Experimenting with Complex Event Processing for Large Scale Internet Services Monitoring

Research Paper

Stephan Grell, Olivier Nano

Microsoft, Ritter Strasse 23, Aachen, 52072, Germany
Tel: +49 241 99784 533, Fax: +49 241 99784 77
{stgrell, onano}@microsoft.com

Abstract: In this paper we discuss an experiment to monitor large scale internet services. The monitoring system is implemented as a Complex Event Processing system to enable better scaling of the infrastructure. We describe the application of the monitoring in two scenarios and give a brief overview of the benefits of the approach and the remaining challenges. The discussion is focused on language expressiveness, debuggability, root cause analysis, and maintaining a stable system.

Keywords: CEP, debugging, load shedding

1 Introduction

Large scale internet services are a class of applications which impose strong requirements on their management infrastructure and on their execution environment. Large scale internet services are services deployed in Data Center environments and running on a large number of machines (from hundreds of machines to thousands of machines). Due to the scale of these services, the management and monitoring system needs to scale accordingly.

Monitoring of large scale internet services can be performed in many different ways depending on the expected monitoring results. This experiment is based on a Complex Event Processing (CEP) system which process the events in near real-time as long as the resource consumption limits are kept. We explore two monitoring scenarios: syntactic transactions generated by a watchdog and business events generated by the services themselves.

In today's monitoring setups there are a lot of tradeoffs between scalability and the speed at which results are computed. Our goal is to evaluate the benefits of implementing, deploying and managing a monitoring system through a CEP system in very large scale setup. The motivation to use a CEP system is its ability to do fast in-memory processing of events (filtering, grouping and aggregating) which enables to do real time analysis. The following sections discuss early results and share some of the challenges which we believe need to be addressed to ease the adoption of CEP as a scalable monitoring system.

In the next section we detail the two scenarios for monitoring large scale internet services. In the third section we present the CEP system we used for our experiments. In the fourth section we describe challenges around the expressiveness of query languages. In the fifth section we discuss the challenges around debugabillity and root cause analysis. In the sixth section we describe the challenges around environment stability and load shedding. At the end we conclude and present potential future work.

2 Monitoring large scale internet services

In this paper we explore two different scenarios to monitor Service Level Agreements (SLA) of large scale internet services. In the first scenario we monitor the services' availability through syntactic transactions. In the second we monitor business events generated by user interactions upon the services.

2.1 Syntactic transactions monitoring

In the first scenario services are monitored via syntactic transactions generated by a watch dog invoking the services. The results of the syntactic transactions are made available via a Pub/Sub system to different listeners. One of them is a SLA monitor which aggregates all the results according to different rules depending on the state of the services. The SLA monitor publishes the results of the aggregations which are emailed to the IT operators. The motivation for syntactic transactions is to have a high degree of control on the load of the monitoring system. Syntactic transactions are scheduled at regular interval and the frequency can be adjusted. For this scenario, a single machine is sufficient to execute the SLA monitoring because the volume of syntactic transactions results is low enough to be transferred to the central SLA monitor. We used a CEP system for this scenario because of its flexibility in updating the standing queries based on new demands as well as its ease to add new queries to monitor new services.

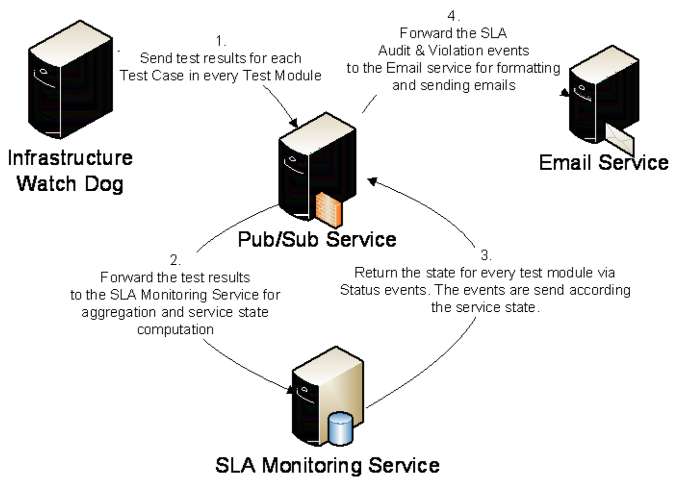
1.
Send test results for each
Test Case in every Test Module
4.
Forward the SLA
Audit & Violation events
to the Email service for formatting
and sending emails
Infrastructure
Watch Dog
Email Service
Pub/Sub Service
2.
Forward the test results
to the SLA Monitoring Service for
aggregation and service state
computation
3.
Return the state for every test module via
Status events. The events are send according
the service state.
SLA Monitoring Service

