

# Analizing resource allocation for a distributed monitoring application

**Bogdan-Constantin Irimie**<sup>1,2,3,\*</sup>

<sup>1</sup>Institute e-Austria Timisoara, Romania

<sup>2</sup>Department of Computer Science, West University of Timisoara, Romania

<sup>3</sup>Department of Computing, Imperial College London, United Kingdom

\*bogdan.irimie90@e-uvf.ro

## ABSTRACT

In a cloud environment monitoring applications should be able to scale in order to satisfy the monitoring requirements offering good quality of service and efficient resource utilization. In order for this scaling to be possible, monitoring applications should be built from the ground up with this goal in mind and mechanism for auto scaling should be implemented in order to provide a fast adaptation time. This report describes the work that has been done in order to enhance a monitoring system with automatic resource allocation capabilities. The work describes possible architectures with their advantages and drawbacks and implementations with their technical challenges.

## Introduction

Monitoring plays an important role in today's offerings of computing because it enables cloud providers and consumers to check if the quality of service they have agreed upon is satisfied, additionally monitoring can be used to verify that a certain level of security is enabled.

Resources in the cloud can be provisioned and released with ease in a short period of time and monitoring systems should be able to scale just as fast in order to adapt to the new monitoring requirements. In order to adapt, the monitoring systems should rely on a resource allocation mechanism that can take into account the architectures of the monitoring systems and provide strategies for resource allocation. The architecture of the monitoring systems is very important because it dictates the means of obtaining scalability, some systems having architectures that allow scaling with ease by providing decoupled systems where each component can be scaled independently, while other systems are built on a monolithic architecture making scaling more challenging and in some cases even impossible.

In this report we will focus on a monitoring system that implements a pipe and filter architecture allowing independent scaling of all components, and thus providing fault tolerance and parallel execution of jobs.

## 1 Analyzing the system and its architecture

The architecture of the system is important because it dictates the means by which scalability can be obtained. In figure Fig. 4 the architecture of the system is presented, this architecture being an extension of [cite paper from SYNASC]. The system is composed of 7 components: FrontEnd, Scanner, Converter, Presenter, Remediator, a document database and a message queue. The first five components send messages using the queue and are completely decoupled from one another, from each component perspective the system is only composed of itself and one or more message queues. This decoupling provides great opportunity for replication and scalability.

## 2 Resource allocation for a system following a pipe and filter like architecture

This section focuses on the main challenges of implementing resource allocation without losing fault tolerance or decoupling between components. Replicas of the same component have no means of organizing, they just take the next job that is available in the queue and execute it, no mechanism for distributing the jobs is in place. Replicas of the same component can consume different amounts of system resources, depending on the type of job they are executing, but because they have no possibility to select the jobs they will execute, they just take the next job from the queue, we cannot predict on which of the replicas the job will be executed. Because of this unpredictability, the task of allocating resources becomes challenging and different methods have been proposed.

## 2.1 Method 1: Allows the system to balance work in a natural way and not intervene in job allocation

1

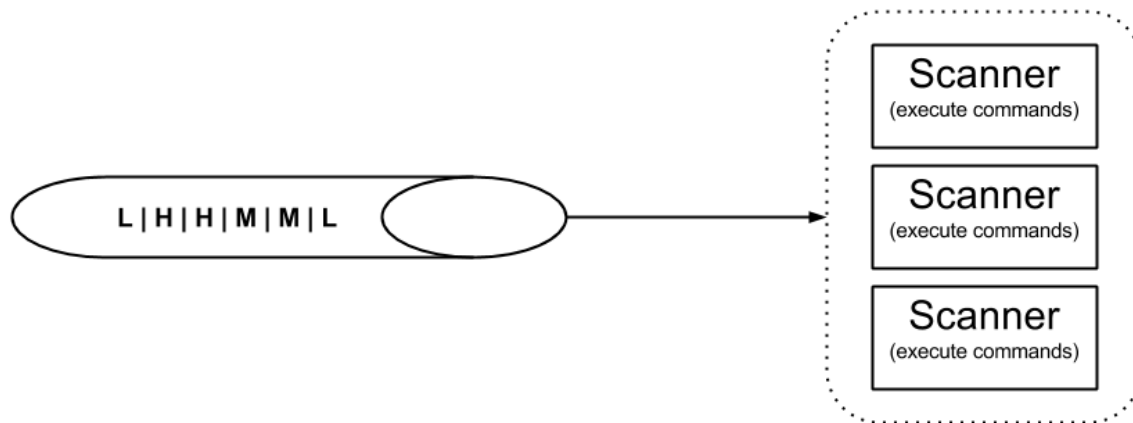


Figure 1. Random distribution of tasks

## 2.2 Method 2: Assign sizes to jobs and constrain component replicas to execute some type of jobs with a higher priority than others

## 2.3 Method 3: Create multiple queues deepening on the size of the jobs

## 2.4 Method 4: Use other types of data structure to send the jobs from one component to another

## 3 The problem

Although the system is very flexible and allows replication of components, it does not contain any component that is able to evaluate its state and make decisions regarding allocation and release of resources consequently scaling must be done manually. Manual scaling the system can become a complex problem because each component has different resource requirements and even the same component can use more or less of a resource, depending on the job that it must execute. Monitoring the system manually and scaling it quickly become unmanageable and an automated way to do the scaling is desirable.

