B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 281

A Data Center Infrastructure Monitoring Platform Based on Storm and Trident

Felix Dreissig[1] Niko Pollner[2] (Advisor)

Abstract: Sensor data of a modern data center's cooling and power infrastructure fulfil the characteristics of data streams and are therefore suitable for stream processing. We present a stream-based monitoring platform for data center infrastructure. It is based on multiple independent collectors, which query measurements from sensors and forward them to an Apache Kafka queue. At the platform's core is a processing cluster based on Apache Storm and its high-level Trident API. From there, results get forwarded to one or multiple data sinks. Using our system, analytical queries can be developed using a collection of universal, generic stream operators including CORRELATE, a novel operator which combines elements from multiple streams with unique semantics. Besides the platform's general concept, the characteristics and pitfalls of our real-world implementation are also discussed in this work.

Keywords: Data Stream, Stream Processing, Apache Storm, Trident, Monitoring, Data Center

1 Introduction

Since the emergence of the data stream model, monitoring of sensor data has often been cited as a classical use case for data stream systems. As new measurement values arrive steadily and continuously, it is particularly fitting to view them as a stream of data. Rather than saving the values to a data store and performing periodic analysis, a data stream system can process them live and permanently.

In recent years, the data streaming paradigm has gained popularity due to the advent of new distributed stream processing systems primarily developed in the IT industry. The first and still one of the most popular systems of that kind is *Apache Storm*, which has been released as open-source software by *Twitter*. While *Storm* itself does not provide many processing guarantees, it also includes the *Trident* interface, which e. g. promises persistent state and exactly-once semantics.

In this work, we present the architecture and real-world implementation of a monitoring platform for data center infrastructure based on *Storm* and *Trident*. To the best of our knowledge, it represents the first published usage of *Trident* for monitoring purposes and one of the first publications on practical *Trident* usage.

Data center infrastructure primarily refers to cooling and power distribution systems. These are auxiliary, but critical to the actual IT equipment like servers and network devices. The

[1] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Computer Science 6, Martensstrasse 3, 91058 Erlangen, felix.dreissig@fau.de
[2] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Computer Science 6, Martensstrasse 3, 91058 Erlangen, niko.pollner@fau.de

particular data center covered in this work uses a modular structure for electricity supply and cooling with a multitude of sensors accessible through an IP-based network.

The platform aims to support different areas of monitoring which are usually handled by separated systems today – health monitoring with alerts in case of problems, data collection and visualization for capacity planing and performance analysis and live observation of metrics, e. g. in a schema of system components.

In the remainder of the work, we will first revisit the foundations of data stream processing and introduce basic concepts of *Storm*. Afterwards, the general architecture of the monitoring platform and its operator model are discussed. Section 4 features a presentation and evaluation of the platform's implementation. This is followed by a review of related works. Finally, we close with a conclusion and an outlook on future developments.

2 Foundations

2.1 Data Stream Systems

In contrast to database management systems, which work on a persistent collection of data, data stream processing systems execute continuous queries on sequential, append-only data streams. According to Babcock et al. [Ba02], a data stream is characterized by the online arrival of data elements, no influence for the system on the arrival order of data, potentially unbounded data and the fact that in general, elements cannot be retrieved once they are processed.

Some operations provided by data stream processing systems have counterparts in database management, while others introduce special concepts. The most prominent of such concepts are windows, which limit the number of elements to be considered for a subsequent operation like a JOIN or an aggregation. Windowing features are provided by virtually all stream processing systems, but the available characteristics can vary widely [Bo10].

Traditionally, queries for most data stream processing systems are expressed in domain-specific, declarative languages similar to SQL. *Storm*, which stems from a background in a Big Data community, is programmed using high-level procedural programming languages, mostly Java or Scala, instead.

2.2 Storm

Instead of a comprehensive set of operators, *Storm* provides a programming and runtime framework for stream processing code. Developers generally implement two basic abstractions – Spouts, which represent data sources, and Bolts, which perform processing steps on data elements. Since Bolts may emit an arbitrary number of elements, including zero, per input element, they can also act as data sinks.

