Assessing the overhead and scalability of system monitors for large data centers

Mauro Andreolini
Department of Information Engineering
Via Vignolese, 905/b
41125 Modena, Italy
mauro.andreolini@unimo.it

Michele Colajanni
Department of Information Engineering
Via Vignolese, 905/b
41125 Modena, Italy
michele.colajanni@unimo.it

Riccardo Lancellotti
Department of Information Engineering
Via Vignolese, 905/b
41125 Modena, Italy
riccardo.lancellotti@unimo.it

ABSTRACT

Current data centers are shifting towards cloud-based architectures as a means to obtain a scalable, cost-effective, robust service platform. In spite of this, the underlying management infrastructure has grown in terms of hardware resources and software complexity, making automated resource monitoring a necessity.

There are several infrastructure monitoring tools designed to scale to a very high number of physical nodes. However, these tools either collect performance measure at a low frequency (missing the chance to capture the dynamics of a short-term management task) or are simply not equipped with instrumentation specific to cloud computing and virtualization. In this scenario, monitoring the correctness and efficiency of live migrations can become a nightmare. This situation will only worsen in the future, with the increased service demand due to spreading of the user base.

In this paper, we assess the scalability of a prototype monitoring subsystem for different user scenarios. We also identify all the major bottlenecks and give insight on how to remove them.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Metrics—*performance measures*; C.2 [**Computer, Communication Networks**]: Network operations—*network monitoring*; C.2 [**Computer, Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Measurement, Performance, Scalability, Data centers

1. INTRODUCTION

The improvement of energy efficiency, the optimization of hardware resources utilization and the flexibility of an on-demand, pay-as-you-go servicing scheme are all driving factors behind the adoption of the Private Cloud Computing paradigm in modern data centers [1]. In spite of this, the size of data centers continues to grow in terms of hardware components, complexity of software interactions and traffic volume due to users. Nowadays, it is is not uncommon to see even mid-sized data centers deployed over one hundred of nodes, each one running several tens of virtual machines, in the range from 20 to 100 and even more [6]. The underlying management platforms interact with the Virtual Machine Monitors and provide a set of system-level tasks oriented to basic management of virtual machines (creation, distruction, live migration). A number of tools and services are available for monitoring a large data center infrastructure (*ZenoSS*, *VMware Infrastructure* among others). At their core, the vast majority of these solutions uses popular open source monitors such as *SAR*, *Nagios*, *Cacti* that collect samples of performance indexes (CPU utilization, memory utilization, disk utilization, network throughput) at the system and at the application level. Usually, the default sampling interval chosen in these products and in previous literature [8] is five minutes.

In this scenario, we evidence two problems. First, the number of performance samples collected in a time unit can be very high. For example, in a data center running 100 physical nodes (each one hosting 100 virtual machines), a monitor that collects 5 distinct performance indexes for each host and for each virtual machine has to manage $N = N_{host} + N_{app} = 100*5+100*100*5 = 50500$ samples per time unit. Second, some management tasks (such as live migration of virtual machines) take typically less than five minutes to complete [3]. Consequently, the performance samples obtained by monitors run into a serious risk of becoming stale. Thus, it becomes crucial to a monitoring infrastructure to shorten the five minutes sampling interval. In this paper, we try to answer to the following basic questions.

1. Is it at all possible to design a low-level monitoring infrastructure that scales to N performance samples, possibly working with short sampling intervals?

2. Which bottlenecks will the proposed monitoring infrastructure come into for different usage patterns (several monitors on few physical nodes, few monitors on several physical nodes)?

Our focus is on assessing the scalability limits of a realistic, barebone prototype of an infrastructure oriented to the following basic operations: monitor performance samples at a high frequency, mark them as invalid if they do not fulfill

CloudCP'11 April 10, 2011 Salzburg, Austria

some acceptance criterium, insert them into a database for later retrieval and processing. We have instrumented this prototype using, where possible, already existing and well established software modules. We have run an extensive set of experiments in the Emulab environment [7], using common off-the-shelf hardware for a data center. Our major findings can be summarized as follows.

