Efficient Analysis at Edge

Tatiana Mangels
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
Munich, Germany
tatiana.mangels@siemens.com

Alin Murarasu
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
Munich, Germany
alin.murarasu@siemens.com

Forest Oden
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
Munich, Germany
forest.oden@siemens.com

Alexey Fishkin
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
Munich, Germany
alexey.fishkin@siemens.com

Daniel Becker
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
Munich, Germany
becker.daniel@siemens.com

ABSTRACT

Digitalization changes traditional business models by using digital technologies to improve existing offerings and to create new offerings. Current technological trends such as artificial intelligence, autonomous systems, and predictive maintenance are ideal candidate technologies to enable digitalization use cases. Often, these technologies rely on the availability of large amounts of data and the capability to process these data efficiently. In contrast to consumer markets, industrial products must fulfill higher non-functional requirements such as fast response times, 24/7 availability and stability, real-time processing, safety, or security requirements. As a consequence, processing capabilities – ranging from multicore and manycores to even high end parallel clusters – have to be exploited to achieve necessary performance and stability needs. In this paper, we introduce a *Distributed Multicore Monitoring Framework* (MoMo) which is a reference monitoring solution developed at Siemens Corporate Technology. It can be used to easily build efficient and stable diagnostic solutions which can help to understand the correctness, availability, reliability, and performance of large-scale distributed systems based on live data. Due to its small footprint MoMo can be used to analyze data directly at the data source which, for instance, can significantly reduce the network load. While MoMo's efficiency comes from the usage of multicore processors (CPUs) for running analysis in parallel, its usability is guaranteed by its capability to easily integrate with other monitoring frameworks and its usage of SPL - a domain-specific language which allows user to easily define diagnostic algorithms.

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4899-7/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3053600.3053619

Keywords

Monitoring, data analysis, parallel computing

1. INTRODUCTION

Due to digitalization trend, the number of software intensive systems increases each year. Often such systems consist of different components, each of which is a software intensive system itself. It is difficult to understand and verify such systems' behavior. In this context, many problems only occur for the first time when all system components are running and interacting with each other [8]. It is impractical or even impossible to defend against all possible issues using only static analysis (before deployment).

To cope with such challenges in diagnostic advice scenarios, monitoring is used, i.e. the continuous collection and analysis of data in order to get a up-to-date view on the state of the system. The main and most important requirement to monitoring solutions is to provide correct analysis results in a timely manner. Delayed or wrong results can increase system downtime or even lead to system failures, which otherwise could be avoided.

As digitalization leads to a growing amount of industrial data being collected from lots of sensors placed e.g. inside factories, software systems must either scale vertically or horizontally. In case of vertical scaling, which refers to making one node more powerful, i.e. larger network bandwidth, more memory, more storage capacity, and more processing power must be provided. This causes significant costs and even when possible, it is often not enough to cope with the high data load. Furthermore, various restrictions make this approach impractical. E.g. prior experience with off-shore platforms shows that in those cases network bandwidth often cannot be scaled up. Horizontal scaling in contrast means distribution of the data analysis over the system. It can be done in a data center or close to the source.

Indeed, software systems can always be vertically scaled by adding more processing power at the long term data storage and more bandwidth on the paths to it. However, many customers may still want to deploy and execute the monitoring rules as close to the data sources (i.e., "field" equipment) as possible. This comes with several advantages:

- **Faster response:** Working on local streams of sensor data not only enables "near" real-time scenarios but also accounts for low latencies required for prescriptive analytics, e.g. actuator control.
- **Better utilization:** Often, it is quite difficult or inefficient to send and store all device data to geographically distributed data centers (e.g., in cases when there is a very restricted bandwidth between the platform and the on-shore monitoring center). So, the analysis of "field" equipment using "local" computers provide a better utilization of available computational resources.
- **Security and privacy:** There is a possibility for a lot of security leaks when data from "field" devices is transported via internet and then stored on external servers. So, keeping data either on the "field" device or at least as close as possible to ones without leaving the local network makes security and privacy easier to achieve.