Storm is distributed by nature and designed to run on a cluster of independent hosts for scalability and fault tolerance. The system will distribute the Spout and Bolt instances across the cluster and re-assign the workload upon failure of individual hosts. However, keeping persistent state in Bolts was not supported until the recent *Storm* version 1.0. Processing guarantees for the failure case are configurable between at-least-once and at-most-once semantics [To14].

2.3 Trident

Trident is an alternative, higher-level programming model building on top of *Storm*, developed and distributed as part of the *Storm* project. Its key promises are high throughput due to "micro-batching" in the order of several hundred milliseconds, a high-level interface similar to *Apache Pig*, support for persistent state and exactly-once semantics.

Operations are defined by implementing the *Trident* Java APIs, which e. g. provide support for common operations such as aggregations and filters. One or more of these operations are then mapped to a plain *Storm* Bolt for execution. With most operation types, the developer is unaware that the system combines the stream elements to micro-batches internally.

As long as processing results are not revealed to the outside world (and there are no side effects), streaming operations can be repeated an unlimited number of times without any consequences. This is why state is fundamental to *Trident*'s exactly-once support: Since true exactly-once delivery cannot be guaranteed in a distributed system, the effect of exactly-once behavior is achieved by only communicating updates to outside systems once. This generally requires those updates to be idempotent; the exact requirements are defined by *Trident*'s "transactional" and "opaque-transactional" modes [St]. It should be noted that state is always externalized in *Trident*, as it is only gets stored an external system like a database or a distributed file system and only becomes relevant when communicating results to such systems.

3 Concept

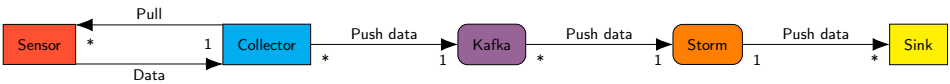

Fig. 1: High-level Architecture of the Complete Monitoring System

3.1 Components

Besides *Storm*, the complete monitoring system is made up of several other components, which are depicted in Fig. 1.

Collectors gather values from one or more sensors and forward them to the rest of the system. For data center infrastructure, they act as a gateway between sensors from the

electrical engineering domain and the software-based monitoring system. For this purpose, existing data collection software like *collectd* can be used. The collectors also mark the boundary between pull-based and push-based data transfers, as they have to explicitly request measurements from the sensors and then actively send them to the processing system.

Apache Kafka is a distributed message queue system commonly used to feed data into and move it out of *Storm* and other stream processing systems. In our architecture, it serves as a common adapter between collectors and *Storm*, for which *Kafka* Spout implementations are readily available. It also guarantees data persistency during *Storm* downtimes because of failures or maintenance. In that case, data can queue up in *Kafka* and be processed afterwards with increased latency, but without data loss.

The central processing point for measurements from all collectors is a *Storm* cluster that first normalizes the data and unifies them to common message schemas in order to simplify further processing. Afterwards, various analyses are performed on the unified data elements. The structure and characteristics of our implementation using *Storm* are covered in Sect. 4.

Finally, normalized raw measurements or processing results get stored to a data sink. Examples for data sinks include a time-series databases for historic collection or a notification system for monitoring alerts.

3.2 Operator Model

3.2.1 General Considerations

Even though *Storm* and *Trident* offer the possibility to express arbitrary queries using a high-level programming language, we developed a set of generic stream operators. These are provided by the platform and can be used to express analytical queries. The operator selection has been identified based on real-world use cases in data center infrastructure monitoring. It aims to be highly universal and reusable, so that development of custom operators is only necessary in special cases. The reusability is supported by the unification of all data to common schemas, as explained in the previous section.

The operator set contains common stream operators such as filters, windowed JOINs and mapping operators. Following the reusability paradigm, arithmetic and boolean operations are provided in form of elementary mapping operations like addition or “less than”. Mappings that work with multiple operands expect so-called Composited Records as input, which consist of multiple individual data elements. Those are generated by a JOIN or CORRELATE, a novel operator to combine elements from multiple streams.

3.2.2 CORRELATE

CORRELATE can be considered a ‘symmetrical version’ of the NEXT operator from *Cayuga* [De07]: While NEXT buffers an element from the first stream and waits for the

next one from the other stream, CORRELATE buffers the first element from either stream until an element is received on the respective other one. Upon its arrival, a Composite Record consisting of both elements is emitted. If yet another message from the first stream is received before one from the second stream, the new message replaces the buffered one. Another way to describe the CORRELATE operator is to view it as JOIN between count-based windows of size 1, which lose their content as soon as it becomes part of a Composite Record.