Fig. 1: Outlines the communication between the services and the services involved

The SLA monitor uses the following set of rules to aggregate the results of the syntactic transactions. Every service has its own set of tests (test cases) which are combined in a test module. Every test module has one SLA associated. The state of the test module is determined by the worst state of its test cases. A test case can have the following states: up, low, or down. The state is computed based on the success percentage of the test runs: above 95% the state is up, between 5% and 95% the state is low, and below 5% the state is down.

The SLA computes the states of a Test Module every 10 min. If the Test Module undergoes a state change, a state change event is published. Otherwise the Test Module stays in its current state and state stay events are published every 30 min for down, 60 min for low, and 6 hours for up. Every published event contains the entire list of the test case results for the test module at that time with the aggregated values over the reporting period.

In addition to monitoring the Test Module states, the SLA monitor is also monitoring its own infrastructure. The SLA also raises an alarm if it did not receive any test events for a period of 5 min. No events for 5 minutes are an indication that either the Pub/Sub is down or the watch dog.

To support the debugging of the services, the exceptions that the watch dog creates in case of failures will be reported with the state of the test case. Every test case report contains the latest exception. We decided against a list of all exception because that list might get long over the course of up to 6 hours of monitoring.

This scenario brings the requirement of maintaining a complex standing query in an environment with limited human access. The writing was complicated enough that multiple iterations were needed to get the state model right and to understand the issues. To get this scenario right, we need a way to understand what the CEP system is doing from log files and being able to analyze them efficiently from remote.

2.2 Business events monitoring

In this second scenario the requirements are to monitor business events generated by the services. Business events are events which represent transactions happening in the service logic as opposed to lower level events (like heartbeat, disk operations, etc). The business events generated are for example login/logout of users, search operation triggers, etc. As these events are user driven, they are generated as users hit the services. Because of the events' frequency and the number of machines running the services in the cluster, it is not feasible, nor performing, to move all the generated events to a central machine for processing.

For this scenario we have setup a distributed version of our SLA monitoring system. Every machine that is running an instance of a service is also running an instance of our SLA monitoring system which receives and aggregates the business events generated by the local service over short period of times (around 10 minutes). A central instance of the monitoring system then receives, through the Pub/Sub, all the aggregated events and performs a per user aggregation over the events from all services. The final aggregation results are made available through the Pub/Sub for other services, such as billing or user satisfaction monitoring.

This scenario puts requirements on the resource usage by the SLA monitoring infrastructure. As the SLA monitors run on the same machines as the services themselves we need to guarantee that they will execute within certain resources boundaries (such as CPU, memory, and network bandwidth).

This scenario also brings requirements on the quality of the final aggregated data. Depending on the technique used to maintain the service utilization, certain business events can be discarded to ensure the overall performance of the SLA monitoring system. However, because these business events and their aggregate are used by other systems such as a billing system, it is important to ensure that effect of the discarded data on the overall results is controlled.

