Analysing resource allocation architectures for a distributed monitoring application

Bogdan-Constantin Irimie*†
*Institute e-Austria Timisoara, Romania
†Department of Computer Science, West University of Timisoara, Romania

***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. The paper 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 cloud 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, by checking different host parameters.

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 relay 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 have architectures that allow scaling because they are build from decoupled components that can be scaled independently, very similar to the microservices architecture [1], while other systems are build on a monolithic architecture making scaling more challenging and in some cases even impossible. For systems that are built from multiple components , like the ones that fallow a pipe and filter architecture, resource allocation can be a difficult task because components have little or no knowledge about the rest of the system.

In this paper we will focus on building different architectures for a pipe and filter system in order to allow the system to automatically scale depending on the load and the quality of service we want to enforce. System scaling is obtained via replication for all components. Replication does not only allow us to ensure scalability, but fault tolerance as well.

The rest of the paper is structured as fallows. Section 1 presents similar efforts to provide resource allocation while section 2 describes the pipe and filter architecture and the system built using this architecture. Resource allocation strategies and the architectural changes required are described in Section 3. Gathering of performance metrics is presented in Section 4 and we conclude with conclusions and further work that can be done.

1. Related work

In [1] the authors survey the resource allocation efforts for distributed systems. They categorise the systems in three types: cluster, grid and cloud. Characteristics are analysed for each category and systems are included in categories based on their characteristics. The authors of [2] present a proactive resource allocation mechanism based on extended queueing network models that is able to decide what resources to rent in the cloud in order to run an application with appropriate quality of service. The authors used AWS[2] spot instances and tried to find the appropriate bid in order to keep the VM alive and achieve low costs of renting.

LINE [3] tool is a solver for queuing network models and can be used in order to evaluate performance parameters like response time for complex applications. Using this tool and models of the system, we can predict the performance of the system and know what resources should be allocated in order to achieve the desired quality of service.

A cloud controller, FQL4KE, that has self-adaptive and self-learning capabilities is presented in [4]. The cloud controller tries to adjust scaling policies at runtime by combining fuzzy control and Fuzzy Q-Learning. The approach mitigates the drawbacks of reactive resource allocation and proactive allocation by enhancing the fuzzy controller with machine learning capabilities.

2. Pipe and filter architecture and the monitoring system built using it

The pipe and filter architecture implies that each component of the system is able to read data from a stream and output data to another stream. Components play the role of