In order to scale the system automatically, key performance metrics must be gathered and analyzed.

## 4 Gathering performance metrics

Some of the basic performance parameters that can be gathered for a component are CPU, memory and network usage. Knowing the resource utilization of one component allows us to take decisions regarding deployment of that component. For example we can have 2 component, one with high CPU usage and one with high network usage, it would be wise to combine replicas of those components on the same VM or to allocate the ones that require high CPU usage to VM's with powerful CPU. By having this insight on component resource utilization we are able to take those decisions and to use the resources that we have in an efficient way. But resource utilization for components can vary with time, so a static scheme is not very useful, because it becomes outdated in a short period of time. Continuous monitoring the components and taking actions on fresh performance metrics is crucial in guaranteeing optimal allocation strategies.

### 4.1 Monitoring CPU

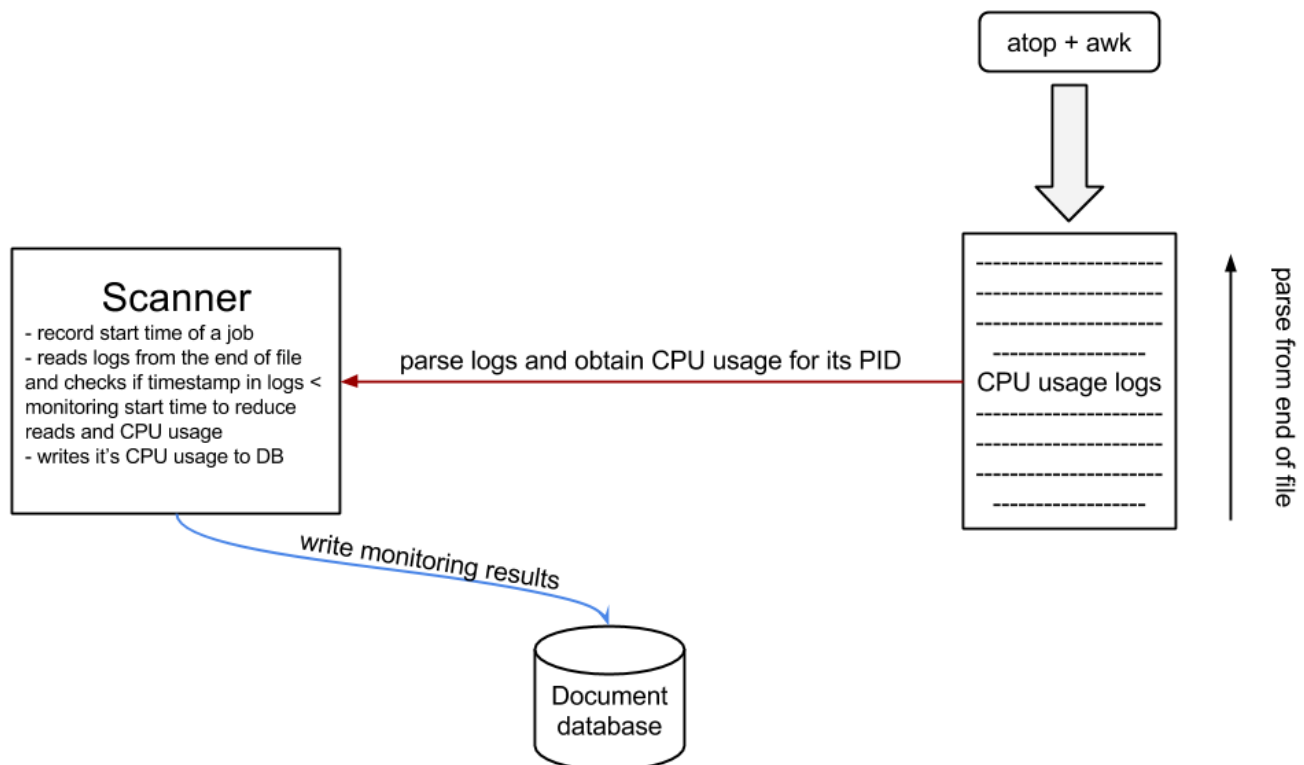
A first step would be to monitor the CPU usage in order to find a rough approximation of how much computing power does a component need. The main challenge here is to find tools that are able to monitor short lived processes (under 1 second) and to report with high accuracy the CPU usage because sometimes components can use very little CPU. Four tools / methods of gathering CPU usage statistics were reviewed:

- Using a class from the Java library called "ManagementFactory" is a good way to measure the Java program, the main drawback is that Jmap is an external program and it cannot be monitored by this library.
- Sigar library for Java can be used to monitor arbitrary process id's (PIDs), but the problem is that we first have to start the component, obtain its PID and only after we can start monitoring it with "Sigar" library, the main problem is that we might lose the CPU usage at the start of the process which can introduce high error rates.

- pidstat and top have the same drawbacks. they miss the CPU usage for the process if it is very short lived, they miss the CPU usage of the process right before it ends, but they are better alternatives than the tools mentioned before because they can monitor processes from the moment they are created, not missing the CPU usage from the start of the process.
- atop can monitor processes from the moment they are created and capture the CPU usage of the process right before it terminated. Because it is able to capture the CPU usage at the start and at the end of the lifetime of a process, it provides very accurate results introducing small overhead on the system. The only drawback that was found is related to the root privileges that are needed in order to run this tool.

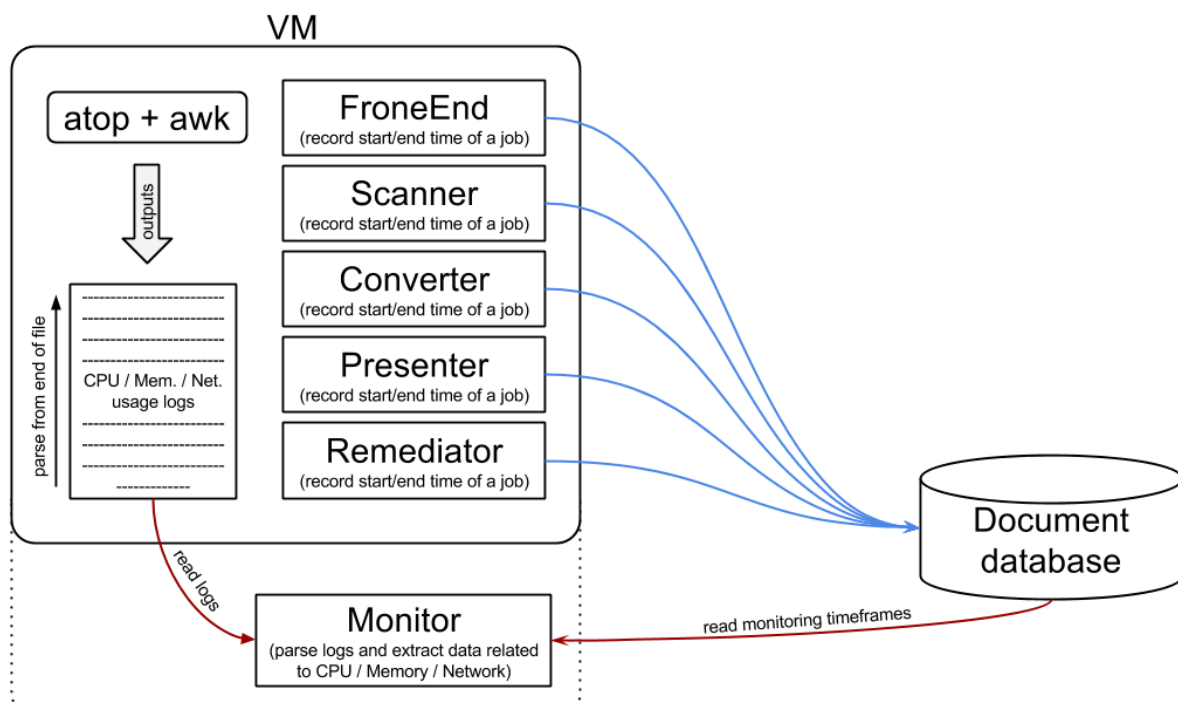
The tool atop was chosen to measure CPU usage because of its high accuracy and low overhead. In the first implementation "atop" was called at the beginning of each new job for each component, but this approach had many drawbacks: because it took some time to start atop, some CPU usage might be lost, and it required launching atop for each job and for each component making it consume considerable system resources. The second approach was to use launch "atop" from an external script, parse its results and store them in a log file. Each component that is interested in its or others component CPU usage can just parse the log and search after the PID of interest between a time intervals. This second approach was much better because it allowed using only 1 atop instance to gather CPU usage statistics for all running processes on a system, and each component that was interested in its CPU usage could have just parsed the log file. Another advantage was that it decoupled the monitoring activities from the components that were monitored. The architecture for the second approach is presented in Fig. 2, the log document is read from the bottom upwards, because the chances are that the monitoring results will be closer to the bottom than to the top.