- Collecting and storing performance samples through standard OS tools is a very cheap operation that consumes only small fractions of computer resources. Retrieving more indexes within the same monitor comes at virtually no cost.
- Collecting system-level performance samples through a single monitor module per physical node allows the monitoring subsystem to scale to hundreds of nodes and to collect up to 6825 samples per time unit (far more than $N_{host} = 500$). The main bottleneck is memory, followed by the CPU. The network is underutilized.
- Collecting system-level and application level performance samples through multiple monitor modules per physical node is problematic for two reasons. First, the monitored host suffers a high resource consumption (memory utilization overcomes 0.1 already with 32 monitor probes). Even worse, the monitoring subsystem collapses at 16 hosts, the memory and the CPU of the database being saturated. Under these conditions, the monitoring subsystems collects and stores 7680 samples per time unit (far less than $N_{app} = 50000$). It is mandatory to use multiple, independent monitoring subsystems (at least 6) to achieve our goal of $N = 50500$ samples per time unit. Retrieving and processing performance samples from different acquisition databases with the goal of providing a coherent representation of resource state is an interesting topic for future research.

The paper is organized as follows. Section 2 describes in greater detail the requirements and the design of the prototype used to carry on the following performance analysis. In Section 3 we present the results of our scalability assessment. Section 4 concludes our discussion with some insights for future work.

2. THE MONITORING INFRASTRUCTURE

2.1 Basic requirements

Scalability. The proposed monitoring infrastructure must scale with the increasing number of hardware and software components in the data centers. On the one hand, the monitor must be able to collect data from a growing number of physical hosts. This requirement is shared with the vast majority of existing tools. On the other hand, with the adoption of virtualization, monitoring an increasing number of processes on a single node is also a crucial task.

Robustness. Current monitoring applications can fail in various ways. Some of them are trivially detectable; for example, when a probe crashes, no data will ever be collected (and stored) until it is resumed. Another common failure is the collection of out-of-range values (for example, a negative utilization). Other failures are more subtle; an otherwise perfectly valid sequence of performance samples is interspersed with outliers. A crucial issue in a modern monitoring infrastructure is to detect these functional anomalies as soon as possible and to mark the corresponding performance samples as invalid. Keeping a copy of the invalid samples helps in diagnosing problems.

Short term processing Many management tasks of a cloud-based data center must take decisions at very short time scales, in the range of seconds, typically under a minute. One notable example is live migration of virtual machines across different physical nodes. Monitoring at short time scales is a critical operations for two reasons. First, there is typically no time for complex analyses on the monitoring data obtained from the probes. The decisions taken using these values must focus on avoiding disasters rather than on achieving optimal performance. Second, the volume of monitoring data can be very high. Thus, even the storage of the performance samples in a database for later analysis can be a problem due to the high computational overhead and to the disk space consumption involved.

2.2 Logical design

In Figure 1 we propose a high level design of the proposed monitoring infrastructure that will be used throughout the experimental analysis. At the heart of the prototype is the *acquisition subsystem*, which is responsible for gathering and storing resource performance samples at regular time intervals. An acquisition subsystem is built from several *monitored hosts* and a set of *acquisition databases* running on one or more hosts. A monitored host is characterized by one or more resource sets $R_1, \ldots, R_j$ that are used by the operating system or by different applications. For example, one of these sets may represent resources (hardware and software that is, CPU, memory, storage, file and socket desctiptors) used globally at the host level; one other set may represent the resources used by a particular process (a VM), and so on. A *monitor module* M_j pipes in the output of the monitor modules to extract periodically performance samples for each resource in a given set R_k. Besides collecting, the monitoring module also checks that the monitored information is actually available and valid. In this preliminary version of the prototype, we have implemented a basic "range check" that marks the monitored samples as invalid if the corresponding values lie outside of a configurable, fixed interval. The monitor module has been designed with modularity in mind and can be easily extended to operate a pipeline of checks, with the goal of implementing more sophisticated data validation strategies. However, introducing further checks into the monitor module, while interesting, would be against the main scope of this paper that is, assessing the scalability limits of a realistic monitoring infrastructure.