In this paper, we present an efficient and stable embedded monitoring component (i.e., *Distributed Multi-core Monitoring Framework* (MoMo)). Please note that we exclude explicitly any security, time synchronization, and software lifecycle discussion in this paper. MoMo is a monitoring solution developed at Siemens Corporate Technology (CT). It is used to easily build efficient diagnostic solutions, which can understand the correctness, availability, reliability, and performance of large-scale distributed systems based on live data. A central goal of MoMo is to support the building of diagnostic solutions in which no relevant event is missed because of an overloaded diagnostic system. MoMo follows a push-data model in which data is sent from external data sources to the MoMo based diagnostic solution. Thus, MoMo can be easily used to connect to publish-subscribe based protocols in which the diagnostic solution is a subscriber.

MoMo's efficiency comes from the usage of multicore processors (CPUs) for running analysis in parallel. To provide the highest performance, the computation in MoMo is done entirely in-memory, i.e. using only the RAM and the CPU cache instead of a disk-based database. This is advantageous compared to other approaches, such as saving data on disk prior to analysis, which would result in a lot of overhead caused by reading and writing to disk [9][7].

Parallel programming is an error-prone process due to challenges such as waiting times, deadlocks, and race conditions. Nevertheless, MoMo is still able to offer high usability, by hiding parallelism from the analysis developer. The approach taken by MoMo to abstract parallelization away from the user relies on data-flow programming concepts. In this approach, the developer only needs to be concerned with creating a graph of operations, which represents fine-grained analyses. MoMo extracts data and function parallelism from such a graph, which for the end user translates to a fast execution on her multicore hardware.

Due to its small memory footprint and limited processing overheads, MoMo can be executed on embedded devices and hence can be used to analyze data directly at the data source for horizontal scaling. To provide even better usability, we integrated a *Signal Processing Language* (SPL) into MoMo.

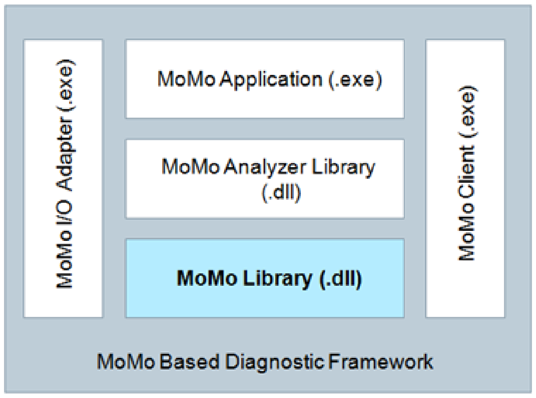

Figure 1: The MoMo library.

SPL is a domain specific language developed at Siemens CT, which allows to define data analysis based on complex event processing in a straightforward manner.

This paper is organized as follows. Section 2 presents an overview of the MoMo architecture. In Section 3 we discuss how analysis can be composed in MoMo. In Section 4 we explore application of MoMo on embedded devices. We discuss lessons learned in Section 5, related work in Section 6 and present a summary in Section 7.

2. ARCHITECTURE

2.1 Objectives of MoMo

MoMo's main design objective is to allow users to easily and efficiently diagnose problems related to the availability, reliability, and performance of large-scale distributed systems in which many software components communicate over a network. However, MoMo cannot accomplish this by itself; in fact, MoMo is a library that is used as a foundation for a complete diagnostic framework. We can see the minimal set of components required by a framework in Figure 1:

- The MoMo adapter is responsible for collecting data from a specific source.
- The MoMo library executes the analysis in parallel.
- The MoMo analyzer library incorporates the diagnostic logic.
- The MoMo application is used to configure and trigger the data collection and analysis; it uses the MoMo library and MoMo analyzer library.
- The MoMo client receives analysis results.

With regard to functionality, MoMo (library) fulfills the following key roles:

1. MoMo collects live data according to a push-model in which the analyzed system sends data without being asked for it.
2. MoMo processes the collected data in parallel by executing a user configured analyzer graph on multicore processors.

3. MoMo distributes analysis results to all the registered client applications, which in turn can use the results in many different ways, e.g. for visualization or notification.