1. http://microservices.io/patterns/microservices.html

2. https://aws.amazon.com/

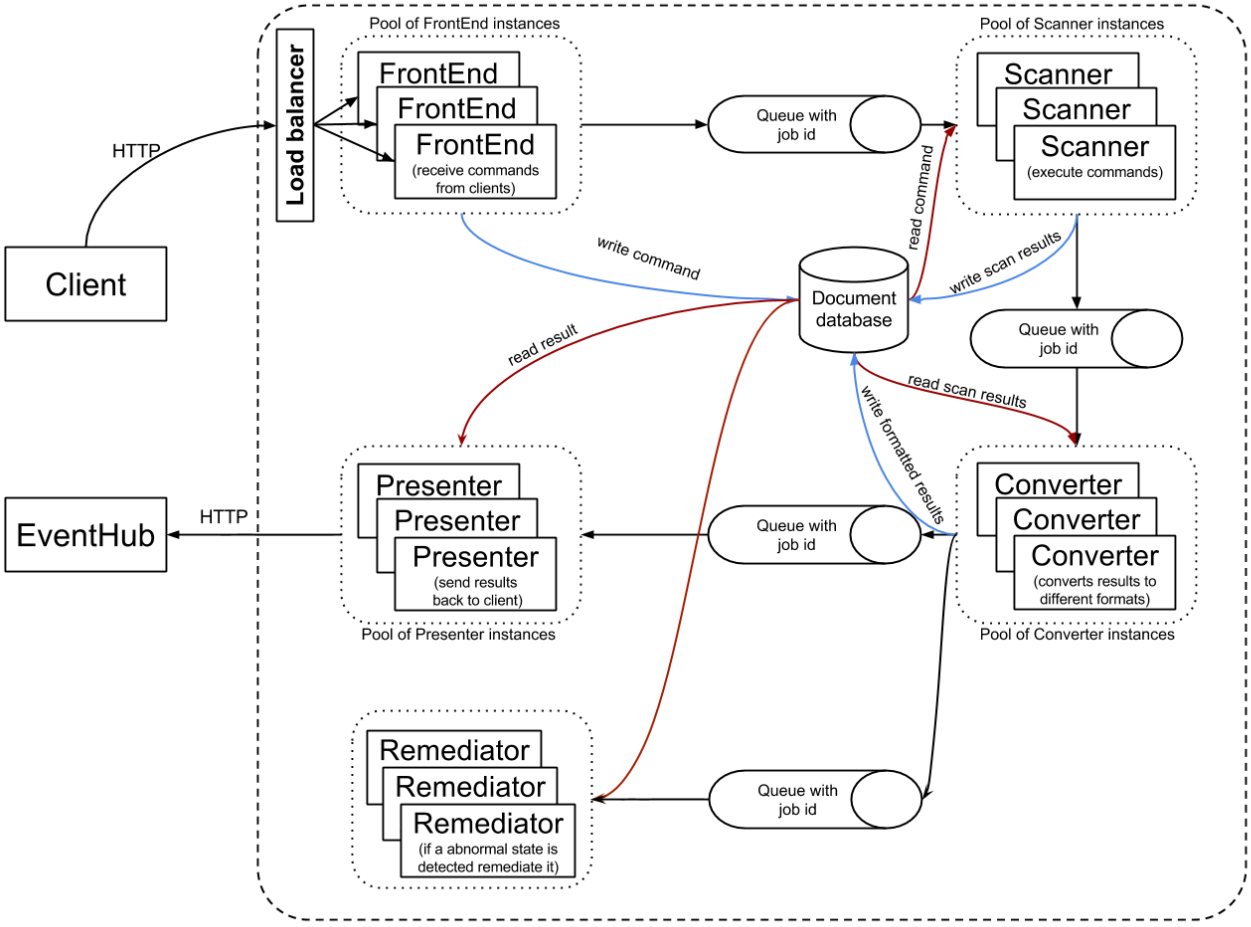

Figure 1: Distributed monitoring system architecture

filters and streams play the role of pipes. Filters should not share state with other filters and should not make assumptions about other filters in the system. Among the properties provided by such an architecture are: simpler understanding of the processing done by a system by analysing the order of the filters and their responsibilities, filters are easy to reuse, maintain and replace, concurrent execution of filters is supported.

For every system the architecture is important because it dictates the means by which scalability can be obtained. Fig. 1 presents the architecture of the system we want to extend. The architecture is an extension of the one presented in [5]. 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 queues and are completely decoupled from one another. From each component perspective the system is only compose of itself and one or more message queues. This decoupling provides grate opportunity for replication and scalability, but it imposes some constraints on the allocation of jobs to replicas of the same component.

The message queues work on the first in first out principle (FIFO) and are only responsible for delivering messages from one component to another. For storing data between phases of processing a document database is used. Because a database is used, traffic in the queues is very low and intermediary and final results are stored in the database for audit purposes. Both of those components, the message queue and the database provide fault tolerance by means of replication.

3. Resource allocation for a system with a pipe and filter architecture

This section focuses on the main challenges of implementing resource allocation without losing benefits like fault tolerance and decoupling between components offered by the pipe and filter architecture. Each of the components can be replicated, but 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. Component replicas have no possibility to select the jobs they will execute, they just take the next job from the queue, so we cannot predict on which component replica the job will be executed. Because of this unpredictability, the task of allocating replicas of components to VMs is challenging and different methods have been proposed in subsections 3.1, 3.2, 3.3, 3.4.