Primary use case for the CORRELATE operator is the combination of measurement data from various origins and of various types. In an ideal setup, all measurements would be taken simultaneously, making correlation rather simple and the derived metrics very exact. In practice, however, the frequency and exact moments of measurements are outside the control of the processing system. In contrast to windowed JOINs, CORRELATE ensures that every input element is used in at most one output element. With windows, a new result would instead be generated for every input message. Especially when combining lots of streams with different and possibly irregular message frequencies, this would weaken the significance of resulting metrics.

Correlations between more than two streams may be performed as well. In that case, the operator buffers elements from multiple streams until at least one element has been received per stream. Note that while the explanation refers to "streams", *Trident*'s standard API does not support operations with multiple input streams. Therefore, the following uses of CORRELATE expects all input elements to be in the same stream. Which elements to perform correlation on must be specified as parameter of the operator in that case.

3.2.3 Example Query

We now discuss usage of the operators for an example in data center infrastructure monitoring, the calculation of the Power Usage Effectiveness (PUE) value. Despite several criticisms, it is still the prevalent metric for a data center's energy efficiency and is defined as follows:

$$\text{PUE} = \frac{\text{Total facility power}}{\text{IT equipment power}}$$

Total facility power is usually higher than IT equipment power because of infrastructural overheads such as cooling and power distribution. Thus, the ideal PUE value of 1.0 is merely theoretic.

An implementation using our stream operators is depicted in Fig. 2. The CORRELATE operator combines the measurements required to calculate total facility power, in this case IT equipment power and one value from the main power distribution. The former consists of the power values from several sub-distributions, which are all part of the correlation. In the next step, their sum is calculated with the ADD operator. Afterwards, the facility power is divided through that sum to calculate the PUE metric.

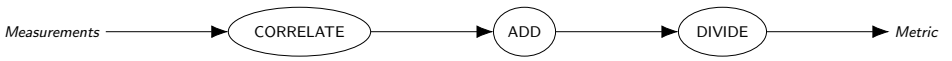

Fig. 2: Operator Graph for the PUE Calculation Query

4 Implementation

As mentioned previously, the platform has been implemented using *Storm* and *Trident*. *Storm* was selected since it is well-established, available as open-source software and enables scalability through its distributed nature. At the time of the implementation, the current *Storm* version was 0.10. It lacked recent features such as built-in windows and stateful Bolts, which made *Trident*'s state management capabilities appear very attractive. Other features like the built-in operators and exactly-once semantics also played a significant role in the decision for *Trident*. All our *Trident* code is designed for "opaque-transactional" properties [St].

4.1 Operators

While the implementation of mapping operators and filters in *Trident* was straight-forward, CORRELATE and windows provided bigger challenges. Support for filters is readily provided through a base class, with the filter condition being specified for individual implementations. Mappings correspond to "Functions" in *Trident*, another base class where only the method generating the mapping result is to be implemented.

CORRELATE and windows have to keep an internal buffer of messages for future output. However, *Trident*'s built-in stateful operations do not support operations which keep state and emit elements at the same time. Therefore, we had to use regular operations and manually perform state management. Nevertheless, our state implementation adopts *Trident*'s abstractions or at least follows similar practices. It stores data in the existing *Apache ZooKeeper* cluster already used by *Storm*, but does not use *ZooKeeper*'s coordination features.

4.1.1 State

The *ZooKeeper* State class implements *Trident*'s State interface. When a state commit begins, a new *ZooKeeper* transaction is created and stored together with the specified *Trident* transaction ID in memory. Keeping that information locally is feasible, as only one State object should be used per *Trident* micro-batch. Once the actual commit happens, that *ZooKeeper* transaction gets executed. Between commit begin and finish, storage instructions for arbitrary data can be added to the transaction. The implementation takes care of saving two versions of data for "opaque-transactional" properties. This is achieved by always storing two versions of the data to *ZooKeeper* together with a *Trident* transaction ID.

