all functionality specific to gathering and usage of data. This might reduce productivity, however, it adheres to the end-to-end argument [6], which can be considered applicable to the diverse and performance sensitive activities of cluster monitoring.

4.6 Conclusion and future work

This chapter presented and discussed the underlying infrastructure provided by WallMon. The ideas of the infrastructure is centered around the collector-handler model and real-time aggregation of data. Its design is based on the client-server model, and its architecture is extensible through a module system that encapsulates functionality specific to monitoring (collectors) and actions taken upon gathered data (handlers). Experiments show that sampling data with process-level granularity comes at an acceptable cost.

Future work for WallMon's infrastructure include adding support for exploiting multiple cores when executing collectors and handlers. This functionality is currently single-threaded. Moreover, it should be investigated to which degree a synchronization scheme between collectors would be beneficial. In the current approach, there is no synchronization between collectors, which implies that collectors could drift over time, making them sample data at significantly different points in time. However, for certain application domains, such as visualization, the current approach might be sufficient.

Chapter 5

Data Visualization and Use of WallMon

Cluster monitoring is diverse, especially when it comes to taking action upon gathered data. Such actions depend on the application domain, and common actions include logging [23], visualization [2, 1, 13] and applying statistics [10, 24]. In WallMon, the focus has been to apply and explore visualization techniques. A strong motivating factor for this choice was the presence of high-resolution display wall with support for interactive user input.

This section starts by presenting the different visualization techniques explored in this thesis, before explaining the the design and implementation of the primary technique explored. This particular technique is then evaluated through a set of case studies, before the technique is discussed and concluded.

5.1 Idea

The thesis started out exploring visualization techniques based on charts, before settling upon and focusing on a technique inspired by the concept of information flocking [3].

5.1.1 Charts

The thesis started out exploring visualization techniques based on charts, such as the prototype shown in figure 5.1. The goal of these charts were to provide a high-level view of cluster load, before later on exploring ways of interactively accessing details not present in these high-level views. The bar chart in figure 5.1 provides a useful high-level view of the CPU utilization on a cluster: the chart shows the relative, aggregated user- (green part of bars) and kernel-level (red part of bars) CPU utilization of processes of equal name, and aggregated CPU utilization of all present processes (located on top of the chart). With this chart, users can quickly observe the total CPU load on a cluster, and see which processes that are responsible for this load.

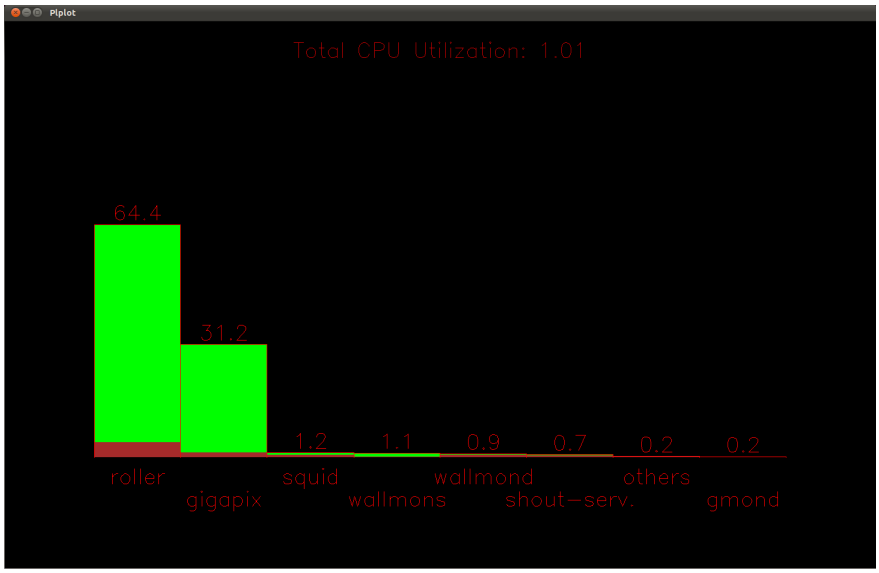


Figure 5.1: Early visualization prototype based on bar charts in WallMon. The chart shows the relative, aggregated user- (green part of bars) and kernel-level (red part of bars) CPU utilization of processes of equal name on the Tromsø display wall cluster, and aggregated CPU utilization of the entire cluster (located on top of the chart). The charts are sorted in descending order, with the process group with the most utilization to the left. Only 8 process groups are shown in the chart; the group *others* contains aggregated utilization for all other process groups.

Although the prototype in figure 5.1 served as a useful starting point for exploring visualization approaches based on charts, there were concerns. Firstly, much research on chart based visualization, especially in the field of cluster monitoring, have been carried out; it would be difficult to make WallMon distinguish itself and come up with a novel approach. Secondly, an approach based on information flocking appeared to be interesting and, to some extent, novel.

5.1.2 Information flocking

Data gathered by WallMon is visualized by an approach inspired by information flocking [3], a technique that leverages humans' ability to see patterns in color and motion. The approach was motivated by WallMon's goal for sampling fine-grained data at near real-time rate. Several prototypes early on in the WallMon project showed that traditional charts and lists were unsuited for achieving such goals. The primary obstacle experienced with traditional approaches was the difficulty of providing a high-level view of and the relationships between monitored processes. However, traditional charts and lists might be suitable for visualizing detailed information about a single process, or a handful of processes.

Several figures throughout the thesis, including figure 1.1 and 5.2, show the primary components of WallMon's visualization. At the heart of the visualization is a chart with moving entities, where an entity's shape represents a performance metric, such as CPU or memory, and its color represents a process name. The chart's horizontal axis shows the relative

resource utilization of the different performance metrics, while its vertical axis represents something specific to the different metrics.

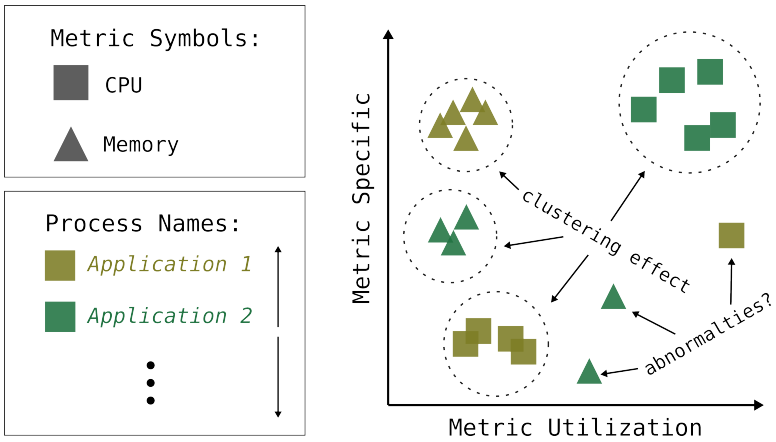


Figure 5.2: Illustration of information flocking inspired visualization and its user interface

Another component of the visualization is an interactive list, where each entry groups together processes with equal process name. When accessing one of these entries using a touch gesture, another list appears. This list holds detailed data about each process under this process name, such as how many threads the process is running and on which host it is running.

5.2 Architecture

Figure 5.3 shows the overall architecture for the infrastructure behind WallMon's visualization. As the figure shows, the collector-handler model provides the platform for the visualization: arbitrary many collectors supply a handler with data, which in turn filters and forwards the data to a `visualization engine`. The `visualization engine` manages and schedules arbitrary many entities, and provide the entities with abstractions similar to abstractions found in very simple game engines, such as self-defined coordinate systems. Entities do the actual rendering to display. The `visualization engine` also receives input from an `event system`, which forms the basis for interactivity in the visualization. The details of this component is described in chapter 6.

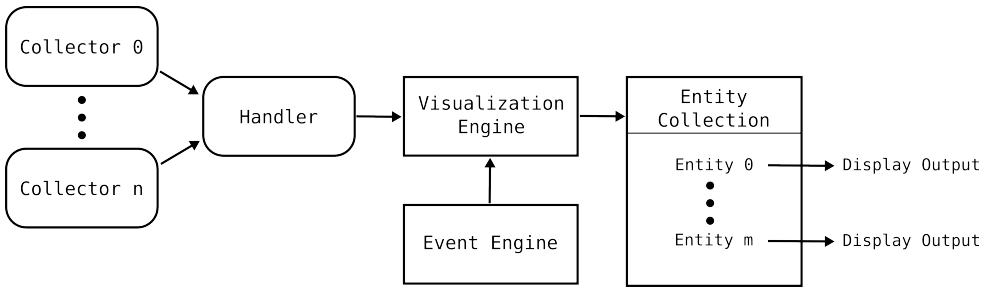


Figure 5.3: Architecture of WallMon's visualization.

The motivation for using a game-like approach to orchestrate the visualization, is reuseability and reducing complexity. By having a uniformed way, in the sense of entities, to add new components to the visualization, it is simple to make changes and add new components. The reduction of complexity is also present when it comes to event management and providing an interactive visualization. In short, an entity is provided with a high-level interface for receiving events. This is further detailed in chapter 6.

There are several reasons for not using an already existing game engine, or a system that provides similar abstractions to the `visualization engine` component in figure 5.3. Firstly, performance is critical in WallMon's visualization. When developing software from scratch, one have better control over performance. Secondly, to make WallMon's visualization run on a display wall it was necessary to have access to low-level OpenGl calls. This might not have been available in a simple game engine. Lastly, the abstractions provided by the `visualization engine` are few and simple, and therefore might not necessary require an existing game engine.

5.3 Design and implementation

WallMon's visualization is written in C++, and runs on top of the underlying infrastructure provided by WallMon. This infrastructure is described in chapter 4.

5.3.1 Information flocking visualization

Figure 5.4 shows and summarizes WallMon's visual implementation of information flocking. The figure shows that WallMon implements four different performance metrics, each represented with its own geometric symbol: CPU with a square, memory with a triangle, network with a diamond, and storage with a 8-sided polygon. Each of these symbols represents a performance metric for a specific process, meaning that a single process, executing on some cluster node, has one of each of these symbols associated with it.

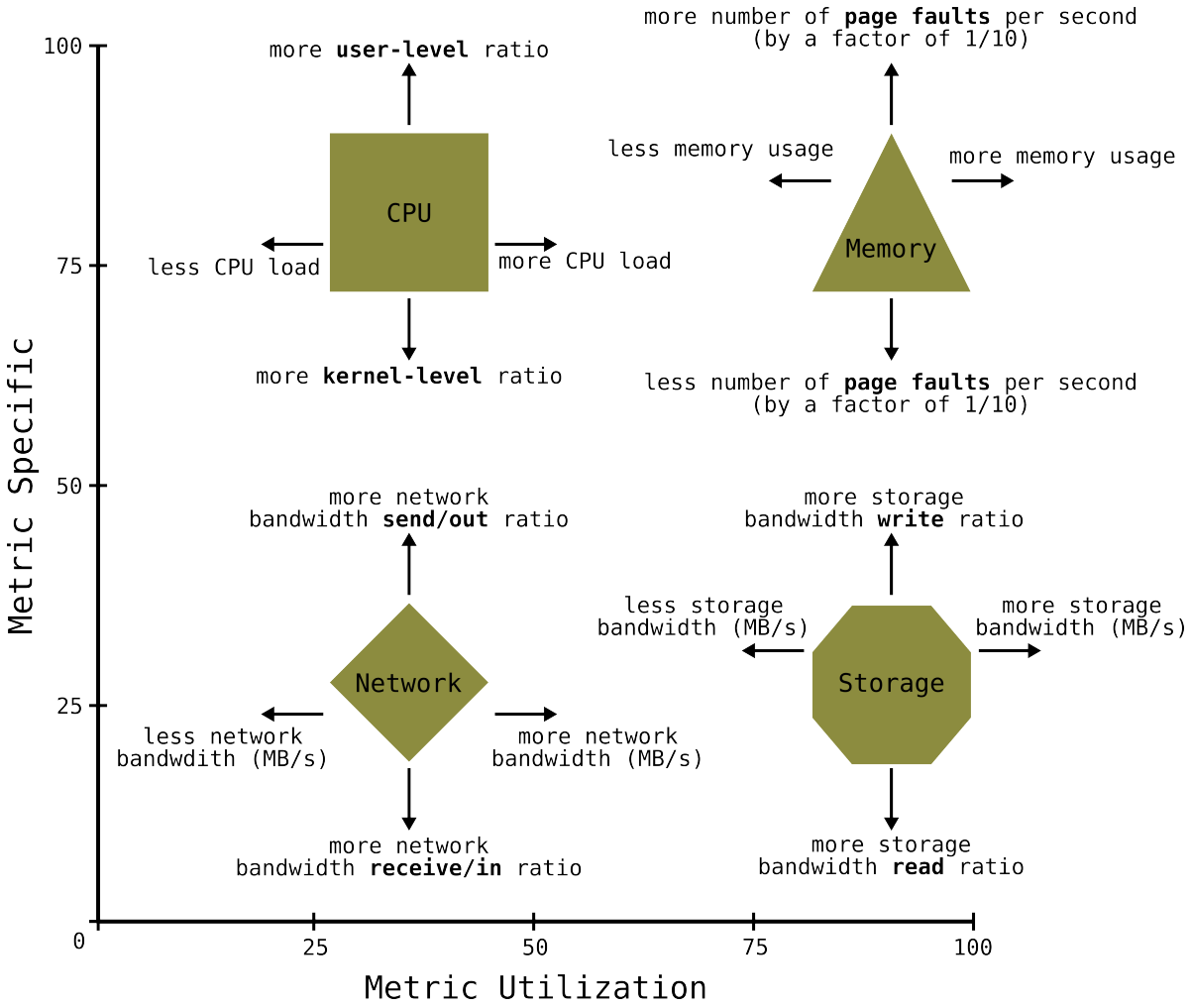


Figure 5.4: WallMon's visual implementation of information flocking. The figure shows the different performance metrics implemented, and summarizes the intended meaning of them.

The performance metrics of figure 5.4 are contained within a two-dimensional space, where the horizontal axis represents utilization, and the vertical axis represent something specific to the performance metrics. Table 5.1 describes the details of these axes for the four different performance metrics. Two key characteristics of this two-dimensionally represented chart, is generality and macro-view of data. Generality is achieved since it is straightforward to a add additional metric symbols to the chart, while a macro-view of data is provided since each of the metric symbols can only visualize a limited amount of data.