MoMo also contains a set of general-purpose analysis methods, which provide functionality for time series forecasting and calculating statistical parameters. On top of these methods, users can create their own specific analyzer classes, which indicate what exactly is broken or predict failures, thus considerably reducing the costs for fixing problems.

MoMo's focus is placed on the following non-functional requirements:

- Efficiency, meaning that data collection and analysis in MoMo are fast operations.
- Scalability, meaning that multicore processors are harnessed in order to increase the throughput in terms of number of analyses per second.
- Flexibility, meaning that many types of systems can be analyzed using MoMo at the cost of defining the necessary analyzers and adapters.
- Lightweight, meaning that the overhead caused by MoMo on the analyzed system is minimal because of the use of UDP for data collection.

2.2 Data flow in MoMo

An essential concept in MoMo is that MoMo manages the life cycle of analyzer instances. An analyzer is a class that is able (1) to process data collected by MoMo and (2) to return results represented as MoMo messages. MoMo is able to handle many such analyzer instances concurrently. Besides that, analyzer instances can also be wired to each other such that results are passed from one analyzer instance to the next. The key point here is that analyses performed using such objects are executed in parallel whenever MoMo sees that they are independent.

Besides analyzers, MoMo also incorporates:

- A delay container, which is used to sort messages based on their time stamps before they are forwarded to analysis. This is important because communication between MoMo and the data sources is based on UDP, i.e. messages may be received in the wrong order.
- A message dispatcher, whose responsibility is to add messages to the queues of the interested analyzer instances.
- A scheduler, whose responsibility is to create tasks for parallel execution given a set of analyzer instances with their respective messages of interest. A task can be roughly defined as a pair (analyzer instance, message to analyze).
- A worker thread pool, which is used to avoid oversubscribing the available cores with too many threads.
- Information on a set of clients interested in analyzer results. In MoMo, a client is a remote process identified by IP address and UDP port to which analysis results are sent, e.g. for visualization or logging. Output adapters, which allow us to communicate to a large variety of systems, can be built as MoMo clients. They realize the translation from the language of MoMo to the specific language of the system that we analyze.

Figure 2 shows the data flow diagram of MoMo in which messages travel from different points in a distributed system to MoMo, inside MoMo, and finally, outside MoMo. Within MoMo, messages are placed in the delay container where they wait for a configured amount of time and subsequently, they are sorted according to their time stamps. From there, analyzer instances are asked if they are interested in the message coming out of the delay container – if yes, the message is pushed (as reference) to the queue of interested analyzer instance.

Messages assigned to an analyzer instance are analyzed serially (first in, first out) whereas messages assigned to different analyzers can be analyzed in parallel. Involving an analyzer instance in two parallel tasks can result in race conditions. This explains why serial execution is necessary when processing the same message queue. Analyzing a message is realized by creating a task and submitting it to the thread pool for execution. During analysis, new messages can be created, which, as depicted in the figure, can travel outside MoMo or inside MoMo to other analyzer instances.

MoMo can be deployed in any system which publishes data and offers means to read/subscribe to this data. However, since MoMo cannot understand the messages of an arbitrary distributed system, an adapter implementation is needed. Figure 2 shows input and (if needed) output adapters which translate the messages of the analyzed system to MoMo format and vice versa. In case the system messages contain numerical sensor data, analysis defined by means of SPL (Subsection 3.4) can be directly applied to the translated messages. However, there is also a possibility to define custom analyzers for custom data.

3. ANALYSIS DEFINITION

In this section, we will consider how an analyzer instance's messages can be used for inter-analyzer communication to "wire" the analyzers.

3.1 Independent Analyzer Instances

MoMo is a Qt/C++ framework. To define an analyzer, an abstract class `MomoAnalyzer` must be extended. In particular, the concrete implementations for pure virtual functions `IsInterestedIn()` and `Analyze()` must be provided. The function `IsInterestedIn()` is a kind of a filter, it defines, which data must be analyzed by this analyzer. The function `Analyze()` defines the actual analysis.