As soon as a set of performance samples has been collected and marked as valid, it is inserted into one *acquisition database*, which makes the data available to a short term data filtering subsystem. The acquisition database, being accessed very often in a mixed read-write scenario, is most likely to become the bottleneck of the whole monitoring infrastructure. For this reason, it must be characterized by a very high performance, even at the expense of more sophisticated functions (triggers, stored procedures) that may be offered by more traditional DBMS. Even more, the acquisition database does not need to offer persistent storage, since

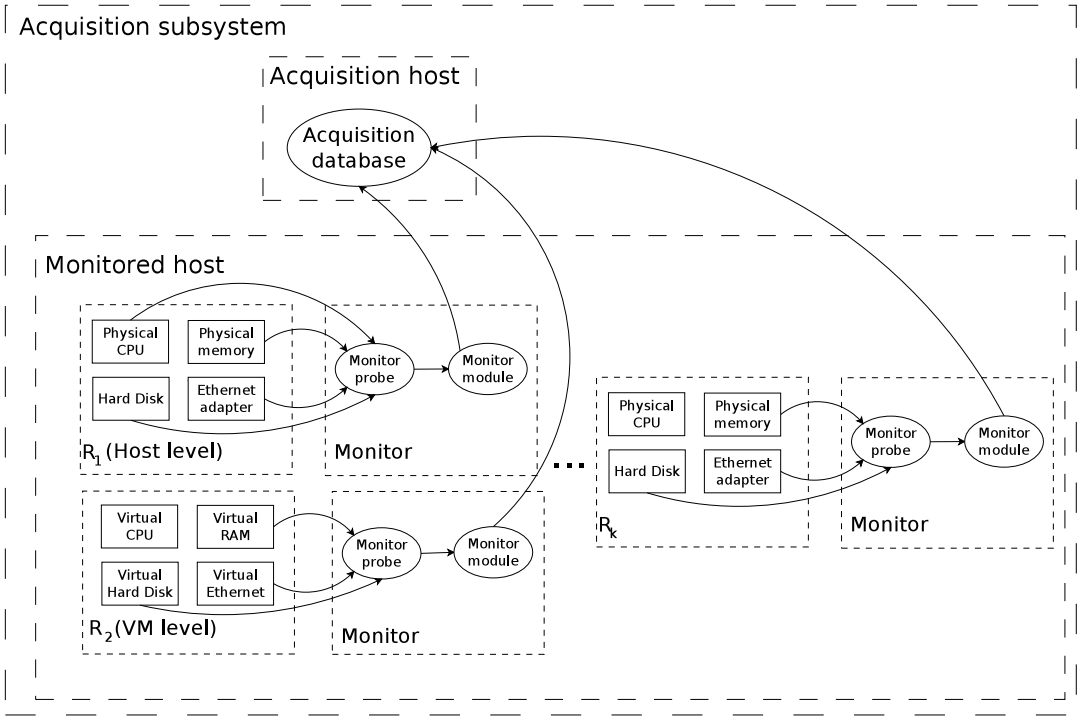


Figure 1: High level design of the monitoring infrastructure.

the collected raw performance samples, after being processed by other modules, can be simply discarded. Adding more RAM-DBs is straightforward in a multi-core host; we just instantiate another DB server on another port and pin the process to one particular core, to avoid CPU cache bouncing effects which would introduce context switching overheads.

2.3 Implementation details

We have implemented the monitor module in 85 lines of Python code. The choice of Python is motivated by its simplicity and by the availability of simple libraries for database communication and inter-process communication. The monitoring tool used to extract the performance samples is *vmstat*. It is also possible to extract per-process statistics through the *pidstat* utility, which has a lower resource consumption because it operates on a per-process basis instead of summing statistics over all the processes executing in the machine. Our choice fell on vmstat for the sake of simplicity; in this way, we avoid dealing explicitly with different process IDs, and the starting scripts are simpler. Another advantage of vmstat over pidstat is that we can collect more performance samples. We believe that this assumption does not invalidate our results.The monitor modules are written in Python.

We have chosen as a DBMS platform MySQL v5.1 and the MEMORY storage engine, which stores the entire set of tables in RAM. We have also investigated other solutions, but they where either object-oriented or unveiled some problems in their programming API.