Metric	Symbol	Utilization (horizontal axis)	Specific (vertical axis)
CPU	Square	Dynamically detects and takes number of cores, both physical and logical, such as HypeThreading, into account. For example, 100 percent utilization means exclusive usage of all present cores during a measurement interval. Does not say anything about the distribution of execution time among the different cores.	Ratio between user-level and kernel-level execution. The more to the top of the axis implies more user-level usage, and vice versa. For example, in the middle of the axis represents equal share between user- and kernel-level execution. Does not say anything about the distribution of execution time among the different cores.
Memory	Triangle	Dynamically detects and takes total amount of memory into account. For example, with 4 GB of total memory, 50 percent utilization represents a usage of 2 GB memory for a particular process.	Number of page faults per second divided by 10. This measurement is a sum of minor page faults (no external storage access triggered) and major page faults (external storage access performed). The division by 10 allows for visual support for higher number of page faults.
Network	Diamond	Uses a fixed limit of 50 MB/s as a base for calculating utilization. For example, 50 percent utilization implies a bandwidth usage of 25 MB/s for a particular process.	Ratio between amount of data sent and received on the network. The more to the top of the axis implies more data sent than received, and vice versa. For example, in the middle of the axis represents equal share between amount of data sent and received.
Storage	Polygon	Similar to network metric, except bandwidth is towards persistent storage. Also uses a fixed limit of 50 MB/s.	Similar to network metric, except that bandwidth is towards persistent storage.

Table 5.1: Detailed description of performance metrics in WallMon's visualization.

Another component of the visualization is an interactive list, where each entry groups together processes with equal process name. When accessing one of these entries using a touch gesture, another list appears. This list holds detail data about each process under this process name, such as how many threads the process is running and on which host it is running. This component is further described in chapter 6.

5.3.2 Scene abstraction and display wall support

In WallMon, a scene is an abstraction that hosts and provides entities with a virtual coordinate system. Scenes are implemented by the visualization engine shown in figure 5.3, and they are created during the start-up phase of WallMon's visualization, making it easy to rearrange scenes. Figure 5.5 shows an example of the scene abstraction. This example is similar to how scenes are arranged in the implementation of WallMon's visualization.

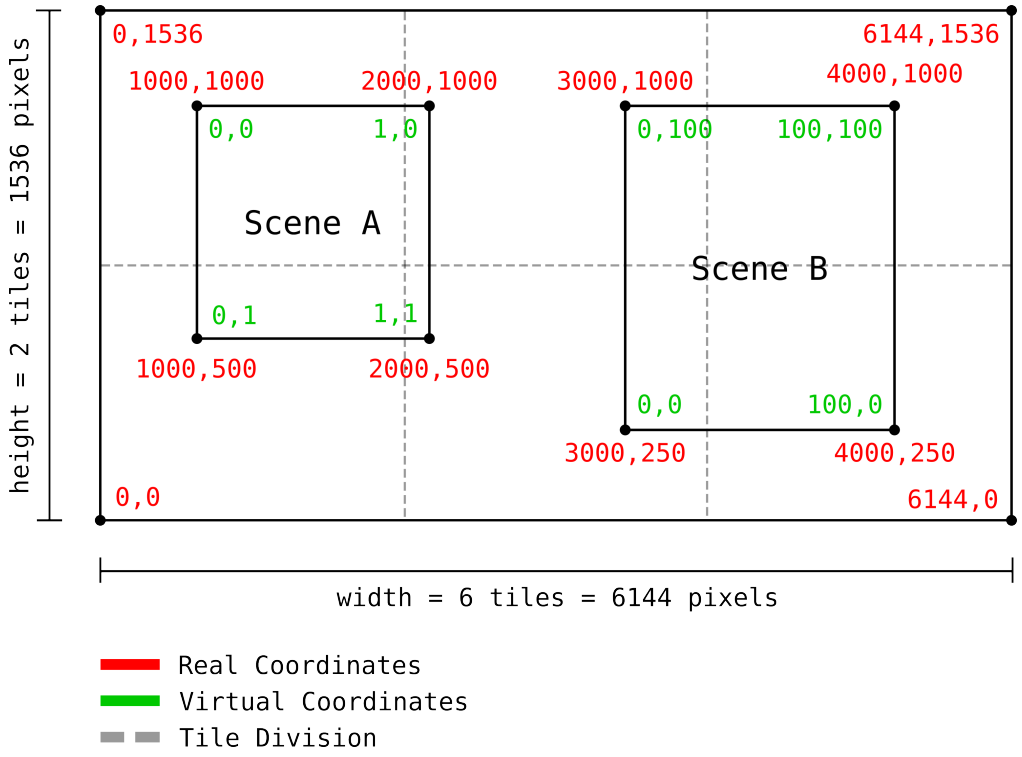


Figure 5.5: Example of scene abstraction in WallMon. The example shows two scenes on a display surface spanning 6144x1536 pixels. Both scenes employ a virtual coordinate system specific to the particular scene and the entities hosted within the scene.

The figure shows two scenes spanning six tiles of a display wall, three tiles in width and two tiles in height. Each scene provides its set of entities with a specific virtual coordinate system, making it simple to move and scale the scenes with regards to real coordinates without changing the implementation of the entities.

As the tile division in figure 5.5 shows, the example is executed across six tiles on a display wall, where each tile displays its portion of the visualization, which becomes coherent with the presence of all six tiles. In WallMon, this is achieved with a relative straight-forward approach: each tile, that is, each visualization engine, is aware of and uses its relative position on the display wall to adjust which area to display. Implementation-wise, this is achieved by `glOrtho()` provided by OpenGl. A downside of this approach is that each tile receives and processes the same amount of data, which would be all data related to the entire visualization. On the other hand, it is a simple approach.

5.3.3 Entities and visualization engine

Entities provide visual output, are hosted within a scene, and are executed by the visualization engine. The interaction between the visualization engine and entities happen through a shared interface described in the following listing:

- `OnInit()`. First call to the entity. Only called once.
- `OnLoop()`. Includes logic, such as calculating collisions between entities.
- `OnRender()`. Display output via OpenGl.
- `OnCleanup()`. Last call to the entity. Only called once.

The visualization engine executes entities sequentially, and listing 5.1 shows its logic in pseudocode. `process_touch_events()` and `run_touch_event_callbacks()` in the listing are related to the interactivity part of the visualization, which is described in chapter 6. The visualization engine is implemented as a thread, and it interacts with the handler thread (figure 5.3) through a thread-safe API.

```
def visualize_forever():
  initialize_opengl()
  create_scenes()
  set_display_area() //for display wall support
  while true:
    foreach scene in scene_list:
          scene.run() //executes calls to entities within scene
          process_touch_events()
          run_touch_event_callbacks()
          update_display_surface() //swaps and updates framebuffer
```

Listing 5.1: Pseudocode demonstrating the visualization engine.

5.4 Evaluation

5.4.1 Methodology

This section evaluates WallMon's visualization through performance benchmarks and case studies. While the performance benchmarks present tangible performance numbers, the case studies of the visualization are subjective. The goal of the case studies is to show the usefulness of the visualization, which in this section is defined as the ability to show an overall performance view, and the ability to show and let users discover performance patterns.

5.4.2 Microbenchmarks

The microbenchmark shown in figure 5.6 measures the time consumption of the major parts of the visualization engine. The sample number along the horizontal axis in the benchmark's chart represents the current iteration of the main loop in the visualization engine. This loop is shown in listing 5.1. The vertical axis represent amount of time in milliseconds that a

particular part within the visualization engine consumed for a particular sample. During the benchmark, the total number of processes on the cluster were **5190**, resulting in a total number of **20894** entities in WallMon's visualization. Also, the frame rate of the visualization remained stable at **35** frames per second during the benchmark. The following list describes the parts of the visualization engine that have been measured:

- **Clear Screen**: erases all previously data rendered by using the OpenGl call `glClear()`.
- **Run Scenes**: schedules and executes entities.
- **Process Events**: receive events from event engine (figure 6.2), and process them, making event callbacks ready.
- **Run Event Callbacks**: execute event callbacks of entities.
- **Swap Frame Buffer:** updates the screen/frame buffer by using the SDL/OpenGl call `SDL_GL_SwapBuffers()`.
- **Other**: includes overhead related to instrumentation for the benchmark.

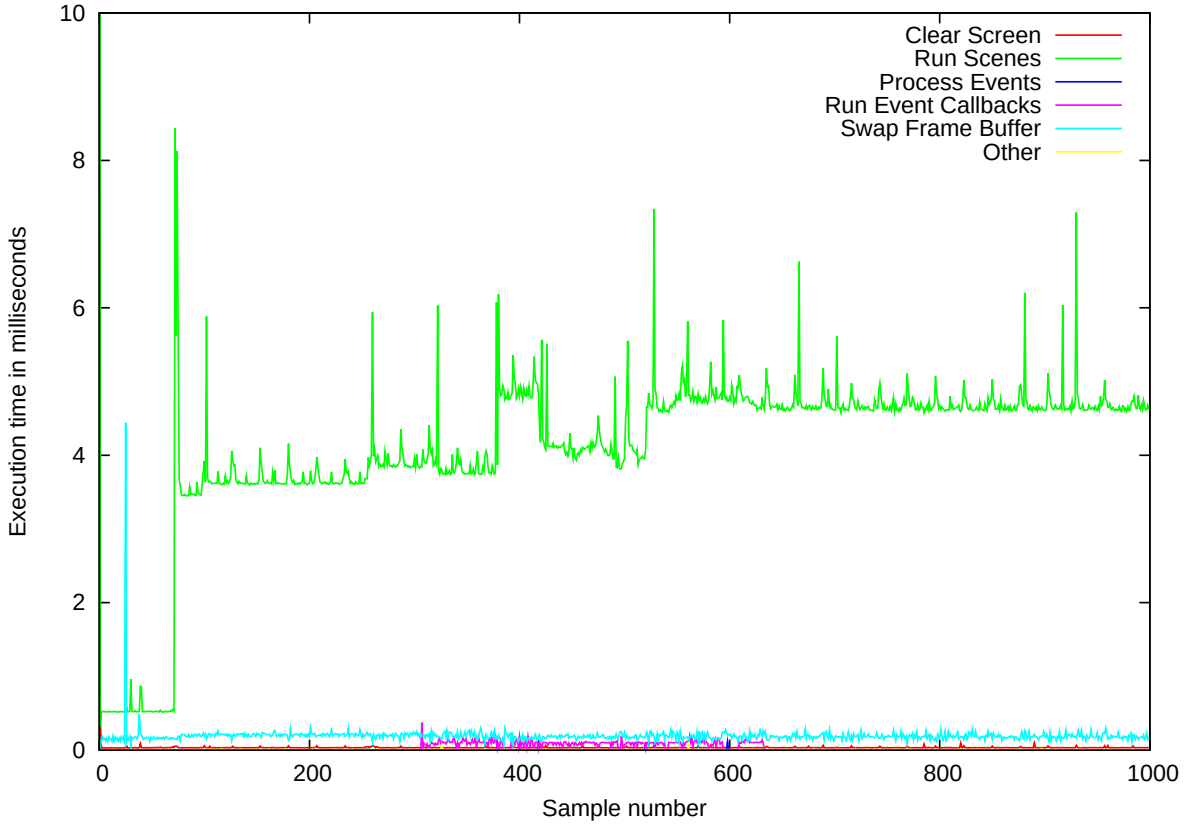


Figure 5.6: Microbenchmark of the visualization engine in WallMon's visualization. During sampling the total number of processes on the cluster were **5190**, resulting in a total number of **20894** entities in WallMon's visualization. The green line labeled "Run Scenes" represents the resource consumption related to scheduling and executing entities.

5.4.3 Case studies

5.4.3.1 Roller coaster

Roller coaster is an application often used as a demonstration application for the Tromsø display wall. This application visualizes a ride along a roller coaster, and does not have support for interactive input. Figure 5.7 shows the finger print of the roller coaster application in WallMon's visualization taken by Wallgrabber, while figure 5.8 shows an image of the application and WallMon taken by a camera. The application, represented as *roller* in the visualization, executes on all 28 tiles, and only the CPU metric of the application is visually present. The squares representing the CPU metric are clustered tightly together at the top-middle of the chart, implying a high ratio of user-level CPU execution and a CPU utilization of about 12 percent (logarithmic scale). Throughout the visualization, the squares moves very little and closely remains within cluster showed in figure 5.7.

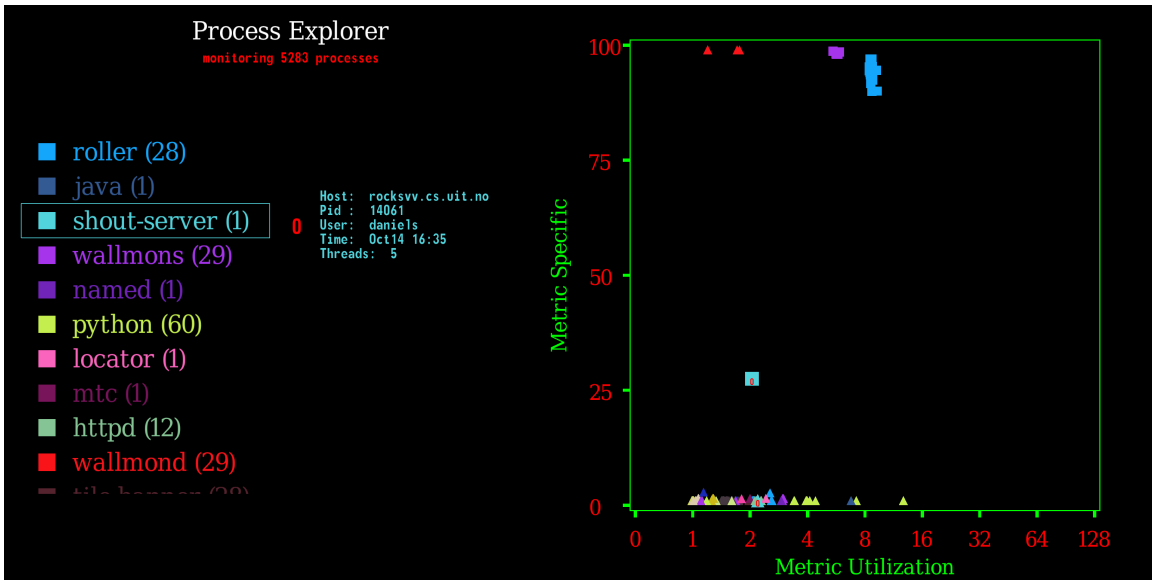


Figure 5.7: WallMon's visualization of the roller coaster application. The processes of the roller coaster application, represented as *roller* are located at the top-middle. The single square towards bottom-left, which is of similar color to the roller coaster squares, is the *shout-server* application.