MoMo's performance is greatly influenced by the number of analyzer instances that are managed by the Momo class instance, i.e. the more, the better, as this means that there is more concurrent work to do and more parallelism can be exploited. When analyzer instances do not communicate to each other, we refer to them as *independent analyzer instances*. In such cases, to make sure that CPU cores are fully utilized during analysis using MoMo, we need to make sure that the number of independent analyzer instances is at least as big as the number of cores. We recommend that the number of analyzer instances is in fact several times bigger than the number of cores, e.g. 10s or even 100s of analyzer instances. Having multiple independent analyzer instances of the same class results in *data parallelism* [3], i.e. the parallel execution of one algorithm on different data.

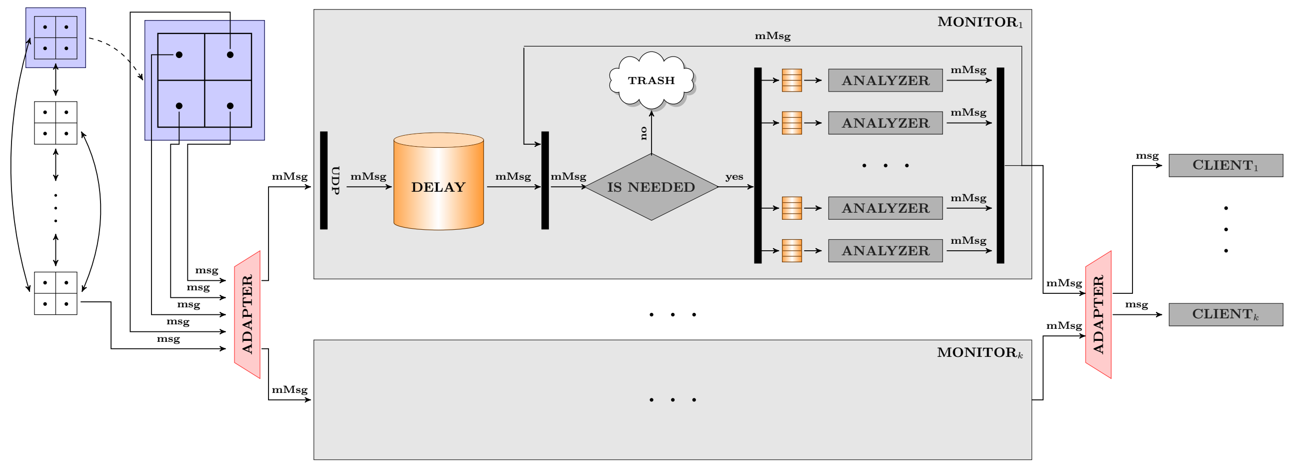

Figure 2: Data flow in MoMo.

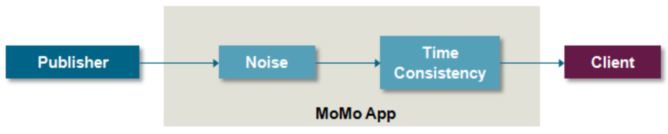

Figure 3: A noise-time-consistency pipeline.

3.2 Pipelines

MoMo goes beyond the case of independent analyzer instances. To define more complex hierarchical analysis, analyzer instances can send data to each other by means of MoMo messages. By setting the *recursive* flag, the message with the data from analyzer is fed back into the MoMo data flow. A first use case is the one of a pipeline of analyzer instances like the one shown in Figure 3.

The first stage of the pipeline checks whether there is noise in the sequence of incoming values. We encapsulate the logic for noise detection in a `NoiseAnalyzer` class, which is derived from `MomoAnalyzer`.

The second stage of the pipeline checks whether alerts returned by the first stage are time-consistent, i.e. there are multiple noise alerts generated within a time window of a certain size. The reason to introduce the second stage is to reduce the number of false positives generated by the first stage alone. Simply put, if a noise alert comes in isolation, it is ignored (it is probably a false positive); if many time-near values violate the check, the alarm is said to be consistent across time, thus real. In the latter case, the alarm is sent to the registered MoMo clients.