Fig. 3 presents an architecture that is even more decoupled than the one that was implemented, the parsing is done in a different component "Monitor" and the components only need to save the beginning and the end time for a job they executed, the parsing of the monitoring data would be done by the "Monitor" component. This architecture adds more flexibility, because the Monitor component can be on the same machine to minimize network traffic (otherwise the log would have to be transferred to the machine that holds the "Monitor" component) or on a different machine.

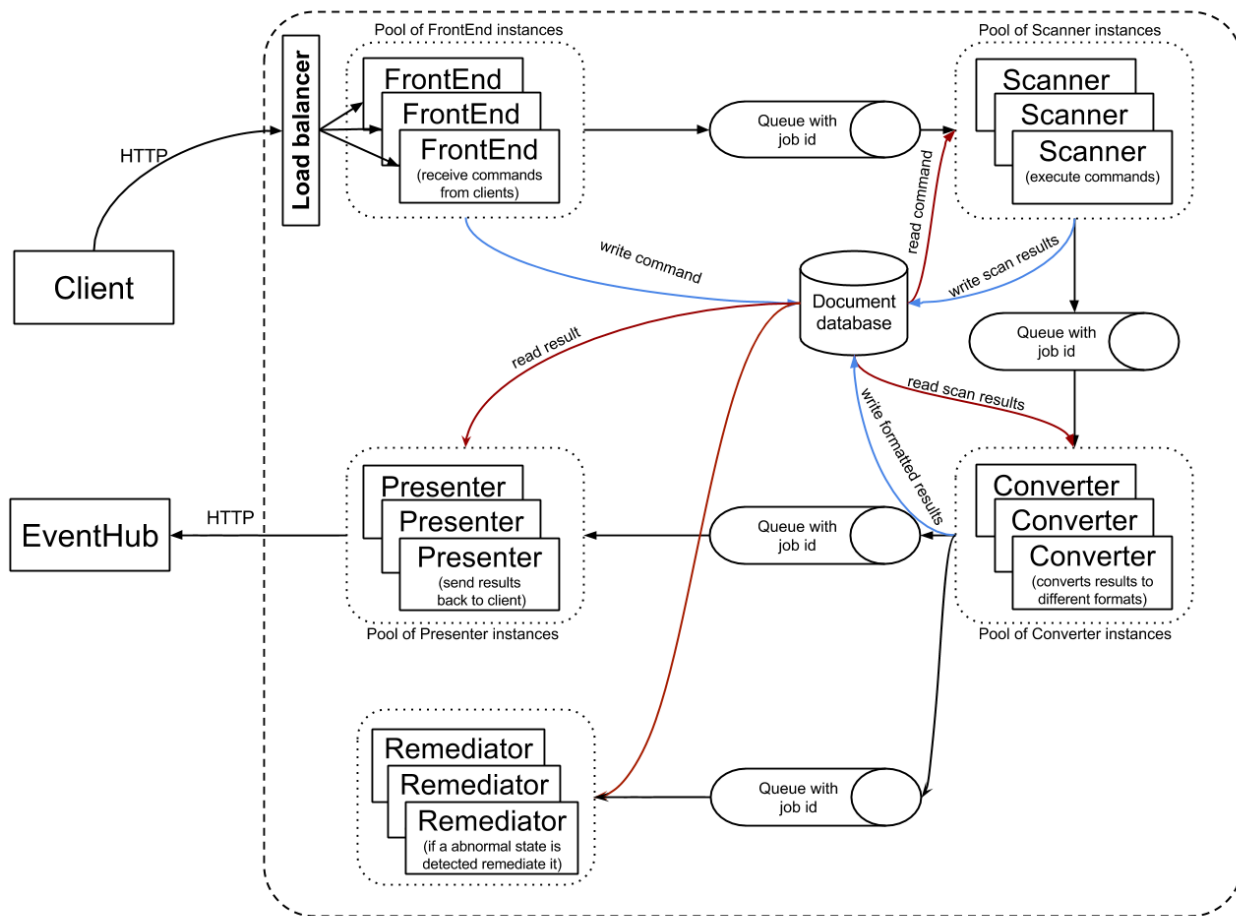


**Figure 2.** Distributed monitoring system architecture

## Acknowledgements



**Figure 3.** Distributed monitoring system architecture



**Figure 4.** Distributed monitoring system architecture