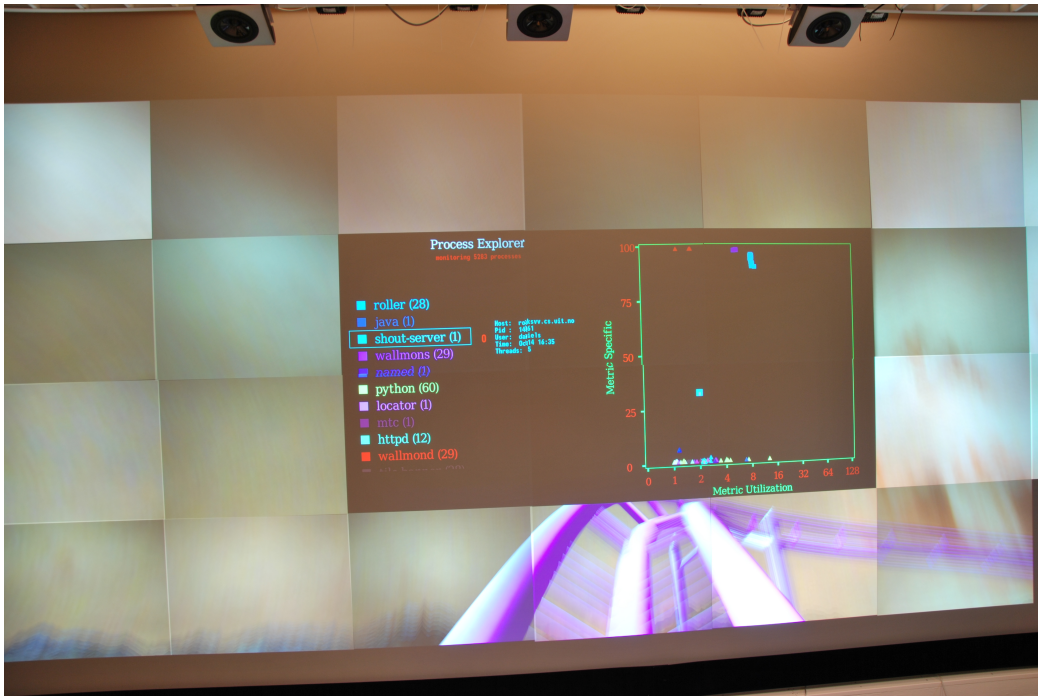


Figure 5.8: The roller coaster application and WallMon running on the Tromsø display wall.

The performance pattern of the roller coaster application indicates that all the processes in the application perform similar actions, and that the application has one thread doing most or all of the work due to the consistent CPU utilization of about 12 percent (100 percent / 8 cores = 12.5 percent utilization per core).

5.4.3.2 Gigapix

The Gigapix application [7] uses all 28 tiles of the display wall to render a 13.3 gigapixel image of Tromsø. The application accepts interactive input for navigating within the image. The original 13.3 gigapixels image [25] consists of 2200 photo images of Tromsø. These 2200 images have been pre-processed and split into hundred thousands of smaller images, which the Gigapix application glues together in real-time based on interactive input from user.

Figure 5.9 shows the Gigapix application during a computational phase where a user is navigating the image, while figure 5.10 shows the Gigapix application during an idle phase where no interactive input have been provided for some time. Both figures are images taken at a desktop computer outside the cluster, and not by the Wallgrabber application. Figure 5.11 shows the data set displayed by Gigapix at the moment the image in figure 5.10 was captured.

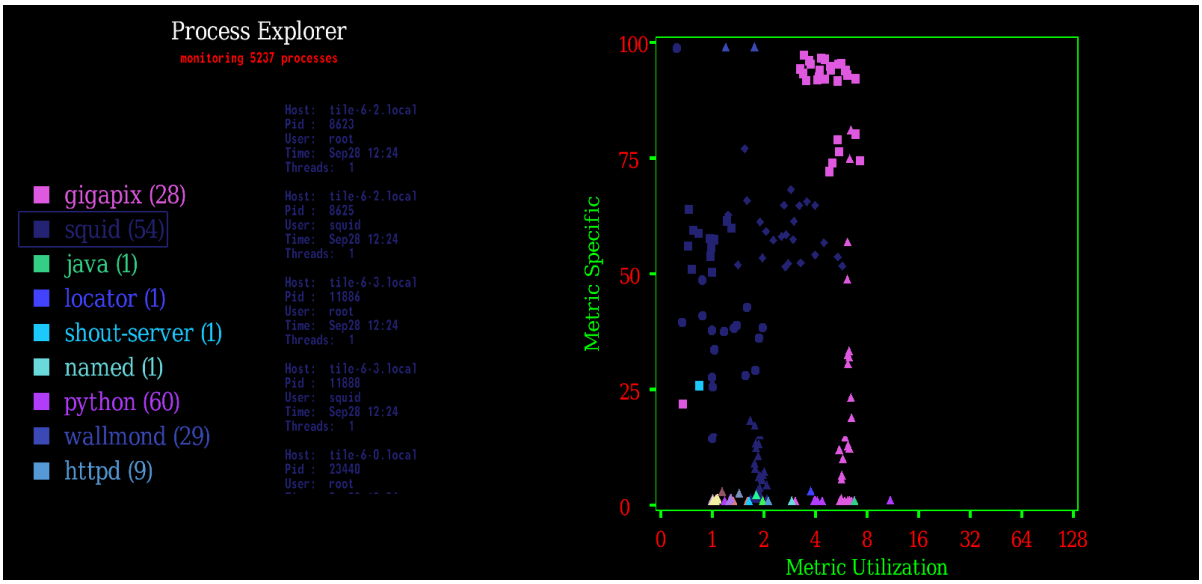


Figure 5.9: WallMon's visualization of the Gigapix application during computational phase. The Gigapix application, *gigapix*, consists of 28 processes where its CPU metrics are located to the top-middle. Gigapix make use of Squid, represent as *squid* and consisting of 54 processes, for fetching images.

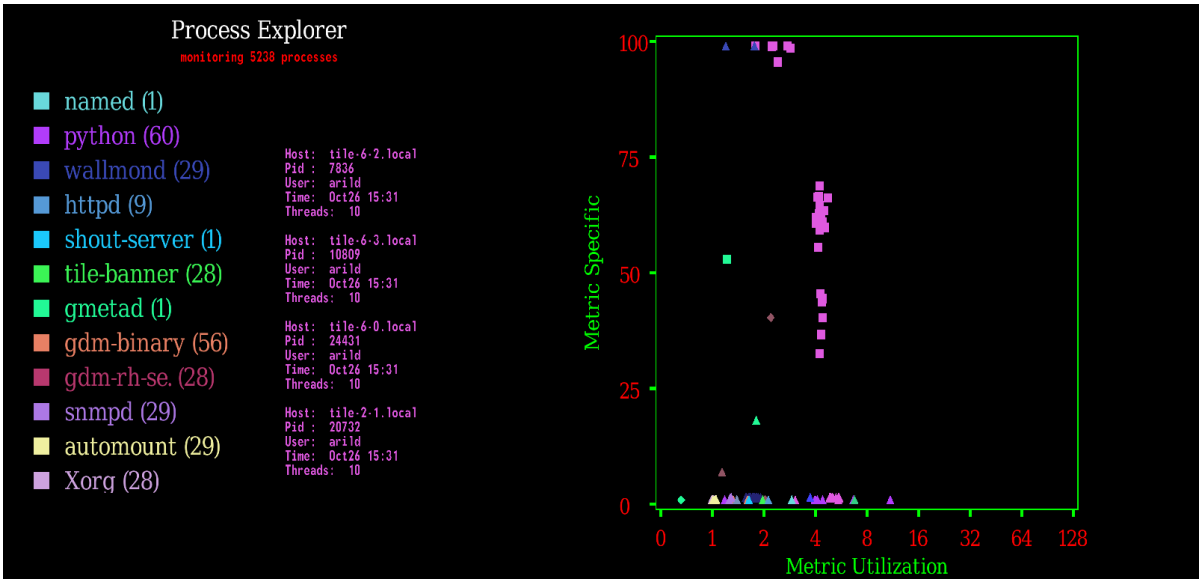


Figure 5.10: WallMon's visualization of the Gigapix application during idle phase. In this phase the CPU metrics are spread from bottom to top in the middle of the chart.

Figure 5.11: Data set displayed by Gigapix during idle phase shown in figure 5.10.

Figure 5.9 and 5.10 reveal several interesting performance patterns of the Gigapix application:

- The idle phase of Gigapix in figure 5.10 shows that the CPU utilization of Gigapix is similar to the CPU utilization in its computational phase as shown in figure 5.9. The reason for this is the rendering engine used to implement Gigapix. This engine renders data at a fixed number of frames per second, making the application consuming about the same amount of CPU.

- In its idle phase in figure 5.10, the CPU metrics of Gigapixs remain scattered from bottom to the top, meaning that the ratio between user- and kernel-level execution time is different among the Gigapix processes. It is unclear why this happens. One reason could be the data set being rendered, which is shown in figure 5.11.

- In its computational phase in figure 5.9, the CPU metrics of Gigapix have a higher ratio of user-level execution time compared to its idle phase in figure 5.10. One reason for this could be the cost of decoding new images fetched from a remote location.

- The computational phase in figure 5.9 shows Squid's performance pattern triggered by Gigapix. Squid is a proxy server, which in Gigapix's context is used to speed up web servers through caching. The caching effect of Squid is present in the visualization: Squid has slightly more outgoing network traffic than incoming, indicating that data requests from Gigapix are cached by Squid. When a request is cached, Squid does not need to retrieve data from the web server. This indication is supported by Squid's slightly more read bandwidth from storage than write bandwidth, indicating that Squid uses a persistent storage cache that is frequently accessed.

5.4.3.3 Weather Research & Forecasting Model

The Weather Research & Forecasting (WRF) Model [8] is a model for calculating and forecasting weather in parallel environments. The implementation of the model evaluated in this thesis is based on MPI, and for exploiting multiple cores, the implementation simply runs multiple processes per node (no shared memory optimization). The overall performance

pattern of the implementation is similar to traditional high performance systems: a master continuously hands out tasks to workers, who do computations and report back results. Figure 5.12 shows the application during the computational phase where workers execute tasks handed out by the master, while figure 5.13 shows the application during the idle phase where works wait for the master to aggregate and commit results from the workers. Both figures are images taken at a desktop computer outside the cluster, and not by the Wallgrabber application.

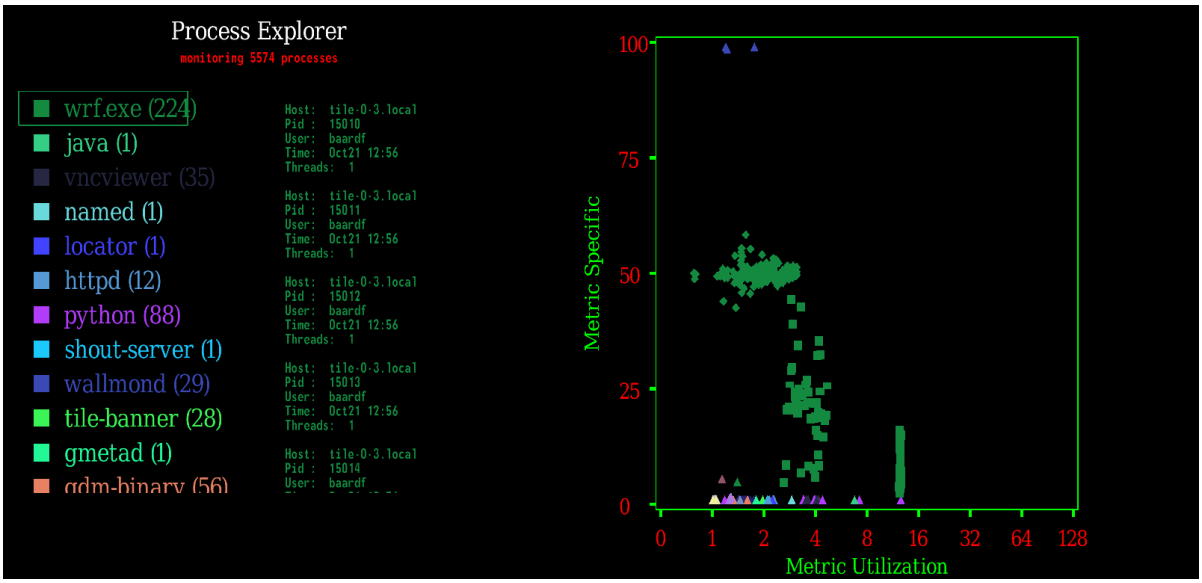


Figure 5.12: WallMon's visualization of The Weather Research & Forecasting (WRF) Model during computational phase. The application, *wrf.exe*, consists of 224 processes where the network metrics are clustered in the left-middle, while the CPU metrics are grouped together in two clusters: one slightly to the left and scattered from the bottom to the middle, while the other slightly to the right at the bottom.