Replicas of the ”Scanner” component were used as a case study, jobs present in the queue were marked as ”H” (high), ”M” (medium) and ”L” (low) denoting the CPU usage necessary to complete a job. The following methods were proposed in order to extend the system, taking it one step closer to automatic allocation of resources.

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

The ”Scanner” replicas execute jobs from the queue on the first come first served order. If a scaner component is idle it will just execute the next job in the queue, but because jobs consume different amounts of CPU, some replicas may execute jobs that require high CPU usage while other replicas will execute jobs that require low CPU usage.

The main advantages of this strategy is that it does not require any changes to the system, so no extra complexity is added. Fig. 2 presents the architecture of the first approach. Another advantage is that components are decoupled as envisioned in the initial architecture Fig. 1.

The main drawback of this approach is that, because balancing is done by chance, in worst case, the system could be completely unbalanced, some component executing only high intensive CPU jobs while others executing only low CPU intensive jobs. If more than one ”Scanner” replica run on the same VM, than the chances are even lower for this to happen because, even if some component would be ”unlucky” enough to only execute CPU intensive jobs, they would be balanced by other components that execute low intensive CPU jobs from the same machine.

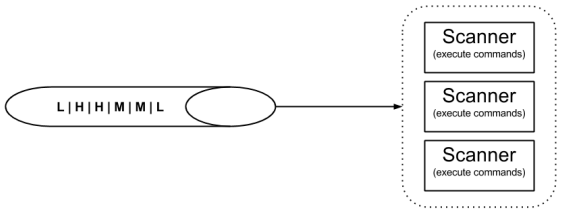

Figure 2: Random distribution of jobs

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

The second approach Fig. 3 proposes using some priority scheme inside the ”Scanner” component replicas so that we can place ”Scanner” replicas on machines that satisfy the required computational needs. For example, if we have a machine with high CPU power, we can allocate ”Scanner” replicas that execute CPU intensive jobs to that machine.

The main advantage of this approach is that we can allocate ”Scanner” replicas with different priority schemes to different types of VM’s, obtaining a coordinated assignment of component replicas with a known resource utilization profile. Each replica of the ”Scanner” component withdraws messages from the queue, and it makes a decision if it can execute that job or not based on the job type, for example some replicas can only execute CPU intensive jobs, marked with ”H”, while others can execute only low CPU intensive jobs, marked with ”L”.

If a replica cannot execute the job, it will just put it back in the front of the queue. Because of this, some messages may spend a very long time in the queue, or even never get executed, if they are always delivered to components that cannot handle them and are put back in front of the queue. Another problem is related to the stress put on the queue because messages are inserted back in the queue if they cannot be executed by a ”Scanner” replica losing the ordering of jobs as well.

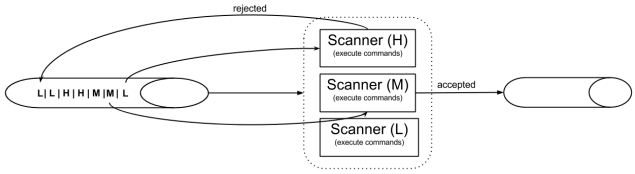

Figure 3: Distribution of jobs based on resource usage

3.3. Method 3: Create multiple queues depending on the size of the jobs

The third method Fig. 4 uses a messages queue for every type of job, for out example, there are three types of jobs ”H”, ”M” and ”L”, we need three queues and each ”Scanner” replica will have a priority scheme that will dictate the priority it gives to each of the message queues. If the ”Scanner” replica does not find messages in the queue with the highest priority, it will take messages from another queue with lower priority.

The main advantage of this method, compared to the second method 3.2 is that messages are only withdrawn from queues reducing the load on the queue, and replicas of components don’t have to search for jobs they can execute, they can just check if there are jobs available on the queues in the order of their priority scheme. Another advantage is that replicas will not be idle or losing time searching

for jobs, they will be executing jobs from queues with low priority to them if there are no jobs in the high priority queue.

The main drawback is that a new message queue is required if we insert a different type of job, and the "Scanner" replicas will be more complex because they need to keep track of all the queues and the priority schemes.