In order to realize such a pipeline with MoMo a user must (1) create an instance of an analyzer class which detects noise and analyzes only external messages and (2) create an instance of an analyzer which checks time consistency and analyzes only messages from the noise analyzer. The possible parallel execution will be exploited by MoMo automatically.

3.3 Graphs

Even though pipelines are powerful, they are not enough to cover important analysis use cases. Complex analyses often require that they are expressed as graphs composed of more basic analyses. Let us consider a use case which objective is to discover a broken sensor in order to replace it. The analysis setup for it is depicted in Figure 4.

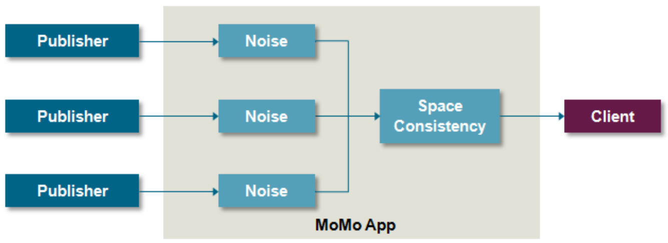

Figure 4: A noise-space-consistency graph.

The graph has three noise nodes. Each is responsible for detecting the noise in the data originating from a sensor – publisher in the graph. The sensors here are redundant, which helps to decouple the failure of a sensor from the failure of the machine where it is placed. In other words, even if a sensor is broken, the other redundant sensors' data can be used to detect a problem affecting the machine.

The output of each noise node is gathered into a space consistency node, which checks if an alert is returned in a consistent manner by all the noise nodes. If the outputs of the noise nodes differ, we detect a sensor divergence, which is an indicator for a broken sensor.

In order to realize such a graph with MoMo a user must (1) create three instances of an analyzer class which detects noise and analyzes only external messages and (2) create an instance of an analyzer which checks space consistency and analyzes only messages from the noise analyzers. The possible parallel execution will be exploited by MoMo automatically.

3.4 SPL Integration

The MoMo framework is flexible and allows users to implement and deploy analyzers written in Qt/C++. However, to do this, a user needs expertise in programming. To increase usability of the framework we integrated with MoMo the *Signal Processing Language* (SPL). SPL is a domain-specific language, developed at Siemens CT, which allows users to easily define diagnosis algorithms. The listing below shows a simplified grammar that needs to be followed when defining SPL algorithms:

```
signal( "pattern text" ) =
$PrName1: truth(Expressions over events/signals)
: temporal constraints over start/end of $PrName1
and
$PrName2: truth(Expressions over events/signals)
: temporal constraints over start/end of
  $PrName1 and $PrName2
and
...
```

In particular, SPL allows to define statements over events or signals in order to infer patterns. Let us consider the statement above: As an input we get data from e.g. signals. We can define a logical expression over this data. As soon as this logical expression is evaluated to true, we say that the process `$PrName1` starts. As soon as this logical expression is evaluated to false, we say that the process `$PrName1` ends. Furthermore, we can also define temporal constraints over start and end of this process. Similarly, we can define process `$PrName2` with the difference that now we are allowed to define temporal constraints over start and end of `$PrName1` and `$PrName2`. To clarify this, let us consider the following example:

```
signal("Turbine Start-up") =
$t1: truth(#"S" < 200):duration(>=1m)
and
$t2: truth(#"S">9500):$t2:start>=$t1:end:
     duration(>=5m);
```

In this example, we want to detect a start-up of a turbine based on its rotor speed, here represented by signal `S`. The SPL statement says that if the rotor speed remains under 200 for longer than one minute and after this, remains over 9500 for longer than five minutes, the turbine start-up is detected.

This formula is parsed and translated into a graph of MoMo analyzers, which can then be directly applied to monitor and analyze the data.

4. APPLICATION ON EMBEDDED DEVICES

Due to its small footprint MoMo's based solutions can be executed on embedded devices and hence can be used to analyze data directly at source and scale the monitoring system horizontally. In fact, we have shown that it is possible to run a MoMo solution on Raspberry Pi 3. The size of the exemplary solution we used in our study with seven analyzer instances is under 6MB. Simulated sensor data was sent to MoMo running on Raspberry Pi 3 over LAN.