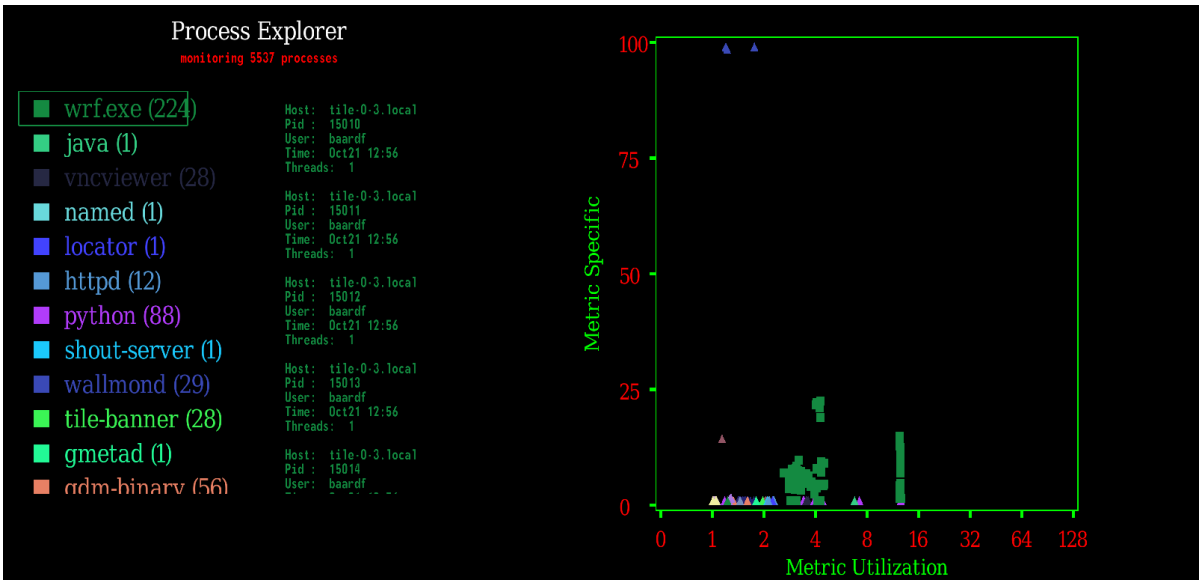


Figure 5.13: WallMon's visualization of The Weather Research & Forecasting (WRF) Model during idle phase. The application, *wrf.exe*, consists of 224 processes. Only the symbols for CPU metric is present, and they are grouped together in two clusters: one slightly to the left and at the bottom, while the other slightly to the right at the bottom.

Figure 5.12 and 5.13 reveal several interesting performance patterns of the WRF application:

- During the computational phase in figure 5.12, the symbols for the CPU metric are grouped into two distinct clusters. This clustering indicates that the workers within the application are not equal, which could be due to hierarchical structuring of workers.
- In the cluster of squares to the left in figure 5.12, the squares consistently moves up to the middle before moving down to the bottom. This pattern indicates the common life-cycle of a worker: the worker receives a task over the network (kernel-level execution), performs computations on the task (user-level execution), and sends the result back to the master (kernel-level execution). The rapid bouncing up and down also indicates a task size that requires few computations, which suggests that a larger task size could increase performance. This indication is also supported by the CPU utilization of these processes; only 4 percent CPU utilization implies a 1/3 saturation of a CPU core.
- In the cluster of squares to the right in figure 5.12, the squares are almost statically (no movement) located in a straight line at the bottom, slightly to the right. The lack of movement indicates that these processes do not execute any tasks from the master, but are focused on something else. They could be responsible for aggregating results from other workers in a hierarchical structure, which is a known pattern for scaling large parallel applications.
- The cluster of diamonds to the middle-left in figure 5.12 shows a consistent amount of network traffic: the symbols steadily moves within the utilization range of 1 to 4 percent. This pattern also indicates the common life-cycle of a worker: workers receive data, do computations (no bandwidth), and send back result. An interesting

observation is that the ratio between the amount of received and sent data is about equal, indicating that the task size is equal to the size of the result.

- Figure 5.13 represents the idle phase of the application. In this phase the master is aggregating and committing results. There are no network traffic present, however, the workers consumes about the same amount of CPU cycles as they did during the computational phase (figure 5.12), as if they where carrying out network communication in a similar manner as in the computational phase. This patterns indicates some sort of busy-waiting, for example in the sense of non-blocking calls to the network layer for receiving data.

5.5 Discussion

5.5.1 Microbenchmarks

The microbenchmarks presented in section 5.4.2 show that scheduling and executing entities is significantly the most expensive part of the visualization engine. Reasons for this include that each performance metric for each process is represented as a separate entity that require execution time. Moreover, these entities perform OpenGl calls, which has some cost. As the figure shows, the graph for running scenes starts out at a cost less that a millisecond, then spikes to above eight milliseconds, before stabilizing on four-five milliseconds. This pattern occurs since, in the beginning of the benchmark, the processes on the cluster have not yet been registered on the severer. The spike occurs when all the processes discovered by the client are registered at the server.

The second most expensive part in this benchmark is swapping the frame buffer. Running event callbacks also have some cost, at which can be seen from around sample number 300 to 600. In this interval a user was heavily interacting with the visualization.

5.5.2 Case studies

Evaluating data visualization systems is a challenge [26]. Evaluations of data visualization systems can take place in laboratories, however, demonstrations in real settings are often necessary for the system to be convincing. The following criteria might be used for evaluating a visualization system:

1. **Usefulness.** To which degree does the system address the domain it targets. Do users benefit from it?

2. **Usability.** How easily do users interact with the system? Are the information provided in clear and understandable format? How fast do users learn to use the system?

3. **Discovery.** To which degree does the system aid users in discovering new patterns in the data, and answer questions the users did not know they had in advance. This criteria can be important since user often do not know what they are expecting and looking for.

WallMon's initial goal was to provide a macro-view of process-level resource usage on clusters. Taking this goal into account, WallMon shows usefulness through the case studies. For example, in the Gigapix application in section 5.4.3.2 users can at a macro-level observe the performance relationship between the Gigapix application and the Squid application.

For usability, the visualization comes with advantages and disadvantages. Experience shows that users do not easily understand the visualization. Some users find all the moving metric symbols confusing and overwhelming. On the other hand, the visualization is centered around only a single chart that, to a large degree, presents different metric symbols in a consistent way. For example, utilization is always represented along the horizontal axis. Once this chart is understood, users have learned most of the system. Chapter 6 elaborates on the evaluation of the interactive mechanisms in the visualization.

When it comes to discovery, WallMon will aid users in different ways. Experiences show that WallMon provides discovery if users have no idea what they are looking for, and are only interested in getting an overall performance picture of a cluster. In such a scenario, users quickly learn which processes account for performance usage, and which performance metrics are significant. On the other hand, experiences show that WallMon lacks discovery when going beyond such a macro-level view. Although the lists in the visualization sometimes provide useful process-level details, experiences show that the amount of discovery provided is lacking.

5.5.3 Visualization features

The current implementation of WallMon's visualization provides a simple graphical representation. Several of these features have potential for improvement. Moreover, several other graphical features have been discussed during the development of WallMon, but not implemented.

Existing features

The case studies show that WallMon's visualization provides a useful way of observing and discovering performance patterns among processes in distributed systems. However, several limitations were experienced. One such limitation occurs when a system consists of many processes that show a tight clustering effect, such as the WRF application in section 5.4.3.3, it becomes difficult to distinguish between and count the number of the different processes within the cluster. One way to address this issue could have been to apply collision detection to metric symbols in order to enforce non-overlapping among symbols, or only partial overlapping.

Another limitation is the details exposed about the different processes through lists, where it would be useful to have access to even more details. For example, several interesting performance patterns were discovered in the WRF application, however, they remained indications due to the lack of process specific information. On the other hand, exposing process-level data at fine-granularity, such as stack traces, is not the goal of WallMon. Profilers can be

used for such analyses.

A third feature that could have been improved is the tail on metric symbols that represents history (previous values). When the tail of the symbols reaches a certain size combined with the presence of many symbols, the visualization quickly becomes overwhelming. An improvement could have been to apply a fading effect on the tails. With this approach, the part of the tail showing the most recent history would have been sharp looking, while the part showing the oldest history would have barely been visible. This approach would probably reduced the amount of graphical output, and it would look more natural.

Future features

One graphical feature that would have been interesting to explore is including another dimension in the visualization chart. For example, this dimension could represent the performance history of a metric, either the history for the metric's entire life-cycle, or for a recent time period. Moreover, this extra dimension could solve the problems related to overlapping symbols and the tail of the symbols. On the other hand, some users already find the visualization complex. A third dimension would most likely increase the complexity.

Another interesting feature, described in section 6.1 in the interactivity chapter, is a vision for more generic lists. For example, in addition to the ability to group processes on process names, it would be interesting to group processes on hostname, or the operating system user processes execute in the context of. Such a feature would probably enhance WallMon's ability to provide discovery; the system would be more able to aid users in discovering new patterns in the data, and answer questions the users did not know they had in advance.

5.6 Conclusion and future work

This chapter presented the visualization part of the WallMon system. The visualization is inspired by information flocking [3] and is implemented on top of underlying infrastructure provided by WallMon. The performance benchmarks carried out on the implementation of the visualization show acceptable overhead, while the case studies carried out show that the visualization is useful; it achieves WallMon's goal to provide an efficient macro-level performance view of clusters. However, some users find it to be complex and difficult to understand.

Future work for the visualization should focus on the visual presentation, and not improving performance. Improvements to existing features include addressing overlapping metric symbols and providing more process-level details. Future features includes investigating the benefits of adding another dimension to WallMon's visualization and providing more generic lists.

Chapter 6

Interactivity

The thesis naturally expanded into exploring interactivity due to the presence of a display wall with support for interactive user input. However, as this chapter describes, interactivity is useful in the domain of cluster monitoring.

6.1 Idea

6.1.1 Motivation

The primary goal of WallMon's visualization is to provide an overall process-level performance view of a cluster. Such a view should be available at start-up of the visualization, and not require any configuration by the user. This type of view might be sufficient in most cases, however, there are scenarios where it would be useful to interactively explore the data further, such as increasing the data granularity.

6.1.2 Approach

Lists form the visual building blocks for interactivity in WallMon. A factor that influenced this choice, was the accuracy of the event system provided by the Tromsø display wall. Both accuracy in position, and discovery and detection of events are limited compared to for example a desktop mouse or touch-enabled smartphones. The idea was that the physical gestures needed for navigating lists would be manageable to implement. Moreover, lists are ubiquitous, meaning that most users intuitively know how to use and navigate within them.

Figure 6.1 shows the initial idea towards browsing and exploring process-level data in WallMon. In the leftmost list in the figure, processes with equal process name have been grouped together. The idea behind this approach is that processes in distributed systems often have similar process names, making it easy to isolate and explore details about these processes. On the other hand, it might be interesting to group processes together based on other criteria, such as hostnames as show in the rightmost list of figure 6.1, or the operating system user the processes execute in the context of. The list in the middle of figure 6.1 holds specific processes together with some detailed data about each of them.

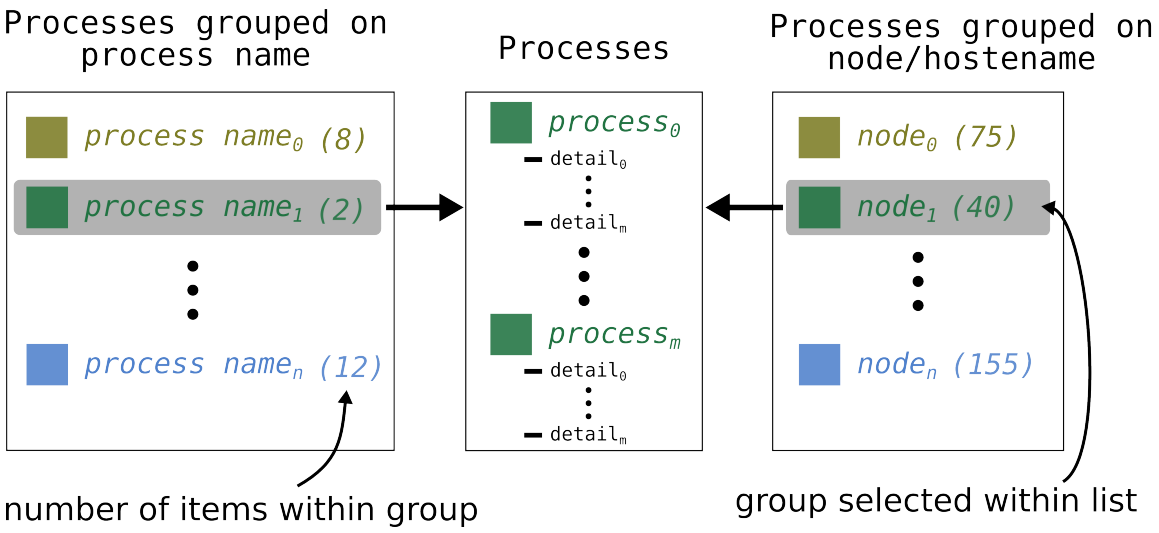


Figure 6.1: Idea for using lists to interactively browse and explore process-level data

Since there can be tens of thousands of processes present on a cluster, it will be necessary to navigate within the lists in figure 6.1. To begin with, the idea was to have a scrolling mechanism combined with a meachanism for selecting items within lists.

6.2 Architecture

Figure 6.2 shows the architecture for how interactivity is provided in WallMon. First of all, an *event engine* in WallMon accepts interactive input from two sources: Shout, which is the event system for the Tromsø display wall, and events from local desktop mice. The event engine parses the input, locates the entities that the events map to, and forwards its findings to the visualization engine, which is described in chapter 5. The visualization engine does some processing of events and forwards the events to entities. With this approach, much of the event processing is unified in one place, while the arbitrary many entities in the system are only concerned with responding to events.