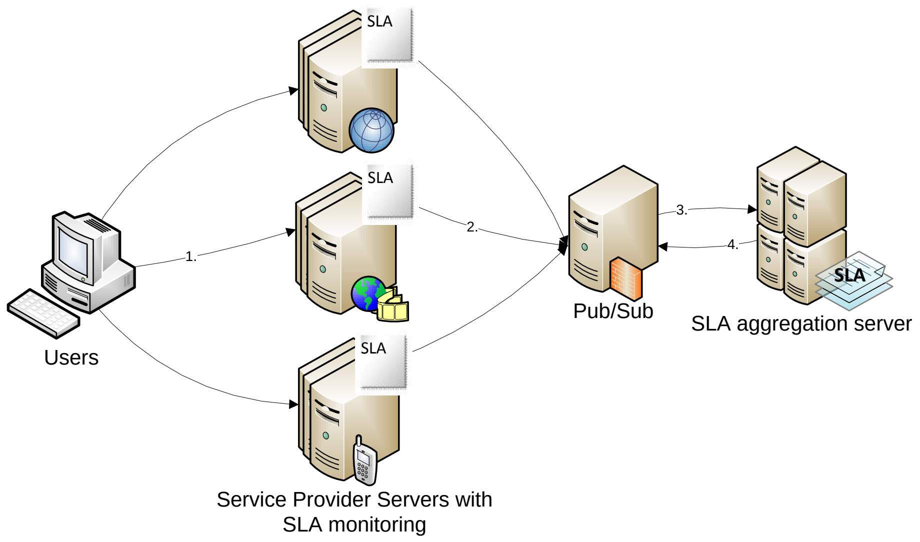

Fig. 2: Showing the distributed cloud service scenario with the user interactions

3 The SLA monitoring system

The SLA monitoring system (1) we implemented for the experiments is generic and comparable in term of operators and queries to other CEP systems (2) (3). It consists of the following parts (also shown in **Fig. 3**):

- SLA language for writing SLA documents (similar to standing queries)
- SLA editor to create the SLA document as box diagrams
- SLA runtime to monitor an SLA document, single node or distributed
- SLA dashboard for monitoring the service performance against the SLA and to analyze SLA runtime log files.

The language contains four main building blocks: Input Adapters, Probes, Computations and Audits/ Violations. Probes take the data from the Input Adapters and extract that data into the internal data format. They also filter the incoming data, so that only the information needed is extracted. Computations contain a set of operators. The operators work on streams or the output of other operators. Computations get their inputs from the Probes or other Computations. Audits and Violations define data validations and trigger an output event when data is out of range. Data computed by Audits and Violations is publish on the Pub/Sub system and made available for other systems to consume.

In addition the language contains supporting concepts for time based triggering, logging, and assigning the elements to specific runtimes. The assignment of elements to the runtimes can be updated for some elements during the runtime. The elements that reside in different runtimes are connected by streams which span machines.

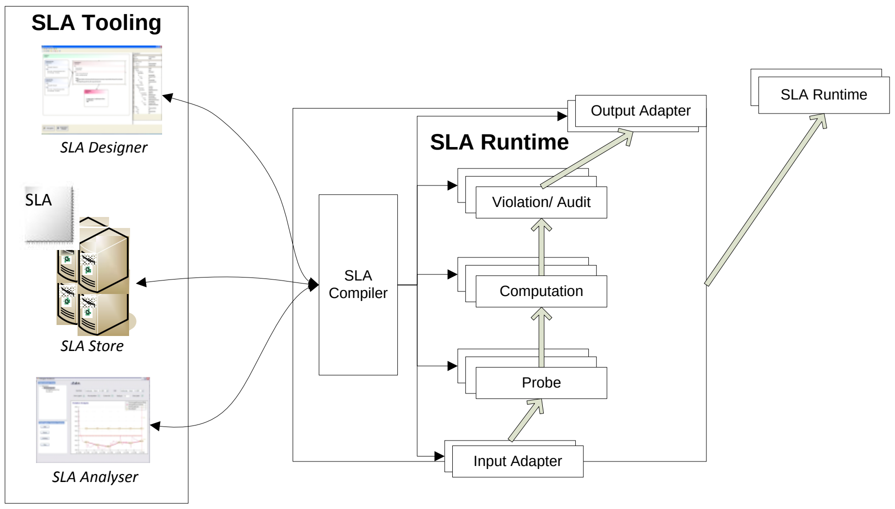

Fig. 3: Overview of the SLA monitoring system with the different tools and the main concepts inside the language

4 Language expressiveness

SLA monitoring brings interesting challenges when implemented as a CEP system. Certain SLA requirements are difficult to express as queries and would benefit from automata support description for example. Next, we list some examples of issues expressing SLA with standard query languages.

SLA monitoring usually works within fixed windows as the state of a system is usually defined over a fixed period of time. This means that hopping windows need to be simulated on top of the sliding windows semantic of the underlying CEP system.

A SLA monitoring system needs to determine the state of services and send out information about it. In our experiments state changes are of high importance and the most expressive way to implement them was to build a state machine with standing queries. In our first scenario described in section 2, we built 3 sub-queries for the analysis of the current state and nine additional sub-queries for state transitions. The nine sub-queries include a state stability transition.

Timing the publication of violations and audit reports is also challenging. For regular audit reports they need to be published at precise time of the day and violations reports depend on the state of the monitored service and their frequency depends on past violation reports.