3. EXPERIMENTAL ASSESSMENT

3.1 Testbed

We have used the Emulab network testbed facility [7] to deploy the architecture proposed in Section 2. All machines are equipped with several Gigabit Ethernet interfaces, 2GB physical memory and a SATA 7200 RPM local hard disk. The acquisition and short term processing engines run on Dual-Core 2.4GHz nodes, while the monitors run on Intel 3.0GHz nodes.

Each test runs for a fixed time of 4000s, which we believe is sufficiently long to obtain a statistically relevant number of performance samples. During this time, a given set of monitoring hosts is configured to spawn a desired number of monitoring processes. These processes collect performance samples with a default frequency of 1HZ and send them to a specific acquisition processes, which stores them in a RAM DB. In some tests, a short term analysis process extracts data from an acquisition DB, performs an exponential moving average and stores the data into a short term RAM DB. To evaluate the resource consumption of the monitoring system without any bias due to external operating conditions, the machines hosting the testbed do not run any other application and the testbed is setup from scratch for every test.

3.2 Scalability

In previous literature [5], the overhead of a monitoring subsystem tends to be measured in terms of CPU utilization at the monitored host. This can be very misleading since (a) it makes the assumption that the CPU is the main bottleneck resource; (b) the acquisition DB contributes significantly to the scalability of the architecture. In this paper, we will consider several critical resources at both sides. On the monitored host, we need to limit the resource utilization of the probes to avoid wasting too many system resources. Finding a good threshold is a difficult subject and

is certainly out of the scope of this paper; previous works also show that it can be very difficult to discern significant changes in the load staus of a resource [2, 4]. So, we are left with defining a reasonable interval of resource utilization arbitrarily; we choose the [0.0, 0.1] range. If any of the considered resources stays in this interval, we assume that the monitoring subsystem is operating efficiently. If at least one resource exceeds this threshold, we assume that the monitoring subsystem is taking too many resources. On the other hand, on the acquisition host, we want to use the available computing resources at their full potential. Here, we are interested in assessing the scalability limits of the acquisition DB.

To answer the remaining question of this paper, when scaling to multiple monitored hosts we also compute the number N of performance samples collected and stored per time unit and compare it with current values $N_{host} = 500$ (number of host-level samples collected in a medium-size data center per time unit) and $N_{app} = 50000$ (number of process-level samples collected in a medium-size data center per time unit) mentioned in the introduction. The quantity N is a function of the number of monitored hosts h, the number of monitor modules p executing in each monitored host, and the number of metrics m collected by a single monitor module during time interval t: $N(h, p, m, t) = \frac{h*p*m}{t}$.

3.3 The acquisition subsystem

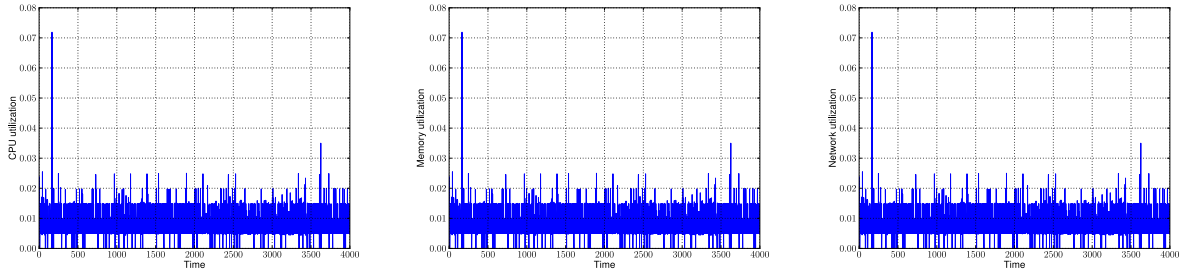
Figure 2 shows the resource utilization over time of a single monitor module that collects one performance index and sends it to a remote RAM-based DB. Since both the monitor and the RAM-based DB consume only CPU, resident memory and network resources, we only report the corresponding performance samples in Figures 2(a), 2(b) and 2 (c), respectively. We have verified that, in an otherwise idle system, the consumption of other system resources is very low, when not null; thus, they will be omitted in the analysis. As can be seen, except for a single CPU utilization outlier value (0.0719 at t=167s), the resource consumption of a single monitor process is very low and contained during a run. The medians of CPU, memory and network utilization are 0.01, 0.003 and 0.0007, respectively. Figure 3 shows the resource consumption of the acquisition DB over the same run. The medians of CPU, memory and network utilization are 0.01, 0.003 and 0.0007, respectively. We can conclude that retrieving performance samples through standard OS tools and storing them into a DB is a very cheap operation even within a dynamic language runtime environment.