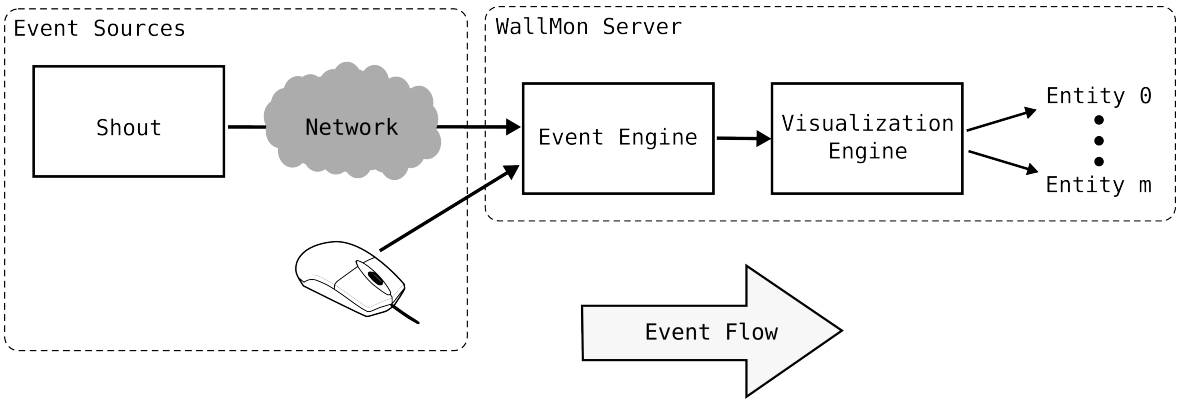


Figure 6.2: Architecture for providing interactivity in WallMon.

6.3 Design and implementation

The interactivity mechanisms in WallMon are part of WallMon's visualization and are hence implemented in C++ and run on Linux.

6.3.1 Event processing and interfaces

The design and implementation of event management in WallMon is exposed in figure 6.3. The figure shows the three most important tasks of the event engine:

- Event parsing includes decoding and extracting necessary information from the event.
- Coordinate translation happens in two stages; the first stage translates the coordinates to WallMon's global coordinate system, while the second stage translates coordinates to a scene's coordinate system. For example, Shout provides coordinates that matches the entire display wall, that is, coordinates within 0-7168 horizontally and 0-3072 vertically. Since WallMon might be running on a handful of the display wall tiles, the coordinate system will be different. The second stage is related to the scene abstraction which provides virtual coordinate systems; WallMon's global coordinates must be translated to match a given virtual coordinate system.
- Event filtering depends on coordinate translation; the translated coordinates will be used to check if an event "hits" any entities. If there are no hits, the event is discarded, otherwise it is put in a queue shared with the visualization engine.

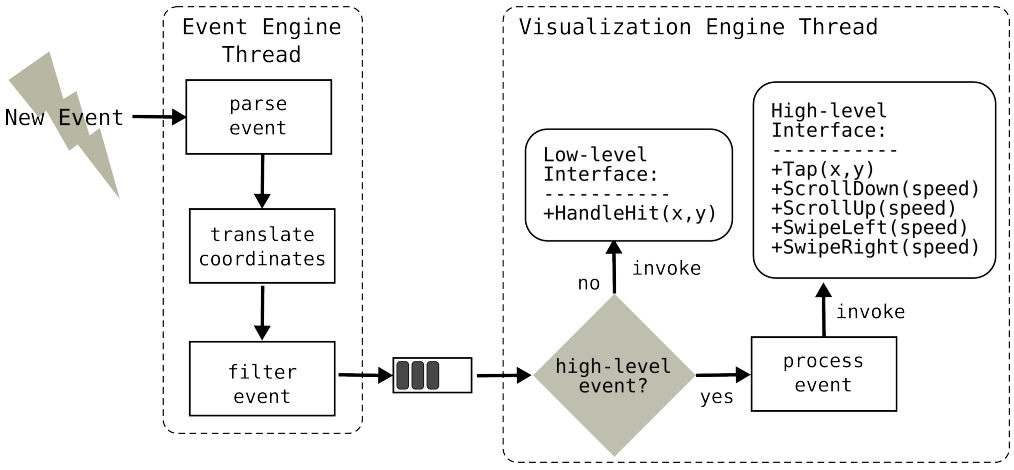


Figure 6.3: Design and implementation of event management in WallMon.

At the time the visualization engine receives events via a queue shared with the event engine, the events have been associated with entities. The reason for not having the event engine to continue and finish the process of event management, is to provide a guarantee of sequential execution within entities; if the event engine were to carry out the callbacks shown in figure 6.3 they could overlap with the callbacks the visualization engine already is carrying out towards entities, making construction of entities more complex.

The visualization engine distinguishes between two types of entities. Each type supports its own interface for accepting events, as shown in figure 6.3. In the *low-level interface*, an entity handles the event itself, while in the *interface-level interface* the visualization engine processes and book-keeps data about previously received events in order to provide a higher level interface, such as callbacks for scrolling and swiping gestures. The higher level interface is the foundation for the interactivity provided by lists in WallMon.

6.3.2 State synchronization

Problem statement

Event management in WallMon started out using a best-effort approach; each WallMon server would run its event management mechanisms completely independently of other servers. This worked sufficiently to begin with. For example, an early interface for WallMon used buttons for navigation and control, where no synchronization problems where observed when running on the display wall. However, when the interface based on lists were introduced, several problems were discovered. These problems are present in figure 6.4, which is an early version of WallMon using lists. The figure shows two lists spanning four tiles. In the leftmost list spanning two tiles, it is clear that the two tiles are not synchronized; in the middle of the list one can observe an item not properly rendered. In the rightmost list, also spanning two tiles, there is an even bigger problem present: the tiles do not render the same process group from the leftmost list. This synchronization problem occurred when an item was selected in the leftmost list; the two tiles do not believe that the same item was selected.

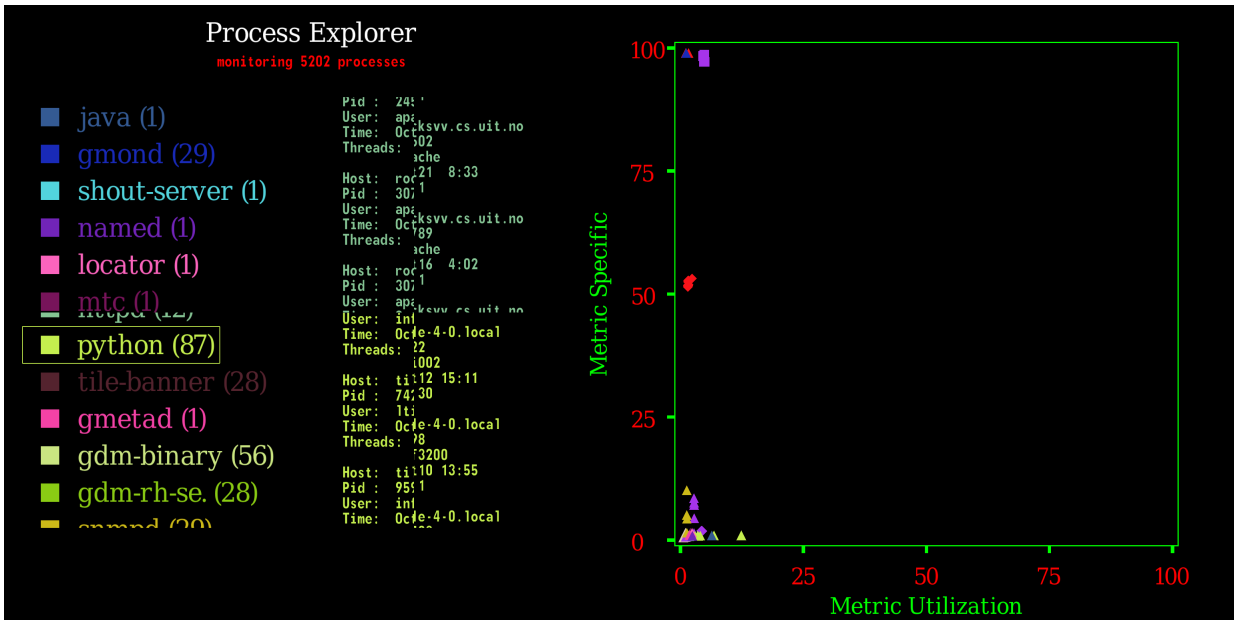


Figure 6.4: Example of state synchronization challenges in WallMon.

Approach

Two approach were considered to address the synchronization issue. The first approach attempts to integrate synchronization into the event processing carried out by the visualization engine, making synchronization transparent to entities. On the other hand, the other approach attempts to implement synchronization within entities. This approach has the drawback that different entities is likely to each implement its own synchronization scheme. However, despite its drawback, this approach was settled upon and implemented due to its simplicity. Figure 6.5 shows the design and implementation of the approach.

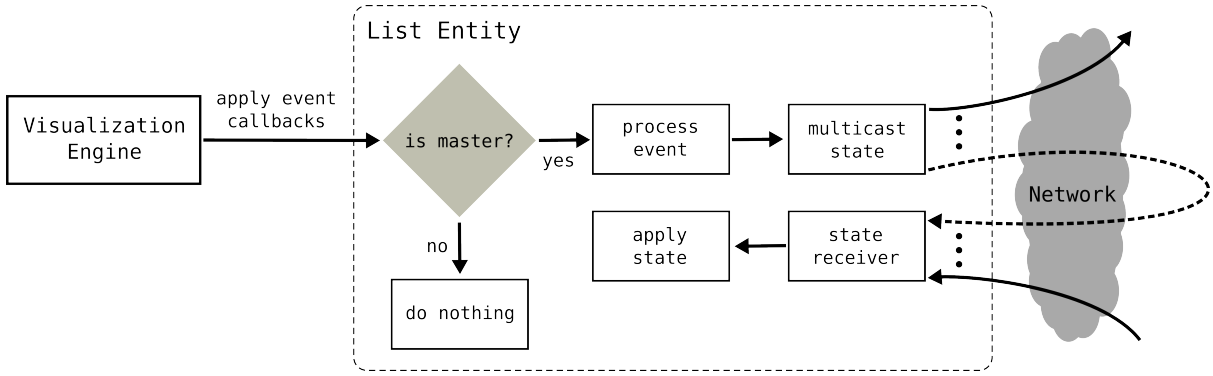


Figure 6.5: Design and implementation of approach for state synchronization.

As figure 6.5 shows, the visualization engine applies event callbacks to a *list entity*, which is the entity implementing and visually rendering lists in WallMon. Among the list entities across servers, one list entity is elected to serve as a master. For all other list entities there will be no actions on event callbacks. However, the master will process the event on an event callback, and multicast the resulting state to all other list entities. Although no performance benchmark has been carried out on the implementation of WallMon that uses synchronization, users could perceive a slight increase in response time after the synchronization was added.

6.3.3 List navigation policies

Navigation within lists are based on the high-level event interface described in figure 6.3: `Tap()`, `ScrollDown()`, `ScrollUp()`, `SwipeLeft()`, and `SwipeRight()`. Scrolling up and down mean a gesture where an object, normally an arm, is moved downwards and upwards, respectively. Swiping to the left and right mean a gesture where an object is moved from right to left for the left swipe, and the other way around for the right swipe. The tapping event is based on radius, and is intended to be triggered with a flat palm, as opposed to the scrolling and swiping events that should be used with a vertically aligned palm.

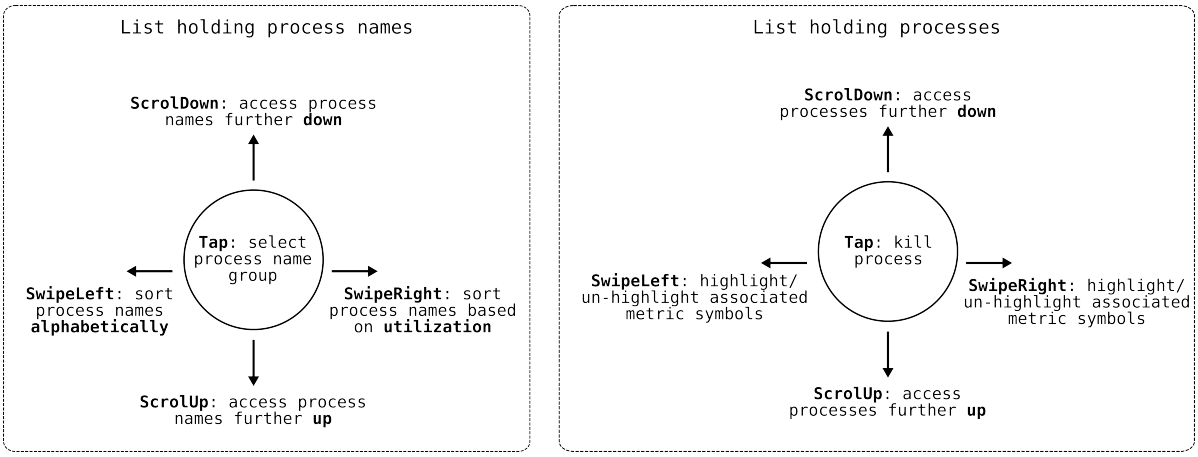


Figure 6.6: Navigation policies of lists in WallMon's visualization.

Figure 6.6 summarizes the actions of the aforementioned gestures in WallMon's visualization. As the figure shows, a tap event in the list holding processes will physically kill the process targeted by the tap. This feature is implemented by `ssh`, and can be useful for testing and exploring fault tolerance in distributed systems. A swipe right event in the list holding process names causes the process names to be sorted on utilization. By having process name groups sorted on utilization, users can easily locate the process groups which graphically display utilization. Listing 6.1 demonstrates pseudocode for relevance ranking of process name groups, which is used during sorting based on utilization. As the listing shows, the calculation of relevance for a process group begins by summarizing the utilization for a process name group. During the summation, the different metrics are weighted differently: network and storage are weighted more than CPU, which is weighted more than memory. After the summation, the average utilization of the process group is calculated. Lastly, large process groups are slightly favored by taking their number of members into account. The background for this policy to calculating relevance, is trail and error; the current policy seems to work fine in practise.

```
def calculate_score(process_group):
    score_sum = 0
    foreach p in process_group:
        score_sum += p.cpu_utilization + p.network_utilization * 1.5
                p.storage_utilization * 1.5 + p.memory_utilization * 0.5
    score_avg = score_sum / process_group.size()
    score_final = score_avg + score_avg * 0.02 * process_group.size()
    return score_final
```

Listing 6.1: Pseudocode demonstrating approach for relevance ranking of process name groups. The final score of a process group is calculated by combining the average utilization of processes within a group and the total number of processes in a group.

6.4 Conclusion and future work

This chapter presented the interactivity part of the WallMon system. Interactivity in WallMon supports Shout, an event system related to the Tromsø display wall, and desktop mice, and it is integrated into WallMons visualization, which is presented in chapter 5. The implementation of interactivity in WallMon provides high-level interfaces for reducing complexity of event management, and carries out distributed state synchronization where necessary. User experience suggests that navigating and using the interactive mechanisms in WallMon's visualization is intuitive, however, the mechanisms are sensitive and unintended actions easily occur.