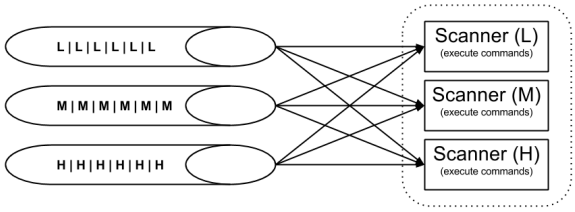

Figure 4: Multiple queues, one for each type of jobs

3.4. Method 4: Use other types of data structure to send jobs from one component to another

The forth method Fig. 5 tries to eliminate all the problems related to job type selection by the "Scanner" replicas by replacing the message queue with a different data structure, in our case a map.

In this approach, each "Scanner" replica can inspect all the jobs that can be executed and pick the one best suiting their capabilities.

Although this approach eliminates most of the disadvantages we had when using message queues, it eliminates most of the guarantees that were offered by them as well. Job order is not preserved, the map is shared by all replicas and operations to it must be serialized to avoid conflicts, or a conflict resolution mechanism should be implemented. Some mechanism for acknowledgement of jobs completion must be implemented in order to avoid losing jobs when components fail.

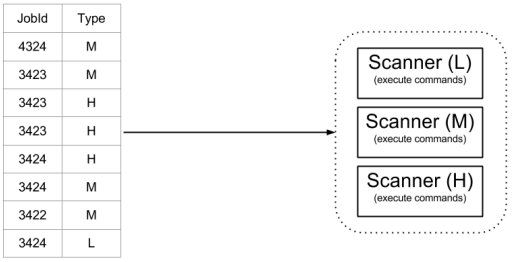

Figure 5: Map with multiple size jobs.

3.5. Choosing between the methods

One of the assumptions made at the beginning of this section was that we can determine in advance the CPU usage for jobs, but this can prove to be very difficult in practice because there are more than three types of jobs and their CPU usage depends on factors like network speed, speed of response for a particular host, network configuration etc.

Each of the described methods has advantages and disadvantages and one should always chose the appropriate method suiting their needs.

As a starting point we chose the method presented in subsection 3.1 because it does not break any of the properties of the system, it can be later extended to any of the more advanced methods and in practice we observed that "Scanner" replicas tend to balanced their CPU usage, without any intervention.

4. Gathering performance metrics

Although the system is very flexible and allows replication of components, it does not contain any component that is able to evaluate it's 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 analysed. 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 efficient to collocate replicas of those components on the same VM or to allocate the ones that require high CPU usage to VM's with more computing power. By having this insight on component resource utilization we are able to take those decisions and use the resources 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 processed, under 1 second, and to report with high accuracy the CPU usage in order to take decisions for component allocation. Four tools/methods for gathering CPU usage statistics were reviewed:

- "ManagementFactory" is a class from the Java standard library that is able to collect performance met-