To monitor the internal state of a single node, it is often necessary to collect several performance indexes. For the sake of efficiency, a single process is used to perform the collection of all the required indexes. In the next set of experiments, we measure the resource consumption of such a monitor module for an increasing number of performance indexes. To give a more compact visualization of the dispersion and skewness of the utilization samples, we use box plots. We enriched the box plots with the value of the median (written in the upper x axis) and with a red line connecting all the medians. Figure 4 and 5 show the resource consumption of a monitor module collecting up to 15 performance indexes and the corresponding utilization of the DB process. The message behind these figures is immediate: if the monitor pipes in several pieces of information from the same process, the collection and storage of additional performance indexes comes at virtually no cost. This is a somewhat logical and expected result, since parsing a vmstat line and building a SQL INSERT statement with a few additional values is cheap. We believe that 15 performance indexes are more than enough to characterize the behavior of most standard hardware and software components; however, since collecting and storing them does not hurt the performance of the acquisition subsystem, in the following experiments the monitors we will do so.

Now, we evaluate the scalability of the acquisition process over an increasing number of monitored nodes. Each monitored node runs exactly one probe; this is a good model of ordinary system monitors that execute a single probe collecting host-related information only. Figure 6 shows the resource consumption of the RAM-DB process over all the experiments. Unfortunately, we could not use more than 64 physical nodes but, judging from the data available, the acquisition process seems to scale pretty well. In order to assess the scalability limits, we performed a least squares regression using parabolas as fitting functions and quadratic residuals. Figure 7 shows that, scaling to larger sizes, the acquisition subsystem will face three bottlenecks: memory (253 hosts), CPU (332 hosts), network (455 hosts). The memory bottleneck is particularly critical for RAM-based DBs, since it will not be possible to store further performance samples. However, even modern, disk based storage engines such as InnoDB and ISAM cache query results aggressively; exhausting the main memory, while not leading to storage errors, would cause a serious performance penalty. If needed, the monitoring infrastructure can be further scaled through parallel, distributed, hierarchical techniques that are out of scope in this paper. Under these conditions ($p = 1$, $t = 1$, $m = 15$), we obtain $N^{mem}_{host}(h = 253) = 3795$, $N^{cpu}_{host}(h = 332) = 4980$ and $N^{net}_{host}(h = 455) = 6825$ samples per second. This means that the considered acquisition subsystem, running at full capacity, is able to monitor a single bunch of (presumably host level) performance indexes at a rate way higher than $N_{host} = 500$ samples/s. We conclude that collecting and storing host-wide performance samples is not a problem. Resource consumption at each monitored host is negligible, as seen previously.

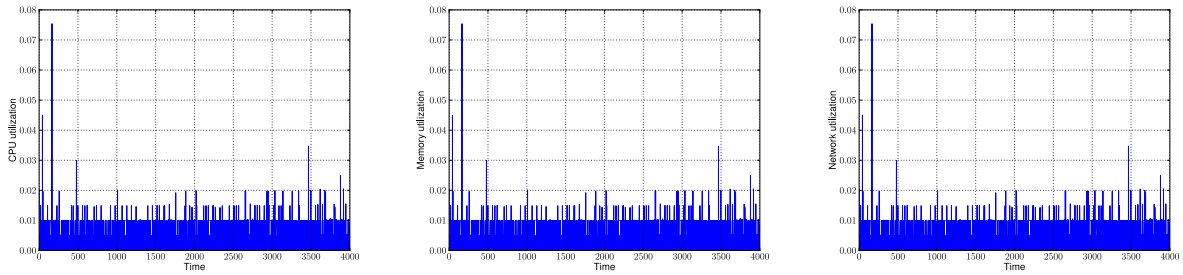
Let us turn to a more realistic scenario, which we believe is a good model of today's data centers that host (and need to control) several (possibly virtualized) services on a single node. We consider a single monitored host executing multiple monitor processes; the performance samples are stored into a single RAM-DB. Figures 8 and 9 report the resource utilization of the monitored host and of the acquisition DB for an increasing number of monitor modules. Both figures confirm that memory tends to be the main bottleneck, followed by the CPU and the network. Starting with 32 probes, the monitors tend to consume over 10% of some system resource (in the order, memory at 32 probes, CPU at 128 probes, network for some value of probes higher than 256). This is a potential limitation in current data centers that may require more than 50 probes per host. Scaling to a higher number of monitors implies adding more physical memory to the host; otherwise, to confine resource consumption within the desired [0.0, 0.1] range, we must limit the maximum number of probes per node to $p = 32$. On the other hand, the RAM-DB does not seem to show any particular signs of congestion.