Future work for interactivity in WallMon includes investigating the possibility for providing a transparent approach to distributed state synchronization among entities. The current approach does state synchronization within specific entities, reducing code reuseability and imposing complexity on entities. Another area to investigate is event processing in order to reduce the sensitivity of gestures, and hence improving user experience.

Chapter 7

Discussion

This chapter discusses the contributions presented in the thesis and the lessons learned during development of WallMon.

7.1 Collector-handler model

The collector-handler model provides an abstraction for gathering and taking action upon gathered data; collectors gather data of interest and handlers take action upon gathered data, while external mechanisms glue together related collectors and handlers. A characteristic of the model is that it adheres to the end-to-end argument [6]; only the collectors and handlers have knowledge of the semantic meaning of gathered data, while the external mechanisms gluing together collectors and handlers are only concerned with efficient data aggregation. This characteristic turned out to be useful in the diverse field of cluster monitoring, however, it is inherently low-level since the user must implement all functionality except data aggregation. Throughout the thesis this aspect was partially addressed on the collector side by implementing reusable libraries for data collection.

Another characteristic of the collector-handler model is its flexibility; different collectors and handlers can be connected in arbitrary ways, as shown in figure 4.4. The initial motivation for this flexibility was to have WallMon run in different environments, such as entirely on a cluster, entirely on a desktop computer, or partially on a cluster and partially on a desktop computer, as seamlessly as possible. The development of the WallMon system documents that this flexibility was to some extent achieved seamlessly, indicating that the collector-handler model provided sufficient flexibility. One additional feature that was considered and that might have provided more flexibility, is a naming registry/server. In a naming registry collectors and handlers could have registered their current address in the form of a hostname or ip-address, removing the need for hard-coding any addresses but the naming registry.

A limitation of WallMon's implementation of the collector-handler model is the one-way communication between collectors and handlers; handlers have no way of sending commands to collectors, except using an ad-hoc approach. This limitation did not turn out to be significant in the development of WallMon. However, there was one particular scenario where such

a feature would have been useful: by having the handler instruct the collector(s) to adjust their sampling rate, users could dynamically adjust the sampling rate through WallMon's visualization and observe the effects.

7.2 Information flocking model for cluster monitoring

The information flocking model is a visualization technique inspired by [3]. Two components make up the model: a chart with moving entities representing different performance metrics and lists which allows for hierarchical exploration of data. A characteristic of this model is generality; the model is not limited to cluster monitoring, and it is straightforward to extend the model by adding additional graphical symbols representing some metrics. One can also speculate that the model would be applicable to metrics in other fields, such as stock prices in stock markets. The model's ability to aid users in discovering patterns might also be useful if applied to stock markets.

Another characteristic of the information flocking models is the macro-level view it provides. Due to the large amount of data collected in cluster monitoring, a macro-level view is enforced since each of the visualization's metric symbols can only visualize a limited amount of data. The primary effect of this is better means to spot and discover performance patterns compared to traditional models, such as traditional graphs/charts and lists.

The aforementioned characteristics of the information flocking model are partially demonstrated through several case studies and its use in the distributed systems course at the Computer Science department at University of Tromsø. The primary lesson learned is that the implementation of the model quickly provides users with a macro-level view of process-level performance behavior of a cluster, aiding the users in observing performance patterns that otherwise would be difficult to discover. However, the implementation of the model lacks details in its visualization. For example, when a performance pattern is observed, or partially observed, it is hard to further explore it. Moreover, some users find the visualization complex and hard to follow.

7.3 Cluster monitoring architecture

The architecture for cluster monitoring developed in this thesis provides real-time aggregation of data, and extensibility through a module system. Real-time data aggregation is based on push-based data transmission, and extensibility allows for integration of the collector-handler model. Although the internals of WallMon is only concerned with aggregating data between collectors and handlers, this results in a simple system that is only concerned with one thing. Moreover, WallMon's implementation of the architecture only supports a one-way communication between collectors and handlers. This further simplified the system.

Chapter 8

Conclusion

This thesis presented WallMon, a tool for interactive visual exploration of performance behaviors in distributed systems. The WallMon system captures and visualizes data for every process on every node, as well as overall node statistics. The development of WallMon has resulted in following contributions:

- The collector-handler model provides an abstraction for gathering and taking action upon gathered data; collectors gather data of interest and handlers take action upon gathered data, while external mechanisms glue together related collectors and handlers.
- The information flocking model is a visualization technique inspired by [3]. Two components make up the model: a chart with moving entities representing different performance metrics and lists which allows for hierarchical exploration of data.
- An architecture for cluster monitoring that provides real-time aggregation of data, and extensibility through a module system. Real-time data aggregation is based on push-based data transmission, and extensibility allows for integration of the collector-handler model.

For gathering of data, WallMon is centered around the collector-handler model; collectors gathers data of interest, such as CPU and memory usage, and forwards it to handlers in a push-based fashion, while handlers take action upon the data. At the architectural level, collectors and handlers are glued together and managed by a light-weight runtime that aggregates data using a push-based approach. WallMon's design is based on the client-server model, and it is extensible through a module system that encapsulates functionality specific to monitoring (collectors) and visualization (handlers). Microbenchmarks show that the gathering of data in WallMon has some cost. Reading data from the procfs virual file system is the most expensive part in gathering of data. It is also the Macrobenchmarks show that reading data from procfs is also the most expensive part when running the WallMon system as a whole.

WallMon visualizes process-level data with an approach inspired by the concept of information flocking [3], originally introduced by Proctor & Winter: "*[...] Information flocking presents data in a form which is particularly suited to the human brain's evolved ability to process information. Human beings are very good at seeing patterns in colour and motion,*

and Information Flocking is a way of leveraging this ability". Flocking behavior is applied to processes in WallMon, and the idea is that distributed systems, which consists of related processes, show, or are expected to show flocking behavior. Case studies show that WallMon's visualization is able to provide a macro-level performance view of processes and distributed systems on clusters of computers. For example, when exploring data sets in a particular distributed system [7], WallMon's visualization revealed that certain visually similar data sets caused some of the participating processes to execute more on kernel-level (and less on user-level). However, some users find the visualization to be complex and overwhelming.

WallMon's visualization is interactive and allows users to explore process-level data. On the Tromsø display wall, interactivity is supported via touch-free gestures, while desktop mice provide interactivity on desktop computers. Interactivity in WallMon assumes a distributed context and carries out distributed state synchronization where necessary. User experience suggests that navigating and using the interactive mechanisms in WallMon's visualization is intuitive, however, the mechanisms are sensitive and unintended actions/gestures easily occur.

Chapter 9

Future Work

This chapter outlines some of the possible future directions that can be researched based on the current status of the work presented in this thesis.

WallMon's information flocking model for cluster monitoring have several directions for continued research. It would be interesting to further explore the model in both the context of cluster monitoring, and in other fields. Within the field of cluster monitoring, the visualization can be extended with additional process-level performance metrics, such as amount of system calls and cache hit ratios. Other fields that the information flocking model might be applied to include metrics in stock markets, such as stock prices. The model's ability to aid users in discovering patterns might be useful if applied to stock markets.

Another research direction for WallMon's information flocking model is to further investigate if the visualization can be used to spot known performance challenges and problems, such as deadlocks, livelocks and inefficient task distribution in high performance systems, by looking for specific unified patterns in the visualization. For example, a deadlock could be recognized as a sudden unexpected drop in CPU utilization among several or all participating processes. Additional further research directions for WallMon's information flocking model includes exploring features such as providing a third dimension of data to the visualization, and providing more generic mechanisms for exploring data.

Future work for WallMon's infrastructure include adding support for exploiting multiple cores when executing collectors and handlers. This functionality is currently single-threaded. Moreover, it should be investigated to which degree a synchronization scheme between collectors would be beneficial. In the current approach, there is no synchronization between collectors, which implies that collectors could drift over time, making them sample data at significantly different points in time. However, for certain application domains, such as visualization, the current approach might be sufficient.

Future work for interactivity in WallMon includes investigating the possibility for providing a transparent approach to distributed state synchronization. The current approach takes an ad-hoc approach to state synchronization, reducing code reuseability and imposing complexity of event management. Another area to investigate is event processing in order to

reduce the sensitivity of gestures, and hence improving user experience.

Bibliography

[1] M. Massie, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2004.04.001

[2] "HP Cluster Management Utility," http://h20311.www2.hp.com/HPC/cache/412128-0-0-0-121.html and documentation therein.

[3] G. Proctor and C. Winter, "Information Flocking: Data Visualisation In Virtual Worlds Using Emergent Behaviours," in *In Virtual Worlds 98*. Springer-Verlag, 1998, pp. 168–176.

[4] D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen, and O. J. Anshus, "Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays," in *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007*, Jun. 2007, pp. 75–83.

[5] T. A. DeFanti, J. Leigh, L. Renambot, B. Jeong, A. Verlo, L. Long, M. Brown, D. J. Sandin, V. Vishwanath, Q. Liu, M. J. Katz, P. Papadopoulos, J. P. Keefe, G. R. Hidley, G. L. Dawe, I. Kaufman, B. Glogowski, K.-U. Doerr, R. Singh, J. Girado, J. P. Schulze, F. Kuester, and L. Smarr, "The OptIPortal, a scalable visualization, storage, and computing interface device for the OptiPuter," *Future Gener. Comput. Syst.*, vol. 25, pp. 114–123, Feb. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1414095.1414287

[6] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984. [Online]. Available: http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf

[7] "The Gigapix application," http://www.cs.uit.no/~daniels/gigapix/.

[8] "The Weather Research and Forecasting Model," http://wrf-model.org/index.php.

[9] "Wallgrabber," http://rocksvv.cs.uit.no:8008/ and documentation therein.

[10] J. M. Brandt, A. C. Gentile, D. J. Hale, and P. P. Pebay, "OVIS: a tool for intelligent, real-time monitoring of computational clusters," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2006.1639698

[11] M. J. Sottile and R. G. Minnich, "Supermon: a high-speed cluster monitoring system," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 39–46. [Online]. Available: http://dx.doi.org/10.1109/CLUSTR.2002.1137727

[12] T. C. Ferreto, C. A. F. de Rose, and L. de Rose, "RVision: An Open and High Configurable Tool for Cluster Monitoring," in *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, May 2002, p. 75. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2002.1017114

[13] K. Ren, J. López, and G. Gibson, "Otus: resource attribution in data-intensive clusters," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1145/1996092.1996094

[14] B. H. McCormick, T. A. DeFanti, and M. D. Brown, "Visualization in Scientific Computing-A Synopsis," *Computer Graphics*, vol. 21, no. 6, pp. 61–70, Nov. 1987. [Online]. Available: http://dx.doi.org/10.1109/MCG.1987.277014

[15] N. Gershon, S. G. Eick, and S. Card, "Information visualization," *interactions*, vol. 5, pp. 9–15, Mar. 1998. [Online]. Available: http://doi.acm.org/10.1145/274430.274432

[16] J. Heer, M. Bostock, and V. Ogievetsky, "A tour through the visualization zoo," *Commun. ACM*, vol. 53, no. 6, pp. 59–67, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1145/1743546.1743567

[17] Hewlett-Packard Company, "HP CMU - A simple, scalable, and flexible Linux Cluster Management Utility," White paper, Hewlett-Packard Company, Mar. 2009. [Online]. Available: http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA2-5035ENW.pdf

[18] D. Chao, "Doom as an Interface for Process Management," in *Proceedings of SIGCHI'01*. ACM Press, 2001, pp. 152–157.

[19] W. Harrop and G. Armitage, "Modifying first person shooter games to perform real time network monitoring and control tasks," in *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2006. [Online]. Available: http://dx.doi.org/10.1145/1230040.1230074

[20] *Time-Varying Data Visualization Using Information Flocking Boids*, 2004. [Online]. Available: http://dx.doi.org/10.1109/INFVIS.2004.65

[21] M. Fowler, "InversionOfControl," 2005. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html

[22] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 137–150. [Online]. Available: http://www.google.com.sg/url?sa=t\&\#38;source=web\&\#38;ct=res\&\#38;cd=1\&\#38;ved=0CAcQFjAA\&\#38;url=http\%3A\%2F\%2Flabs.google.com\%2Fpapers\%2Fmapreduce-osdi04.pdf\&\#38;ei=sAIeS6fDFI3s7APtm93XCw\&\#38;usg=AFQjCNGR2uEfpCUiHvw6I876kTPeiv-mUA\&\#38;sig2=3GFcjJwIVyK18UCN9e3iSA

[23] "InversionOfControl." [Online]. Available: http://martinfowler.com/bliki/

[24] J. M. Brandt, B. J. Debusschere, A. C. Gentile, J. R. Mayo, P. P. Pebay, D. Thompson, and M. H. Wong, "Ovis-2: A robust distributed architecture for scalable RAS," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, Apr. 2008, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2008.4536549

[25] "The Gigapix image of Tromsø," http://gigapix.no/881/gigapiksel/utsikt-over-tromsø.

[26] C. Plaisant, "The challenge of information visualization evaluation," in *AVI '04: Proceedings of the working conference on Advanced visual interfaces*. New York, NY, USA: ACM, 2004, pp. 109–116. [Online]. Available: http://dx.doi.org/10.1145/989863.989880

Appendix A

Published Papers

A.1 *WallMon: interactive distributed monitoring of per-process resource usage on display and compute clusters*

This paper was published in Norsk Informatikk Konferanse (NIK) 2011.

WallMon: interactive distributed monitoring of per-process resource usage on display and compute clusters*

Arild Nilsen
arild.nilsen@uit.no

Daniel Stødle
daniels@cs.uit.no

Tor-Magne Stien Hagen
tormsh@cs.uit.no

Otto J. Anshus
otto@cs.uit.no

Abstract

To achieve low overhead, traditional cluster monitoring systems sample data at low frequencies and with coarse granularity. This makes such systems unsuited for interactive monitoring of distributed cluster applications, as they fail to capture short-duration events, making understanding the performance relationship between processes on the same or different nodes difficult. This paper presents WallMon, a tool for interactive visual exploration of performance behaviors in distributed systems. Data is visualized using a technique inspired by the concept of information flocking. WallMon is client-server based, and captures data for every process on every node. Its architecture is extensible through a module system that encapsulates functionality specific to monitoring and visualization. Preliminary performance results show 5% CPU load at 64 Hz sampling rate when monitoring a cluster of 5 nodes with 300 processes per node. Using WallMon's interactive visualization, we have observed interesting patterns in different parallel and distributed systems, such as unexpected ratio of user- and kernel-level execution among processes in a particular distributed system.