The solution included three Anomaly Analyzers, three Time Consistency Analyzers and one Spatial Consistency Analyzer (the graph of this solution is depicted in Figure 5). The Anomaly Analyzers take input from the sensors and use past and future data (this is achieved by collecting a small buffer of data from the sensor before performing the analysis) to determine if the data is anomalous. If it is, an alert is sent to the next set of analyzers, the Time Consistency Analyzers.

The Time Consistency Analyzers look into the past over a user defined time window and determine if the number

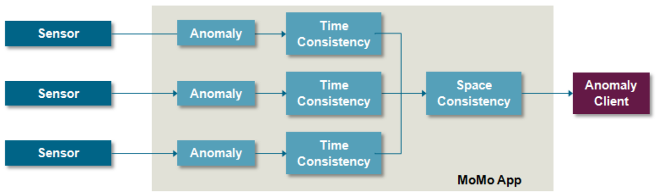

Figure 5: Anomaly graph.

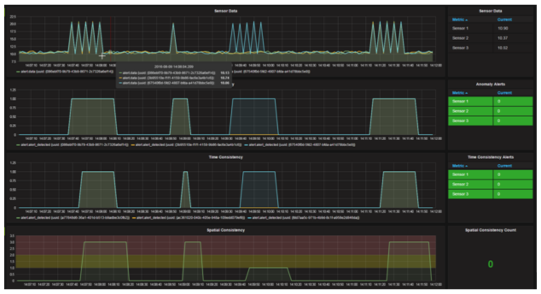

Figure 6: Visualization of analysis results.

of anomaly alerts is more than a user defined threshold. If the threshold was exceeded, another alert is sent to the Spatial Consistency Analyzer. The Spatial Consistency Analyzer gets alerts from all Time Consistency Analyzer instances and observes how many analyzers out of the total had a Time Consistent alert in a user defined time window. The Spatial Consistency Analyzer will, again, trigger an alert if the count is above a user-defined threshold. At this point, the analysis is done and all data is sent to the MoMo Anomaly Client which is also running on the Raspberry Pi. The MoMo Anomaly Client gets the results from the MoMo Analyzers and sends them to a Windows PC, where the data is displayed using Grafana (Figure 6).

5. LESSONS LEARNED

Typical scenarios for application of the MoMo framework are diagnosis of turbines, railway field elements, transformers, devices on oil platforms such as valves vanes. Usually, sensor data (temperature, on/off state, voltages, currents, etc.) is pushed via e.g. a publish-subscribe protocol from these devices and MoMo performs listener based diagnosis, which means it is not intrusive with regard to the analyzed systems. For use cases where MoMo was applied, minimal impact on the monitored system was of great importance, especially when dealing with safety-critical or safety-related systems.

In current typical monitoring scenarios, data sampling is approximately 1 sample/s. When dealing with high-frequency data (100, 1000 samples/s), a sampling rate of 1 sample/s means loss of almost 100% of the data, which makes meaningful analysis of this data infeasible. Increasing the sampling rate to send the data to a server for analysis will explode the network traffic and even negatively impact the functioning of the monitored system. One possible solution in this case is to perform analysis of the data at source while keeping the sampling rate high. Being portable on embedded devices, MoMo supports well this scenario.

No matter the context, providing analysis as fast as possible is a frequently occurring requirement. Delaying the analysis results can result in failures, even broken devices, which translates to high maintenance costs. Thus, performance of data analysis is of high importance. MoMo addresses this requirement through parallelization of analysis execution. Typical diagnosis scenarios expose massive data parallelism which is effectively exploited by MoMo. However, it is important to ensure that there are enough analyzer instances. An insufficient number of analyzer instances leads to insufficient concurrency, which in turn translates to idle CPU cores, resulting in poor performance. Furthermore, the analyzer instances must not be too fine grained. Having too fine grained analysis methods means that the overhead of scheduling will dominate the total execution time, resulting in poor performance. In our studies, we achieved parallel efficiency of ∼82% when calculating on two threads on a dual-core machine and of ∼63% when calculating on three threads with activated hyper-threading. Nevertheless, the parallel efficiency which can be achieved highly depends on the executed scenario.