Let us assess the scalability of this system by adding mon-



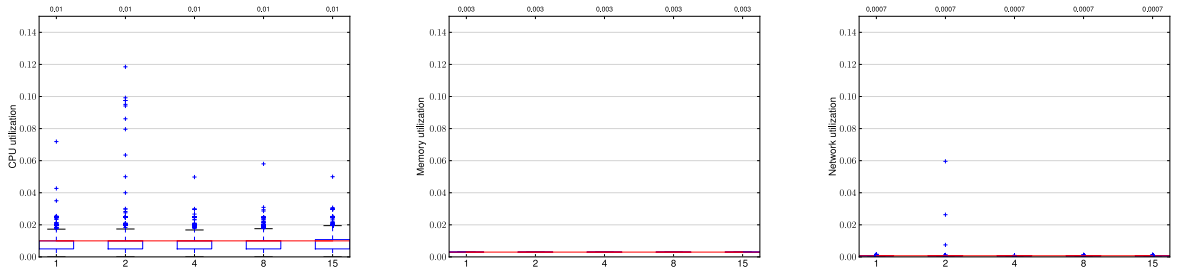
(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 2: Resource consumption of a monitor probe over time ($p = 1$, $m = 1$, $h = 1$)



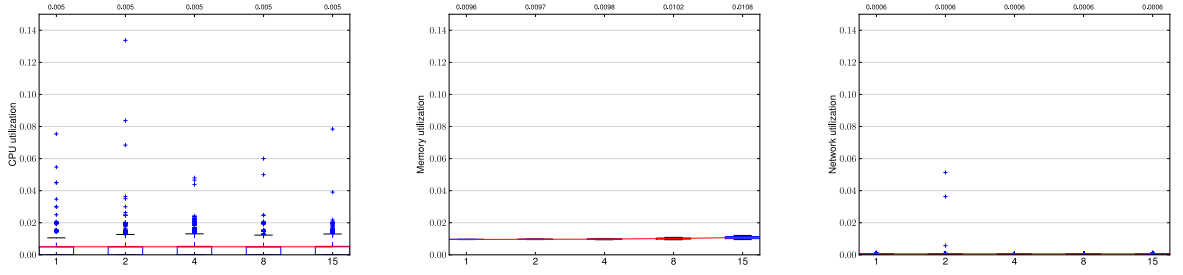
(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 3: Resource consumption of the database process over time ($p = 1$, $m = 1$, $h = 1$)



(a) CPU utilization (b) Memory utilization (c) Network utilization

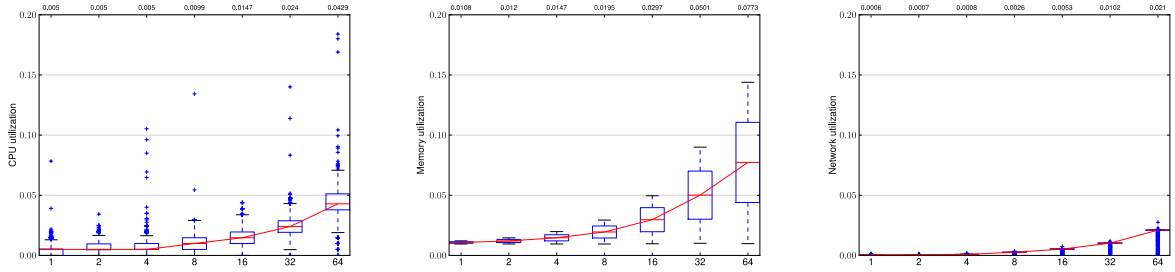
Figure 4: Resource consumption of a monitor probe with an increasing number of performance indexes ($p = 1$, $h = 1$, $m > 1$)