1 Introduction

Traditional cluster monitoring systems, such as Ganglia [1] and HP Cluster Management Utility [2], gather data about resource usage in cluster of computers and presents it visually to users. To achieve low overhead, these systems sample data at low frequencies and with coarse granularity. For example, [1] reports a default sampling frequency of 15 seconds and a data granularity on node level, such as total CPU and memory consumption. However, this makes such systems unsuited for interactive monitoring of distributed cluster applications, as they fail to capture short-duration events and make understanding the performance relationship between processes on the same or different nodes difficult. Interactive monitoring requires frequent sampling of fine-grained data and visualization tools that can explore and display data in near real-time. Suitable sampling frequency for interactive monitoring varies between application domains and applications. In situations where it is important to capture and visualize short-duration events, the sampling rate should

*This student paper is based on research conducted in Arild Nilsen's Master project

This paper was presented at the NIK-2011 conference; see http://www.nik.no/.

match the frame rate of the visualization, which can be as high as 60 frames per second. However, sampling at 60 Hz might capture too much data and make the visualization difficult to follow and understand. Fine-grained data is important when related distributed systems execute simultaneously. Without process level granularity, it would be difficult to observe how behavior in one system impacts behavior of other systems.

WallMon is a system for interactive visual exploration of performance behaviors in distributed systems. The system is client-server based, and captures data for every process on every node, as well as overall node statistics. The architecture of WallMon is modular and extensible: At its core, a light-weight runtime dynamically loads executable modules on the client-side and server-side. The modules encapsulate functionality specific to gathering of data, and the actions taken upon the gathered data. Although WallMon provides process level granularity, it does not, nor does it intend to, provide capabilities of profilers. While profilers are able to gather micro-level data, such as detailed stack traces of processes, WallMon's process level data is at a macro-level, such as total CPU, memory and network I/O consumption of single processes.

The visualization of data gathered by WallMon is based on an approach inspired by the concept of information flocking [3], originally introduced by Proctor & Winter: "*[...] Information flocking presents data in a form which is particularly suited to the human brains evolved ability to process information. Human beings are very good at seeing patterns in colour and motion, and Information Flocking is a way of leveraging this ability*". As figure 1 shows, flocking behavior is applied to processes. The idea behind this concept in WallMon, is that distributed systems, which consists of related processes, show, or are expected to show flocking behavior. Such behavior might also be interesting to observe and explore when multiple distributed systems execute simultaneously.

Figure 1: WallMon running on the Tromsø display wall. WallMon's visualization runs on 3x2 tiles, while an image of Tromsø is rendered in the background. Moving symbols within axes represent different performance metrics of processes running on the display wall cluster.

Preliminary experiments show that process level monitoring in WallMon comes at an acceptable cost. On a 5 node cluster with 300 processes per node, WallMon has a 5% CPU load when sampling at 64 Hz. Its maximum sampling rate on this cluster is about 128 Hz. On this cluster, the primary overhead is caused by obtaining raw process level data from the operating system.

The visualization inspired by information flocking has only been applied to a few distributed systems, however, in the visualization of these systems we have observed interesting patterns. For example, when exploring data sets in a particular dis-

tributed system, certain visually similar data sets caused some of the participating processes to execute more on kernel-level (and less on user-level). This particular example can be seen in figure 1; The majority of quadratically shaped symbols, which represents CPU utilization horizontally and ratio between user- and kernel-level execution vertically, have about the same CPU utilization, however, they are unexpectedly scattered vertically. Compared to traditional visualization approaches, such as using charts and/or graphs, our experience is that WallMon's visualization makes it easier to discover patterns in distributed systems. We are currently experimenting with the visualization technique on a diverse range of distributed systems, and extending and improving WallMon.

2 Related Work

Ganglia [1] and HP Cluster Management Utility [2] are two traditional cluster monitoring systems that are designed for overall cluster management. They obtain low overhead and scalability through data sampling at low frequencies and with coarse granularity. Supermon [4] is a system similar to Ganglia. Both systems employ a pull-based approach for transfer of data, and a hierarchical architecture for scalable aggregation of data. Supermon focuses on and provides frequent sampling of data. It uses a kernel module for Linux in order to avoid context switch overhead when gathering data from the `procfs` file system.

RVision [5] provides a module system similar to WallMon's module system. Both systems allow user-defined modules to be dynamically loaded at runtime. Differences between the module systems include that RVision's is procedure oriented, while WallMon's is object oriented, and that WallMon offers more control over the usage of gathered data.

As with WallMon, Otus [6] also samples data at process level. Otus also have support for sampling data from specific applications, such as Hadoop MapReduce. While WallMon focuses on visualizing data in near real-time, Otus provides detailed post-analyses via charts.

The data visualization in WallMon runs on wall-sized, high-resolution tiled displays. [7] is the current tiled display supported. This tiled display comes with an event system that has been integrated into WallMon's visualization.

3 Architecture and Design

Figure 2 shows the overall architecture of WallMon: The module system and the core runtime which glues together modules. The module system consists of *collectors* and *handlers*. The collector gathers data of interest, and the handler takes action on the gathered data by for instance storing or visualizing it. This organization gives control over functionality specific to monitoring and end-to-end usage of data [8], while leaving data distribution to the core runtime. The core runtime notifies the collector when to collect data, and when data has been collected, the runtime will route the data to the appropriate handler. For instance, figure 2 shows a typical data flow: The core runtime routes data from multiple collectors, each running on a node in a cluster of computers, to a single handler running on a remote workstation.

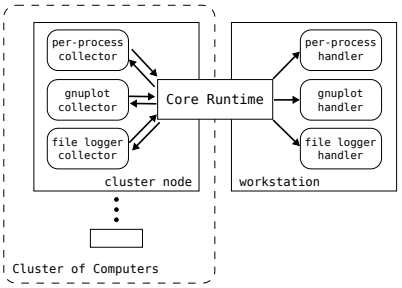


Figure 2: WallMon Architecture and Current Modules Available

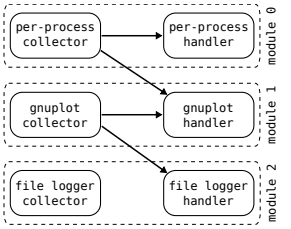


Figure 3: Flexible Mapping of Collectors and Handlers in WallMon

Typically there is a one-to-one mapping between collectors and handlers, as shown in figure 2. For example, the data collected by the gnuplot collector might not be interpretable by the file logger handler. However, the system supports a M:N mapping of collectors and handlers. Figure 3 shows an alternate mapping of the collectors and handlers in WallMon, where the file logger collector is not used, and both remaining collectors each supply two handlers with data.

3.1 Core Runtime

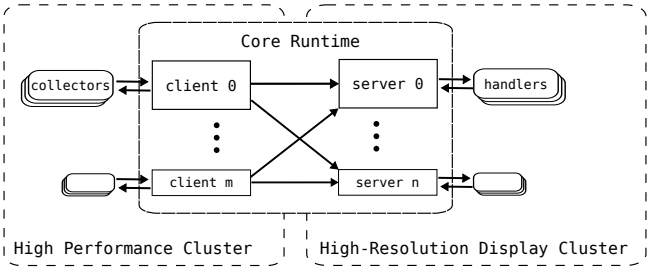


Figure 4: Core Runtime in WallMon

The core runtime in WallMon is based on the client-server model, as shown in figure 4. The core runtime consists of an arbitrary number of clients and servers. Servers are independent of each other and a client might not be aware of all servers. The motivation for clients to be connected to multiple servers, is flexibility. For certain application domains, this model keeps the core runtime general, and places specific functionality in collectors and handlers. Figure 4 shows such a case: To make WallMon run and provide visualization on wall-sized, high-resolution tiled displays, collected data is sent to all nodes/tiles driving the high-resolution display. The client-server model results in a simpler system and lower latency than alternative approaches, such as hierarchical structures employed in [1] and [4]. On the other hand, a hierarchical structure scales to larger number of nodes.

3.2 Module System - Collectors and Handlers

A module consists of a collector that runs on the client-side, and a handler that runs on the server-side. For instance, a typical configuration is one collector per cluster node sending data to a single handler. Collectors and handlers are based on the inversion of control pattern [9]: They implement an interface shared with the core runtime, and do not manage a unit of execution, but wait for the core runtime to invoke them. Collectors are responsible for collecting data at user-defined intervals, which the core runtime will forward to end-points (servers) specified by the collector. Handlers are invoked when data from their associated collectors arrives at the server-side. The action taken upon the data can be anything, and figure 2 shows the handlers (and collectors) currently available in WallMon.

The main motivation for a module system in WallMon is a way of extending the system. This is important since monitoring is diverse, both when it comes to collecting data and taking action upon data. Among others, visualization, applying statistics to monitored data, higher level of abstractions for accessing monitored data, which metrics to monitor and how to monitor them, are ideas and questions that have been considered in WallMon. The module system allows for quick exploration and prototyping of different approaches.

An important guarantee of modules is sequential execution: Collectors and handlers are never invoked concurrently. This guarantee removes the need for synchronization primitives inside collectors and handlers, such as mutexes and monitors. However, there might be necessary to spawn unit of execution within modules. In such scenarios, the user is responsible for handling concurrency.

3.3 Push-Based Data Transfer

In WallMon, clients push monitored data to server(s). This contrasts to many traditional monitoring systems in which servers pull data. For a system that provides near real-time data, a push-based approach has potential to alleviate load on the server-side. Instead of book-keeping when to pull data and carrying out pulling of data, the server is only focused on receiving data. On the other hand, servers have less control over rate and amount of incoming data in a push-based approach. This lack of control could result in clients overwhelming servers.

On the server-side, WallMon uses asynchronous I/O (AIO) for handling multiple persistent connections, one for each client. The primary reason for settling on AIO over an alternative approach where each client-connection is handled in a separate thread, is minimal overhead and simplicity. In AIO there is only a single thread managing all the connections, which eliminates the need for spawning and allocating resources for a new thread whenever a client connects to the server. It is unclear whether AIO or other alternative approaches would contribute the most to scalability in Wallmon.

4 Implementation

WallMon is implemented in C++ and runs on Linux.

4.1 Core Runtime and Module System

Figure 5 exposes internal components of WallMon's core runtime and module system. The `scheduler` and `source` make up the client-side of the core runtime, and together with the collectors they make up the WallMon client, which is implemented as an UNIX daemon. The `scheduler` manages the collectors and puts collected data in a queue shared with the `source`, whose job is to forward data to the WallMon server(s). The server-side of the core runtime is similar to its client-side: a `sink` handles all the incoming connections and puts received data in a shared queue. A `router` takes data out of this queue and invokes appropriate handlers. Each of the shown components of the core runtime is implemented by a single thread. This could be a problem with regard to the `scheduler` and `router`. For example, a handler might block when invoked by the `router`. This would keep the `router` from fetching incoming data and invoking other handlers. A solution for this could be to implement the `router` with a thread-pool. However, this particular scenario has not caused problems in WallMon, and therefore has not been prioritized.

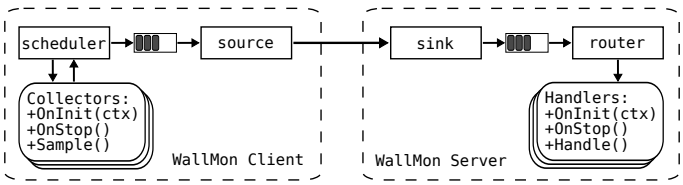


Figure 5: Implementation of Core Runtime and Modules in WallMon

Modules in WallMon are represented as UNIX shared libraries. Figure 5 shows the interface required to be implemented by collectors and handlers. In common they have to provide the `OnInit` and `OnStop` functions that are invoked at the beginning and end of their life-cycle, respectively. while collectors have to provide `Sample` for collecting data, handlers have to provide `Handle` for handling data. The argument passed in `OnInit`, which is shared with the core runtime, is used to control certain aspects of the life-cycle, such as how often a collector is invoked.

4.2 Data Transfer

Asynchronous I/O (AIO) in WallMon is implemented by using the `libev`[1] framework. `libev` is a cross-platform framework that automatically makes use of available AIO primitives on supported platforms. For example, if available, the `epoll` system call is used on Linux, otherwise `select` or `poll` is used as an alternative.

WallMon's network communication protocol relies on Protocol Buffers[2], a serialization format developed by Google. Most of the modules in WallMon also use Protocol Buffers internally. The Protocol Buffers library was chosen due to its simplicity and binary serialization. For example, compared to XML representation, binary serialization in Protocol Buffers is more compact.

WallMon uses TCP as the underlying protocol for transfer of collected data. When

[1] http://software.schmorp.de/pkg/libev.html
[2] http://code.google.com/p/protobuf/

a client connects to a server for the first time, the connection remains open in order to avoid further overhead of handshakes in the TCP protocol. TCP is used since WallMon does not assume that clients and servers are located on the same local area network. Hence, the network between clients and servers in WallMon could be unreliable and the well-known guarantees provided by TCP are important.

4.3 Per-Process Data Collection

In order to gather data about every process on a cluster of computers, WallMon uses the `procfs` file system present in most Linux environments. `procfs` is a virtual file system that, among others, keeps a file for each process on the system. These files are continuously updated with different types of metrics. The collector responsible for gathering per-process data, will open all the process specific files in `procfs` and read them during sampling of data.

Process monitoring of data in WallMon is done at the user-level. This has the advantage of not having to modify the kernel. However, kernel modifications can serve as an optimization, as reported in [4]. In the case of WallMon, where relatively large amount of resources are monitored, a kernel modification could have been justifiable.