6. RELATED WORK

There is a significant number of monitoring solutions available on the market [1], [5], [2], [4], [6]. Most of these are complete solutions covering a wide range of functionalities including data collection based on pull / push / agent, threshold based analysis, storage using SQL or NoSQL, visualization using Web UIs, and alerting. Monitoring solutions, e.g. Nagios[1], Icinga[5], Zabbix[2] are typically based on plugins, which allow to extend the functionality, e.g. the set of metrics that can be measured.

Compared to the monitoring solutions from above, MoMo is more focused as it mainly provides the means to easily express chained analyses and run everything efficiently on multicore processors. Therefore, MoMo should not be regarded as a replacement for Nagios or its peers but rather as a complementary framework used whenever high performance is required, especially in scenarios dealing with high frequency data. In this context, MoMo can be used as a preprocessor for e.g. Nagios in such a way that data generated at high rates is analyzed at source (or close to it), thus avoiding situations in which Nagios is overloaded by too many incoming messages.

Moreover, the usage of plugins as it is normally done today in monitoring solutions comes with drawbacks: (i) Plugins used for data analysis are interpreted scripts, i.e. their execution is slow and (ii) plugins are also stateless, resulting in the necessity that states are saved and restored for every message to analyze, which makes analysis slow. In contrast, MoMo analyzer instances are stateful (no need to persist state), which translates to higher performance.

Parallelization within typical monitoring solutions is achieved by creating a new process whenever a check needs to be executed, which leads to performance loss because of the overhead associated with process management vs. the actual computation, e.g. threshold check. In this regard, MoMo uses the thread pool pattern, which means that no time is wasted with starting and stopping processes and even with context switches.

With regard to plugin inter-communication, it is not explicitly covered in most monitoring solutions, which limits the possibilities for analysis composition. In MoMo, defining graphs of analyzers, which communicate with each other, is one of the main features.

Summarizing, MoMo can provide efficient preprocessing at source, which is not possible with the above mentioned solutions. On the other side, MoMo has less functionality (e.g. visualization, alerting mechanism, persistency) which is realized in those solutions. As stated before, we envision that these solutions are used together: MoMo to address the horizontal scaling and Nagios, Zabbix and others to visualize, alert etc.

7. SUMMARY

The digitalization trend confronts monitoring solutions with new challenges. In order to avoid failures and reduce costs, a growing amount of industrial data must be analyzed in time. The approach of horizontal scaling not only reduces network load and saves storage capacity, but also enables analysis of data which is not possible with vertical scaling due to physical limitations, e.g. in case of off-shore platforms. Due to its small footprint MoMo's based monitoring solutions can be executed on embedded devices and hence can be used to analyze data directly at source and scale the monitoring system horizontally. Furthermore, MoMo provides the means for hierarchical analysis; complex analyzer communication patterns can be easily created. Regarding efficiency MoMo is designed to execute analysis fast in order to increase throughput and to be able to cope with growing amount of data.

8. REFERENCES

[1] N. Enterprises. *Nagios - The Industry Standard In IT Infrastructure Monitoring*, January 2017. https://www.nagios.org/.

[2] Z. LLC. *Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution*, January 2017. www.zabbix.com.

[3] T. G. Mattson, B. A. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley, Boston, 2005.

[4] T. G. Project. *Ganglia Monitoring System*, January 2017. http://ganglia.info/.

[5] T. I. Project. *Icinga - Open Source Monitoring*, January 2017. https://www.icinga.com/.

[6] T. M. O. Project. *Munin*, November 2016. http://munin-monitoring.org/.

[7] N. Savage. The power of memory. *Commun. ACM*, 57(9):15–17, Sept. 2014.

[8] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel. Reminds: A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software*, 2015.

[9] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-Memory big data management and processing: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 27(7):1920–1948, July 2015.