For the CEP usage in an SLA monitoring system we believe that adding domain specific constructs that provide SLA patterns would ease the description of SLA.

These patterns should simplify the pattern recognition inside streams, the dealings with time in the processing as well as the state computation.

5 Offline / root cause analysis

The development of the different standing queries showed that it is easy to introduce errors in standing queries. The generation of the traces and past events analysis helps not only in debugging the standing query during the development phase but also later in production when an unexpected issue occurs. The analysis of past events has been research topic for some time now. It is labeled as "Time travel" or "Replay" (1).

The requirements of our environment exceed the ones of the Borealis project. Our monitoring system runs on servers which cannot be directly accessed. All we can do is accessing log files to understand what the monitoring service has been doing in the past. Another challenge is the limit on the file size. The monitoring system cannot log every incoming event package that it has seen since the beginning of time. And lastly, the obvious challenge of limiting the performance impact on the processing is of high importance.

In addition to our internal motivation as the developer of the standing queries for the monitoring system, we have an additional request by our "users" to be able to drill down into interesting violation events from the standing query. The tools and requirements for this functionality are very similar to the debugging one. The drill down case is a bit more limited, as the systems needs only to hold the events and the internal processing steps which resulted into a violation. All the other events are considered noise which distract from the violation analysis.

To anticipate these requirements the monitoring runtime includes a logging framework that allows logging every result of a "block" in the standing query. These results contain besides the time, a unique identifier, and the data fields also a list of events / results which were consumed by this result. The log messages are made available via a set of adapters, including a file writer as well as a network connector. The network connector allows a tool to directly connect the log messages and display the processing progress. The file writer stores all messages in a rotation log file on disc. The file has a maximum size associated.

A UI tool allows us to display the log messages with the standing query. The display enables the user to replay all log messages for the processing or only the ones for certain sub-path inside the standing query. In addition one can also look at one processing block and see all the log messages for that one with its definition.

The work so far allows us to debug a running standing query and we can understand what happened inside the monitoring runtime in case of a crash. One could also use this data for recovery in case of a failure. However, this does not satisfy all the demands we started out with. This tool does not fully support the drill down requirements. The log framework does not allow selecting which data can be purged from the log file so that only the data of the violation stays. If the log file size is big enough and the drill down request is down quickly after the violation is issued, one has a good chance that the data that lead to the violation is still available. The current approach has also a big impact on the overall performance when the logging is enabled. A second mode that supports the drill down but not the debugging is thought of but not experimented with. One could store the history, similar to the Borealis approach, in memory and only write the log file when a violation happened. This would result in a less frequent interruption and the writing could be scheduled such that it is does not interfere with the processing.

After a couple sessions of debugging and analyzing the log files, the desire for a CEP debugging environment similar to that of programming languages arose. This environment should be capable of working with recorded traces from a test run as well as simulating the execution of the standing queries with a range of inputs. The simulation environment would be helpful as a test environment for the written standing query and allows testing the corner cases and the different paths.

6 Reliable Infrastructure

Large scale internet services require by definition a very stable execution environment and as they scale the management infrastructure needs to scale as well. In the second scenario that we presented in section 2, the requirement is to monitor business events generated by the services. The number of events generated by the services is high and there are a lot of service instances. Therefore a first aggregation of the business events needs to be done on the machines running the services instances. Due to the fact that the number of business events generated depends on user actions on the services, a higher service load results in a high event count which in turn impacts the computation of the first level of aggregation on the services' machines.

This creates a potential problem for the stability of the services' execution environments. Based on the very dynamic nature of the services and aggregations execution under heavy load it is very difficult to understand under which exact conditions the system is at risk.

We are experimenting with a combination of two techniques to limit the risk of system overload and system crash during execution. The first technique applies static capacity management to the standing queries to understand what is potentially executable without too much risk. The second technique is events shedding to enable aggregations to empty its pipeline under overload situations. Our goal with these techniques is to ensure during execution that the SLA monitoring elements deployed on the service machines will never exceed 30% of CPU and memory consumption.

For the capacity management our main interest is to understand statically the memory and CPU consumption of queries based on an event input rates. In the light of work from (1) (2), we have gathered statistics over time of our operators depending on their semantic (selection, aggregation, projection) and build a model of their composition. This enables the SLA runtime to evaluate the burst potential of a given query depending on input rates. This technique gives a rough estimate of potential burst and does not take into account external factors from the environment.