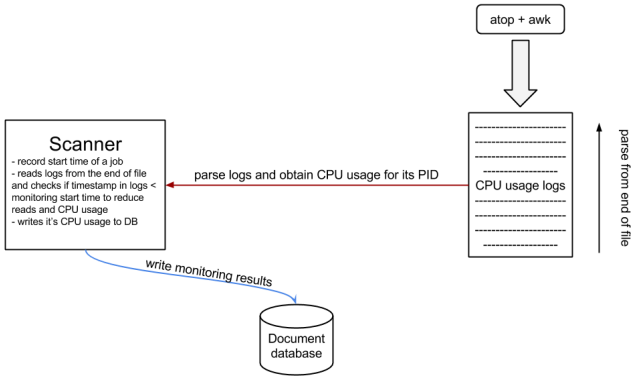

Figure 6: Distributed performance monitoring system architecture

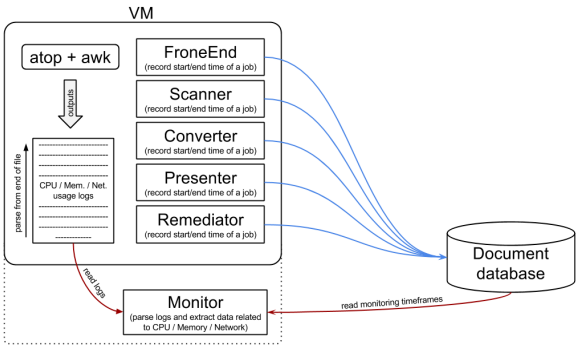

Figure 7: Distributed performance monitoring system architecture

rics from the JVM. The main drawback is that Nma[3], one of the tools used by the system we want to extend, is an external program and it cannot be monitored by this library.

- "Sigar"[4] library for Java can be used to monitor arbitrary process identifiers (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 be introduce high error rates in the allocation phase.
- "pidstat" and "top" tools have the same drawbacks. they miss the CPU usage for the process if it is very short lived and 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 crated, not missing the CPU usage from the start of the process.
- "atop" tool can monitor processes from the moment they are crated and capture the CPU usage of the process right before it terminates. 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 it's high accuracy and low overhead. In the first implementation "atop" was called at the begging of each new job for each component, but this approach had some problems because it took some time to start "atop" and CPU usage was not recorded for the start of the process. This approach required launching "atop" for each job and for each component making it consume considerable system resources.

The second approach was to launch "atop" from an external script, parse it's results and store them in a log file. The second approach was much better because it allowed using only one "atop" instance to gather CPU usage statistics for all running processes on a system, and each component that was interested in it's 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. 6, 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. 7 presents an architecture that is even more decoupled than the one presented in Fig. 6, 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 ads more flexibility, because the "Monitor" component can be on the same machine to minimize network traffic, or on a different machine in order to minimize computational costs on the machines that is being monitored. Using this approach we can use the monitoring system independently of the target we are monitoring, allowing us to separate the monitoring concern from the logic that is implemented in the tools that we are monitoring.

Conclusions

The paper presents extensions for a system that is built on a pipe and filter architecture. Four possible architectures Fig. 2, Fig. 3, Fig. 4, Fig. 5 were suggested, each one with its advantages and disadvantages. Several ways of monitoring performance were tested and a monitoring architecture was built around "atop" software tool.

The work from this paper bring to system from [5] one step closer to the main objective of dynamic allocation of components on virtual machines. Constant change in resource utilization of each component replica must be

3. https://nmap.org
4. https://github.com/hyperic/sigar

taken into account in order to predict the VM where to allocate a component replica and continuous performance monitoring must be implemented in order to record the change in resource utilization of each component replica. The next step is to build a component that is able to use the performance metrics gathered and make allocation plans taking into account the constraints of the architecture of the system.

Acknowledgements

The work conducted in this report was supported by EC FP7 project Secure Provisioning of Cloud Services based on SLA management, SPECS (Grant Agreement no. 610795) and Competitive Multidisciplinary Doctoral Research in Europe (CDocMD) POSDRU/187/1.5/S/155559 project. The author would like to thank members of the Department of Computing from Imperial College London for their advice and guidance.

References

[1] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, J. Kolodziej, A. Y. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, P. Bouvry, H. Li, L. Wang, D. Chen, and A. Rayes, "A survey on resource allocation in high performance distributed computing systems," *Parallel Computing*, vol. 39, no. 11, pp. 709–736, 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S016781911300121X

[2] D. J. Dubois and G. Casale, "Autonomic Provisioning and Application Mapping on Spot Cloud Resources."

[3] J. F. Perez and G. Casale, "Assessing SLA Compliance from Palladio Component Models," *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 409–416, 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6821177

[4] P. Jamshidi, A. Sharifloo, C. Pahl, A. Metzger, and G. Estrada, "Self-Learning Cloud Controllers: Fuzzy Q-Learning for Knowledge Evolution," 2015. [Online]. Available: http://arxiv.org/abs/1507.00567

[5] B. C. Irimie and D. Petcu, "Scalable and fault tolerant monitoring of security parameters in the cloud."