Currently there is limited amount of filtering of data on the collector side. Data obtained from the `procfs` file system is parsed and transformed from ASCII to binary representation, such as integers, before being serialized by Protocol Buffers. One optimization could have been to filter out, for example data about processes whose resource consumption is below a certain threshold, inside the collector. On the other hand, the current approach is flexible for handlers, and simple.

5 Visualization Using Information Flocking

Data gathered by WallMon is visualized by an approach inspired by information flocking [3], a technique that leverages humans' ability to see patterns in color and motion. The approach was motivated by WallMon's goal for sampling fine-grained data at near real-time rate. Several prototypes early on in the WallMon project showed that traditional charts and graphs were unsuited for achieving such goals. The primary obstacle experienced with traditional approaches was the difficulty of providing a high-level view of and the relationships between monitored processes. However, traditional charts and graphs might be suitable for visualizing detailed information about a single process, or a handful of processes.

Figures 1 and 6 show the primary components of WallMon's visualization. At the heart of the visualization is a chart with moving entities, where an entity's shape represents a performance metric, such as CPU or memory, and its color represents a process name. The chart's horizontal axis shows the relative resource utilization of the different performance metrics. For example, for the CPU metric the number of CPU cores is taken into account, however, the chart does not say anything about the distribution of execution time among the different cores. The vertical axis represents something specific to the different metrics. For instance, for CPU it shows the distribution of kernel-level and user-lever execution: The more to the bottom, the more kernel-level execution of the total CPU execution time, and vice versa. For

memory, the vertical axis shows a process' current amount of page faults.

Another component of the visualization is an interactive list, where each entry groups together processes with equal process name. When accessing one of these entries using a touch gesture, another list appears. This list holds detail data about each process under this process name, such as how many threads the process is running and on which host it is running. Within this list, the user has the possibility to physically kill any of the processes; A feature that assumes automatic login via ssh. Such a feature can be useful to test and observe performance effects of fault tolerance in distributed systems.

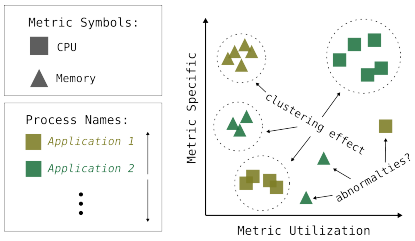


Figure 6: Illustration of information flocking inspired visualization and its user interface

WallMon's visualization provides an useful way of discovering similarities between processes in distributed systems, as shown in figure 1 and 6. If processes are expected to behave similarly, such as slaves in a master-slave model, the visualization quickly reveals whether this is the case or not through a clustering effect, which occur when the measurements of the given performance metrics are similar.

6 Initial Experiments

The experiments have been carried out on a cluster of 5 nodes connected via a full-duplex gigabit Ethernet. The hardware of each node consists of a 2-way quad-core Intel Xeon E5620 CPU at 2.4 GHz with hyper-threading (a total of 16 threads per node) and 24 GB of RAM. Each node runs the Rocks Cluster Distribution with Linux kernel 2.6.18.

6.1 Microbenchmarks

Functionality	Average time	Std dev
Reading from `procfs`	8.74 ms	0.01 ms
Parsing `procfs` data	0.93 ms	0.001 ms
Protocol Buffers data population	0.14 ms	0.0006 ms
Protocol Buffers data serialization	0.03 ms	0.0 ms
Other	0.12 ms	0.0 ms
Total	9.96 ms ($\approx$ 100.4 Hz)	0.0116 ms

Table 1: Microbenchmark of process level sampling in WallMon. One sample includes sampling data about all present processes, which at the time was 318. Samples were carried out consecutively.

The cluster's front-end node, which acts as the entry point for the cluster, executes more operating system processes than the other nodes, which could affect the collector sampling data about processes. At the point the experiments were carried out, the front-end node executed 318 processes, while the other nodes executed about

270 processes each. In this section, process level sampling refers to gathering of data about all present processes. For example, at the front-end, one process level sample gathers data about 318 processes. Gathering data about all present processes might not generally be an ideal approach, however, for benchmarking it is suitable.

Microbenchmarks presented in this section have been carried out on the cluster's front-end node. Table 1 shows the time consumption of the different phases of sampling process level data in WallMon. The results are the average of 1000 consecutive samples carried out as fast as possible. In this particular benchmark, the average sampling rate was 100.4 Hz. On average, this benchmark results in an average sampling rate between 80 to 120 Hz, however, sometimes the sampling rate drops to below 60 Hz.

The data representation of the Protocol Buffers library has also been microbenchmarked. Serializing one sample of process level data at the cluster's front-end results in 13620 bytes, which is an average of 43 bytes per process.

6.2 Macrobenchmarks

Figure 7 shows CPU load at different sampling rates for the client and the server. In both benchmarks, the WallMon client executed on all nodes on the cluster, while the WallMon server executed on the front-end node. During the benchmarks, only one Wallmon module was loaded and executed, the gnuplot module. This module's collector gathers process level data on the client-side, while its handler stores the collected data in memory until all collectors are done, before it uses the data to generate charts (including the ones presented here). For the client benchmark (7a), the values presented are the average of all clients. On the other hand, the value for the server benchmark (7b) originates from the client executing on the same node as the server, which would be the front-end. Moreover, the values were obtained in the same sampling session: Collectors started at one Hz, before steadily doubling the sampling rate up until and including 256 Hz. During the client benchmark, the memory usage remained constant on all clients for all sampling rates.

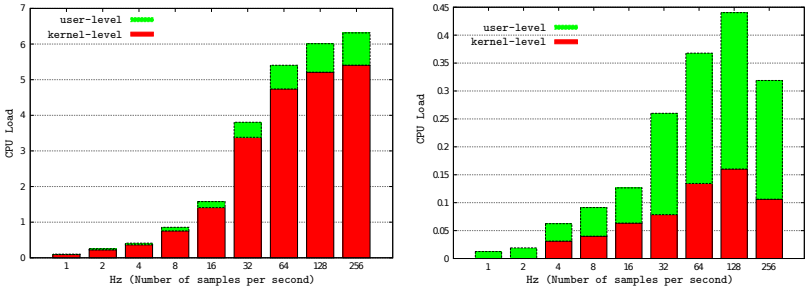


Figure 7: CPU load of (a) WallMon client (left) and (b) WallMon server (right)

The aforementioned configuration of the WallMon servers and clients was also used to measure the bandwidth of the server, as shown in figure 8. However, the values labeled *actual bandwidth* were measured internally by the server. *expected bandwidth*

represents a theoretical maximum, which is based on the average size of network messages received by the server during the measurements. For example, given 5 nodes, a sampling rate of 32 Hz and an average message size of 13000 bytes, a theoretical maximum would be: 5 x 32 x 13000 $\approx$ 1.98 MB/s.

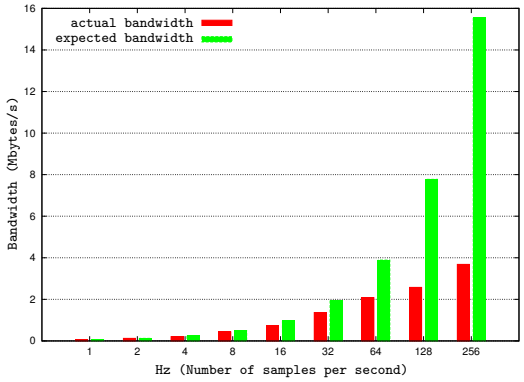


Figure 8: Server bandwidth usage at different Hz

7 Discussion

7.1 Initial Experiments

7.1.1 Microbenchmarks

The microbenchmarks presented in section 6.1 show that reading data from the `procfs` file system is the most expensive operation of sampling process level data. Reading data from `procfs` causes a context switch from user-level to kernel-level. Since each process has its own entry in `procfs`, a context switch will occur for every process included in a sample. The microbenchmarks also show that parsing the raw `procfs` data has some cost. The parsing is done by the `sscanf` function provided by a library in the C programming language. It might have been faster to use other approaches, such as a parser tailored for the format of `procfs`.

The Protocol Buffers library causes minimal overhead in the microbenchmarks. Its fast and has a compact data representation. On average in the microbenchmarks, serializing a n byte value with Protocol Buffers takes up only n bytes. Compared to XML's human readable format, Protocol Buffers provide a more compact representation in WallMon.

7.1.2 Macrobenchmarks

The macrobenchmarks presented in section 6.2 show that up until and including 32 Hz, the client's CPU load doubles whenever the sampling rate doubles. However, when moving from 32 to 64 Hz this pattern stops. What happened during this benchmark, was that the single-threaded collector sampled data too slow (on most samples) for the WallMon client to re-schedule it at non-overlapping intervals. It appears that the drop in sampling rate, presented in section 6.1, is the common case when running the complete WallMon system. It is unclear why this happens.

One reason could be the environment: When running WallMon in environments with, among others, more recent version of Linux, such a drop in sampling rate does not occur. At 256 Hz, the collector is scheduled constantly. This makes sense since the CPU load is slightly above 6%, which matches the share of one out of 16 (100/16=6.25) logical cores present at the cluster nodes.

The primary cause for the large amount kernel-level execution time compared to user-level execution time on the client, is reading from the `procfs` file system and sending data over the network. Reading from `procfs` contributes the most: Data read from `procfs` is represented as ASCII strings compared to compact serialized representation which is sent over the network, and process level sampling requires a context switch for each process in the sample, while sending data over the network requires one or a handful of context switches. Due to the `procfs` overhead, it might have been justifiable to implement a kernel module, as is done in [4]. Such a module has the potential to eliminate the cost of context switching.

The CPU load of the WallMon server can mostly be contributed to the internals of WallMon, the core runtime. The kernel-level execution time can be explained by receiving packets from the network, while its user-level execution comes from de-serializing and routing data. Also, the handler of the gnuplot module contributes to user-level execution time. The relatively low CPU consumption of the WallMon server indicates that the server (and client) is suitable for clusters consisting of more than five nodes. The CPU load at 256 Hz in figure 7b is lower than the load at the previous rate of 128 Hz. This anomaly highlights a challenge that barely has been addressed in WallMon, which is synchronization between collectors. What occurred in this measurement was that the collectors did not finish at the same point in time. It is likely that for some of the collectors, a handful of their measurements at 256 Hz were registered at 128 Hz at the handler executing on the server.

The bandwidth usage of the server correlates with the problem of obtaining data fast enough from `procfs`: At 64 Hz and higher, it is clear that the expected bandwidth is much higher than actual amount of data received by the server. Compared to traditional cluster monitoring systems, such as Ganglia [1], the numbers for the expected bandwidth show that process level sampling requires relatively much bandwidth in Wallmon. Because of this, it might not be practical to sample at high rates and/or sample data about all present processes.

7.2 Module System

The module system's support for arbitrary many modules executing simultaneously and a guarantee of sequential execution within each module, is an abstraction that simplifies. These guarantees and the programing model offered have similarities to the model implemented by Google's MapReduce framework [10]. Although MapReduce targets a different domain, collectors in WallMon and `map` in MapReduce both transform and forward data to a new transformation, and handlers and `reduce` usually aggregates large amount of data into something smaller and/or comprehensible. The inversion of control pattern implemented by modules and the guarantee of sequential execution also matches MapReduce.

One limitation with the module system is that users have to implement all functionality specific to gathering and usage of data. This might reduce productivity, however, it adheres to the end-to-end argument [8], which can be considered applicable to the diverse and performance sensitive activities of cluster monitoring.

8 Conclusions and Future Work

This paper presented WallMon, a tool for interactive visual exploration of performance behaviors in distributed systems. The system is client-server based, and its architecture is extensible through a module system that encapsulates functionality specific to monitoring. WallMon shows that sampling data with process granularity comes at an acceptable cost, and that visualizing this data with an approach inspired by information flocking, might reveal interesting patterns.

The next steps for WallMon include extending and improving the interactive visualization, such as exploring different ways of providing efficient and simple to use functionality for exploration of monitored data. The visualization will also be applied to additional distributed systems, such as high performance systems, which it yet has not been tested sufficiently out on. Moreover, additional experiments on WallMon will have to be carried out. These experiments should be carried out on a larger cluster compared to what was presented in section 6.

Acknowledgements

The authors thank the technical staff at the CS department at the University of Tromsø. This work is supported by the Norwegian Research council, projects No. 187828, No. 159936/V30 and No. 155550/420.

References

[1] F. Sacerdoti, M. Katz, M. Massie, and D. Culler, "Wide area cluster monitoring with Ganglia," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, dec. 2003, pp. 289 – 298.

[2] "HP Cluster Management Utility," http://h20311.www2.hp.com/HPC/cache/412128-0-0-0-121.html and documentation therein, last visited on 19-July-2011.

[3] G. Proctor and C. Winter, "Information Flocking: Data Visualisation In Virtual Worlds Using Emergent Behaviours," in *In Virtual Worlds 98*. Springer-Verlag, 1998, pp. 168–176.

[4] M. Sottile and R. Minnich, "Supermon: a high-speed cluster monitoring system," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 39 – 46.

[5] T. Ferreto, C. de Rose, and L. de Rose, "RVision: An Open and High Configurable Tool for Cluster Monitoring," in *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, may 2002, p. 75.

[6] K. Ren, J. López, and G. Gibson, "Otus: resource attribution in data-intensive clusters," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1145/1996092.1996094

[7] D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen, and O. J. Anshus, "Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays," in *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007*, Jun. 2007, pp. 75–83.

[8] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, November 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402

[9] M. Fowler, "InversionOfControl," 2005. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html

[10] J. Dean, S. Ghemawat, and G. Inc, "MapReduce: simplified data processing on large clusters," in *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

Appendix B

CD-ROM

The contents of the CD-ROM accompanying this thesis are:

- Copy of WallMon's source code, which is also available at `http://bitbucket.org/arild/thesis`. The Mercurial source code control system has been used during the development of WallMon, and can be used to review older versions of WallMon.
- A copy of the single paper presented in appendix A.
- A copy of the thesis, and copies of individual figures presented in the thesis.
- Videos demonstrating WallMon:
  - General use of WallMon on the Tromsø display wall.
  - WallMon monitoring the roller coaster application.
  - WallMon monitoring the gigapix [7] application.
  - WallMon monitoring the Weather Research & Forecasting (WRF) Model [8].