Because capacity management only gives a rough estimate of potential burst and does not prevent overload situation, the SLA monitoring system is complemented by an events shedding gate (7)(9) to stay in the 30% CPU and memory consumption range. Comparable to work from (3), (4) and (5) we introduce a drop operator as an entry gate as a first operator (and only first operator). This operator uses random events shedding which brings maximum efficiency.

The challenges when using static analysis in capacity management of the queries is to evaluate the roughness of the estimates. Because many external factors influence the queries execution at runtime, it would be interesting to refine the queries analysis during execution.

One of the main challenges of applying events shedding is the understanding of the quality impact (8) of the event loss on the final aggregated results as well as determining if a delayed execution (6) could save quality without destabilizing the system. From a quality perspective, it is difficult to predict the impact of the removal of low level events. It is also a fair assumption that the quality impact very much depends on the usage of the output events from the system. Therefore, it would be interesting to investigate special purpose shedding techniques, which are dependent on the SLA.

Another area that we would need to invest more is the understanding on how delayed execution of operators can help in overload scenarios. We believe that a schedule that takes the current load on the machine into account and the knowledge on how much CPU time an operator will consume can make a smarter decision between delayed execution and load shedding.

7 Conclusions

In this paper we discussed the use of Complex Event Processing for the management of large scale internet services. We presented two scenarios focused on SLA monitoring emphasizing the importance of language expressiveness, root cause analysis and environment stability. These issues are from our perspective the most critical and potential prevent a safe deployment of such monitoring system in large scale internet services. We also described our SLA monitoring solution which allows distributing and scaling the processing of events inside the management infrastructure.

Our experiments showed that the usage of Complex Event Processing nicely separates the implementation details from the semantics of the SLA. CEP also includes good support for scalability inside the infrastructure to anticipate future grows. It aids in the removal of the network bottlenecks created by the Pub/Sub layer, by moving computation closer to events' sources.

However as we have underlined there are still many challenges while deploying such system for large scale internet services. Better expressiveness of languages for writing queries would help the developers / IT operators. Debugging queries and enabling root-cause analysis is an important requirement for fine tuning service monitoring and understand service failures. Capacity management of the queries is also an important requirement when deploying the queries to understand potential bottlenecks and availability problems. In addition it plays a safe guard to ensure that the monitoring services do not draw to many resources away from the services.

We have detailed a few exploration paths which are promising. For the debugging and root cause analysis we have experimented with a linkage mechanism inside our logging framework supported by analysis tools. For the capacity management and scalability we are exploring flood gates mechanisms as well as mathematically formulating the query operators' memory and CPU requirements. Our work at simplifying the expressiveness of the standing queries was just identified as a subject of promising future work. More work is still needed to fully answer these requirements. As future work we are investigating a less intrusive debugging approach and an improvement of the automatic scalability.

8 Acknowledgement

The work reported in this paper has been done under the umbrella of the project SeCSE funded by the European Commission under contract FP6-IST-511680.

9 Works Cited

1. *SLA Monitoring: Shifting the Trust.* **O.Nano, M.Gilbert.** Ljubljana : IOS Press Amsterdam, 2005. Echallenges 2005.
2. Borealis. *http://www.cs.brown.edu/research/borealis/public/.* [Online] Brown University.
3. Stream. *http://infolab.stanford.edu/stream/.* [Online] Stanford University.
4. *The Design of the Borealis Stream Processing Engine.* **Daniel J. Abadi, and others.** 2005, Proceedings of the 2005 CIDR Conference.
5. *Query Processing, Resource Management, and Approximation in a Data Stream Management System.* **Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, Rohit Varma.** 2003. CIDR.
6. *Chain: Operator Scheduling for Memory Minimization in Data Stream Systems.* **Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani.** 2003. SIGMOD.
7. *Load Shedding for Aggregation Queries over Data Streams.* **B. Babcock, M. Datar, and R. Motwani.** s.l. : IEEE, 2004. ICDE.
8. *Approximate Join Processing Over Data Streams.* **A. Das, J. Gehrke, and M. Riedewald.** s.l. : ACM , 2003. SIGMOD.
9. *Load Shedding in a Data Stream Manager.* **N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, M. Stonebraker.** Berlin : s.n., 2003. VLDB .