(a) CPU utilization (b) Memory utilization (c) Network utilization

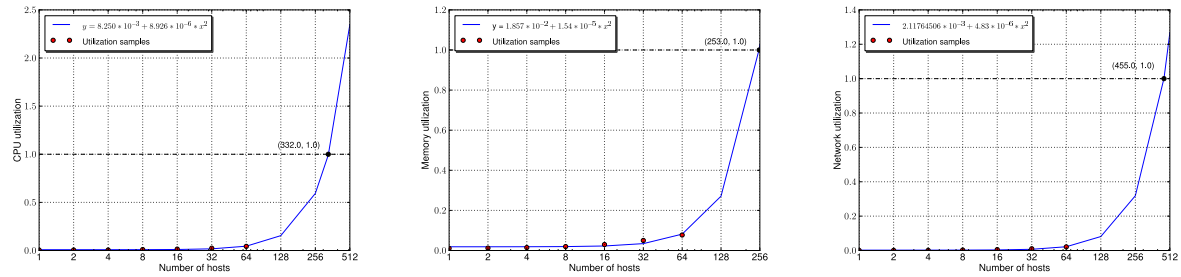
Figure 5: Resource consumption of the database process ($p = 1$, $h = 1$, $m > 1$)

itored hosts with 32 probes each. In Figure 10 we present the resource utilization of the RAM-DB process. We can see that both the main memory and the CPU are exhausted. In these conditions ($p = 32$, $t = 1$, $m = 15$, $h = 16$), the acqui-



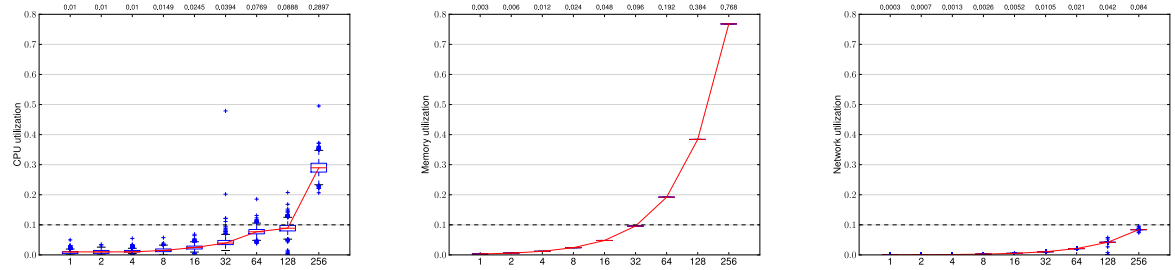
(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 6: Resource consumption of a monitor probe over different hosts ($p = 1$, $h > 1$, $m = 15$)



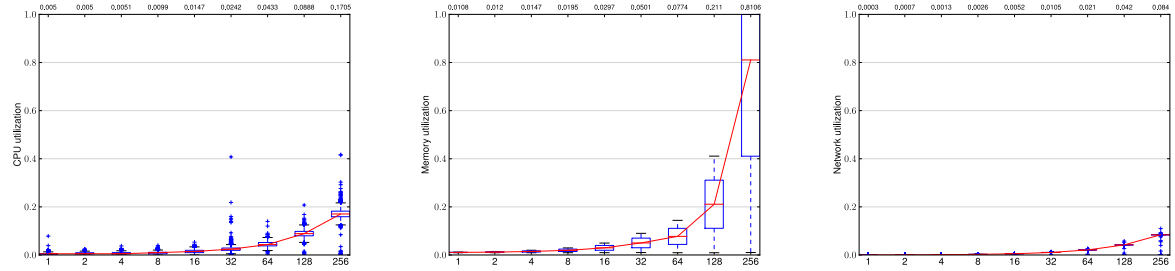
(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 7: Assessing the scalability of the acquisition process ($p = 1$, $h = 1$, $m > 1$)