Reads from the *ZooKeeper* State happen outside of commits, but also require a *Trident* transaction ID to be specified. The recent stored version of the requested data is returned

if that transaction ID is greater than the one stored with the data; otherwise, the previous version is returned. This ensures that only confirmed state, which cannot be overwritten anymore, is ever returned from a read.

4.1.2 CORRELATE

The CORRELATE operator does not edit existing messages, but uses them to emit new Composite Records. The only *Trident* base operation supporting such behavior is the Aggregator interface. As the name suggests, it is actually designed for aggregations, but since it can emit arbitrary elements, it really is the most flexible kind of *Trident* operation.

Each CORRELATE instance has its own *ZooKeeper* State, where every new micro-batch marks the begin of a new State commit. The batch's single elements are added to an in-memory buffer if they are part of the correlation. At the end of the batch, elements missing for the correlation are retrieved from the State. At this step, it is strictly necessary that reads only return confirmed data.

When all relevant records are available locally, it is checked whether they contain records for all required types of elements. If that is the case, a Composite Record is subsequently created and emitted and the used individual elements are deleted from the State. Otherwise, those elements that were previously only kept in memory get written to the *ZooKeeper* State.

Currently, updates are committed to the *ZooKeeper* State at the end of a micro-batch. *Trident*'s built-in stateful operations actually perform commits asynchronously to processing. Ideally, our custom state updates would be triggered by these commit commands as well. However, receiving the commands is not trivially possible and requires a deep understanding of *Trident*'s undocumented internals. It would also require the *ZooKeeper* State to be adjusted to support multiple concurrent transactions.

4.2 Sinks

The *Graphite* time-series event store serves as proof of concept sink in our implementation. We implemented a special *Trident* operation, which stores selected measurement data and processing results to *Graphite*. Other sinks could be connected in a similar fashion.

For graphing of the time-series values, *Grafana* is used. It allows to display data from different sources, including *Graphite*, in a web frontend. Multiple dashboards, each containing an arrangement of graphs, are defined. An example *Grafana* dashboard is shown in Fig. 3.

4.3 Evaluation

The implementation has shown that it is generally feasible to build a monitoring platform on top of a stream processing system and the concept of collectors, processor and sinks.

Fig. 3: Excerpt of the *Grafana* Dashboard for PUE Values (Details about the specific data center obliterated)

Results, even from reasonably complex queries such as PUE calculation, are available in real-time. While the example operator graph from Fig. 2 is rather simple, the operator graphs can get very complex even for mildly more complex use cases. This a downside of the elementary, generic operator model.

On the other hand, the platform's usage can easily be expanded to further use cases from data center infrastructure monitoring or even other domains of monitoring because of the generic operators and common data schemas. Even if only the *Grafana* sink has been implemented as proof of concept, adding further sinks such as an alerting system is straightforward. Therefore, the goal of supporting different areas of monitoring is achieved as well.

One design decision that has proved particularly useful is the incorporation of *Kafka* as queue for the case of *Storm* failures. During development, it prevented several data losses due to programming errors. After bugfixing, the data could be successfully reprocessed on the then error-free code. The distributed nature of *Kafka* and *Storm* also means high scalability of the platform's core components, as both systems can be scaled to a large number of cluster nodes.

However, limitations of *Trident*'s persistent state capabilities have been revealed while implementing stateful operators such as windows or the CORRELATE operator. The current operator implementations work around these limitations, but are not suitable for large-scale production usage. *Trident*'s support for exactly-once processing is also rather limited on closer examination, as one still has to take care of idempotent state updates and a similar effect could thus be achieved with plain *Storm*'s at-least-once guarantees as well.

5 Related Works

Several authors propose monitoring systems with architectures similar to the structure of collectors, processor and sinks, but without the notion of stream processing [BF15; Ka13]. In [Si13], the stream-based *BlockMon* network monitoring system is extended by distributed processing capabilities and its performance is compared to *Storm* and the *S4* stream processing system.

GEMINI2 [BKB11], a monitoring platform for computing grids, is based on the *Esper* stream processing system. Rather than using a central processor like our system, partial queries are performed on hosts closer to the monitored system and their results are subsequently combined to the complete result.