(a) CPU utilization (b) Memory utilization (c) Network utilization

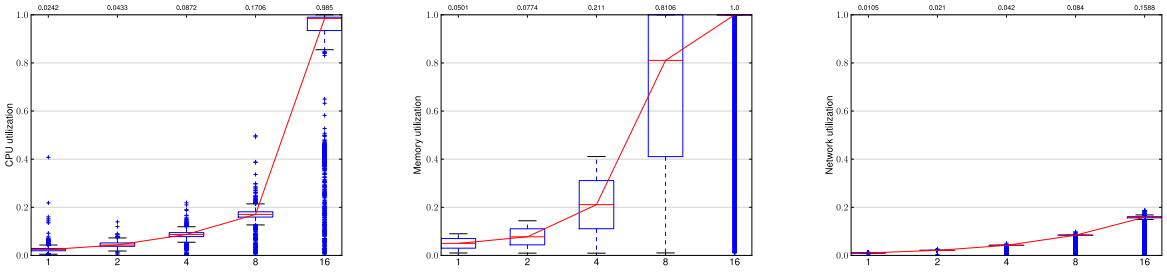
Figure 8: Resource consumption of several monitor probes ($p > 1$, $h = 1$, $m = 15$)



(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 9: Resource consumption of the database process ($p > 1$, $h = 1$, $m = 15$)

sition subsystem collects and stores 7680 samples per second. This number is not even remotely close to $N_{app} = 50000$; scaling to this value would require the deployment of 7 independent acquisition systems.



(a) CPU utilization (b) Memory utilization (c) Network utilization

Figure 10: Resource consumption of the database process for monitor processes on different hosts - 32 probes per host

4. CONCLUSIONS AND FUTURE WORK

As we have seen, a single monitor probe requires a very small amount of resources to gather information and push it to a database. Since the monitor pipes in a whole line of samples, the collection and storage of additional performance indexes come at virtually no costs. Scaling a single probe over some hundreds nodes (up to 455 in our setup) is also fairly straightforward. The monitoring system is able to collect up to 6825 samples per second, which is far more than the desired $N_{host} = 500$.

Adding multiple probes to each monitored host seems problematic in an off-the-shelf hardware/software environment. Monitoring more than 50 applications on a single physical node can run into some resource overhead (more than 0.2 in terms of memory, more than 0.08 in terms of CPU utilization). Right now, there is no other way round other than to limit the number of probes or add more hardware resources to each monitored host. Even worse, in our testbed the monitoring subsystem collapses at 16 hosts, the memory and the CPU of the database being saturated. Under these conditions, the monitoring subsystems collects and stores 7680 samples per second, which is far less than the desired $N_{app} = 50000$. Actually, it is mandatory to use multiple, independent monitoring subsystems (at least 6) to achieve the goal of $N = 50500$ samples per second.

In all of the experiments, the main memory showed up as the main system bottleneck, both in the monitored and in the acquisition database hosts. This bottleneck can be removed by adding more physical memory or by switching to more traditional DBMS that trade memory for speed. The second bottleneck in terms of importance is the CPU of the acquisition host. This bottleneck can be removed by adopting more efficient DBMS (for example, CSQL) or through the deployment of several parallel, indipendent instances in a multi core architecture. In our experiments, the network was never a bottleneck.

Our work can be extended in several ways. For example, the experimental data can be used to build models of interactions for the considered resources. We are currently exploring different fitting techniques and evaluating their statistical significance. The deployment of multiple acquisition databases also opens new interesting issues, such as the evaluation of the speedup and the extraction of a coherent representation of resource state from multiple sources.

5. ACKNOWLEDGMENTS

The authors acknowledge the support of the MIUR-PRIN project AUTOSEC “Autonomic Security”.

6. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[2] M. Basseville and I. Nikiforov. *Detection of Abrupt Changes:Theory and Application*. Prentice-Hall, 1993.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[4] D. Lu, P. Mausel, E. Brondizio, and E. Moran. Change detection techniques. *Int. Journal of Remote Sensing*, 2004.

[5] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[6] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, pages 1–9, New York, NY, USA, 2009. ACM.

[7] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[8] X. Zhu, D. Young, B. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12:45–57, 2009.