Smit et al. [SSL13] present *MISURE*, a stream-based platform very similar to the one introduced in this work: It is also based on *Storm*, can incorporate data from various sources and makes results available to different consumers, including a time-series database. *MISURE*'s primary focus is unified processing of results from the monitoring systems of different, possibly geographically distant, cloud computing providers. Besides this difference in the area of application, *MISURE* does not use *Trident*, but plain *Storm* 0.60.

Only a few stream-based monitoring systems are known to be used in the IT industry. *Volta* [Re15], a cloud monitoring platform for systems from the whole application stack, has been developed at *Symantec* using *Storm*, *Kafka*, and *Grafana*. *Librato* is a stream-based monitoring software as a service. It started out using *Storm* as central processor, but recently switched to a custom system [Je15].

6 Conclusion and Outlook

This work showed that monitoring is not only a common use case example in the data streaming literature, but data stream processing systems can actually be utilized for practical use cases in data center infrastructure monitoring.

The architecture for a monitoring platform based on *Storm* and *Trident* has been developed as well as an operator model. The subsequent implementation revealed that while *Trident* promises support for persistent state, its API has shortcomings for stateful operators. Another contribution of the work is the CORRELATE operator, which combines elements from multiple streams with unique semantics.

As a next step, *Trident* should be replaced with a stream processing system that is better suited for the requirements of our operator model. An obvious choice would be *Storm* 1.0 with its support for persistent state and windows. Another option which we are currently exploring is *Apache Flink* (formerly *Stratosphere* [Al14]), which on top of those features also provides built-in support for application time.

With regard to more applications of the platform, additional data sinks are required. Afterwards, the platform may be expanded towards other monitoring domains such as individual hosts and network infrastructure.

References

[Al14] Alexandrov, A. et al.: The Stratosphere platform for big data analytics, The VLDB Journal 23/6, pp. 939–964, 2014.

[Ba02] Babcock, B. et al.: Models and Issues in Data Stream Systems. In: Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, Madison, WI, pp. 1–16, 2002.

[BF15] Bauer, D.; Feridun, M.: A Scalable Lightweight Performance Monitoring Tool for Storage Clusters. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IEEE, Ottawa, ON, pp. 1008–1013, 2015.

[BKB11] Balis, B.; Kowalewski, B.; Bubak, M.: Real-time Grid monitoring based on complex event processing, Future Generation Computer Systems 27/8, pp. 1103–1112, 2011.

[Bo10] Botan, I. et al.: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems, Proceedings of the VLDB Endowment 3/1, pp. 232–243, Sept. 2010.

[De07] Demers, A. et al.: Cayuga: A General Purpose Event Monitoring System. In: Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, pp. 412–422, 2007.

[Je15] Jenkins, R.: SuperChief: From Apache Storm to In-House Distributed Stream Processing, Librato Blog/, Sept. 24, 2015, URL: http://blog.librato.com/posts/superchief, visited on: 2016-10-20.

[Ka13] Kai, L.; Weiqin, T.; Liping, Z.; Chao, H.: SCM: A Design and Implementation of Monitoring System for CloudStack. In: 2013 International Conference on Cloud and Service Computing, IEEE, Beijing, pp. 146–151, 2013.

[Re15] Renganarayana, L.: Volta: Logging, Metrics, and Monitoring as a Service, Talk recording and slides, Jan. 7, 2015, URL: http://www.slideshare.net/LakshminarayananReng/volta-logging-metrics-and-monitoring-as-a-service, visited on: 2016-10-20.

[Si13] Simoncelli, D.; Dusi, M.; Gringoli, F.; Niccolini, S.: Stream-monitoring with Blockmon: convergence of network measurements and data analytics platforms, ACM SIGCOMM Computer Communication Review 43/2, pp. 29–36, Apr. 2013.

[SSL13] Smit, M.; Simmons, B.; Litoiu, M.: Distributed, application-level monitoring for heterogeneous clouds using stream processing, Future Generation Computer Systems 29/8, pp. 2103–2114, 2013.

[St] Storm Project: Verison 0.10.0 Documentation, Apache Software Foundaton, chap. Trident State, URL: https://storm.apache.org/releases/0.10.0/Trident-state.html, visited on: 2016-10-19.

[To14] Toshniwal, A. et al.: Storm @Twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, Snowbird, UT, pp. 147–156, 2014.