Multi-tenant Pub/Sub Processing
for Real-time Data Streams

Álvaro Villalba and David Carrera

Technical University of Catalonia (UPC)
Barcelona Supercomputing Center (BSC)
{alvaro.villalba,david.carrera}@bsc.es

Abstract. Devices and sensors generate streams of data across a diversity of locations and protocols. That data usually reaches a central platform that is used to store and process the streams. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories. In the former case, stream processing technologies are required to operate on the data; in the latter batch analytics and queries are of common use.

This paper introduces a runtime to dynamically construct data stream processing topologies based on user-supplied code. These dynamic topologies are built on-the-fly using a data subscription model defined by the applications that consume data. Each user-defined processing unit is called a Service Object. Every Service Object consumes input data streams and may produce output streams that others can consume. The subscription-based programing model enables multiple users to deploy their own data-processing services. The runtime does the dynamic forwarding of data and execution of Service Objects from different users. Data streams can originate in real-world devices or they can be the outputs of Service Objects.

The runtime leverages Apache STORM for parallel data processing, auto-scaling and operation placement, that combined with dynamic user-code injection provides multi-tenant stream processing topologies. In this work we describe the runtime, its features and implementation details, as well as we include a performance evaluation of some of its core components.

Keywords: Big Data, Analytics, Stream Processing, Real-time Data Processing, Programming Models, Internet of Things, IoT

1   Introduction

In the last years, Big Data and Internet of Things (IoT) platforms are clearly converging in terms of technologies, problems and approaches. IoT ecosystems generate a vast amount of data that needs to be stored and processed, becoming a Big Data problem. Devices and sensors generate streams of data across a diversity of locations and protocols that in the end reach a central platform

that is used to store and process it. Processing can be done in real time, with transformations and enrichment happening on-the-fly, but it can also happen after data is stored and organized in repositories.

This situation implies an increasing demand for advanced data streams management and processing platforms. Such platforms require multiple protocols support for extended connectivity with the objects. But also need to exhibit uniform internal data organization and advanced data processing capabilities to fulfill the demands of the application and services that consume these streams of data.

To provide answer to this growing demand, ServIoTicy[1] is a state-of-the-art platform for hosting real-time data stream workloads in the Cloud. It provides multi-tenant data stream processing capabilities, a REST API, data analytics, advanced queries and multi-protocol support in a combination of advanced data-centric services. The main focus of ServIoTicy is to provide a rich set of features to store and process data through its REST API, allowing objects, services and humans to access the information produced by the devices connected to the platform. ServIoTicy allows for a real time processing of device-generated data, and enables for simple creation of data transformation pipelines using user generated logic. Unlike traditional service composition approaches, usually focused on addressing the problems of functional composition of existing services, one of the goals of the ServIoTicy is to focus on data processing scalability. Other components that can be connected to ServIoTicy provide added capabilities to automatically create compositions of high-level services using existing tools [13].

The core of the ServIoTicy runtime relies on a novel programming model that allows users to dynamically construct data stream processing topologies based on user-supplied code. These topologies are built on-the-fly according to a data subscription model defined by the applications that consume data. Once a stream subscriber finishes its work, it is freed from the platform until it is needed again. Each user-defined processing unit is called a Service Object (SO). Every Service Object consumes input data streams and may produce output streams that others can consume. Data streams can originate in real-world devices or they can be outputs of Service Objects deployed in the platform.

Advanced streaming and analytics platforms such as ServIoTicy are complex pieces of software that integrate a large set of components under the hood. They hide their complexity behind simple REST APIs and multi-protocol channels. ServIoTicy leverages Apache STORM runtime for parallel data processing, auto-scaling and operation placement, that combined with dynamic user-code injection provides multi-tenant stream processing topologies.

This paper provides insights on the performance properties of ServIoTicy as an starting point for the construction of advanced cloud provisioning strategies and algorithms. The work presented here focuses on the processing topologies built in ServIoTicy, although some details about other platform components are also provided.

---

[1] servioticy.com

The source code of ServIoTicy is freely available as an open source project[2] in GitHub. The platform is also available for single node testing as a vagrant box, downloadable from a github repository[3].

The main contributions of this paper are:

– A technique for user-code injection on a data stream processing runtime that allows for multi-tenant stream processing on-the-fly. This runtime is the core of the ServIoTicy platform.
– An insight on the performance of the code-injection technique, including response time end-to-end in a processing pipeline and across stages.

The next sections of the paper are organized as follows: Section 2 introduces a set of abstractions defined in ServIoTicy for managing data associated to objects; Section 3 describes in detail the stream processing runtime of ServIoTicy; Section 4 presents the evaluation methodology and the experiment included in the paper; Finally, Section 5 goes through the related work and Section 6 provides some conclusions and future lines of work.

2   Abstractions used in ServIoTicy

Several abstractions are used in ServIoTicy to embrace the different entities involved in the existence of IoT ecosystems.

– Web Object: Web Objects are physical objects sitting on the edge of ServIoTicy and capable of keeping for example HTTP-based bi-directional communications, such that the object will be able to both send data to the platform and receive activation requests and notifications.
– Service Object: Service Objects are standard internal ServIoTicy representations of Web Objects. This entity serves mainly for data management purposes and has a well-defined and closed API. That API is needed in order to streamline and standardize internal access to Service Objects, which can in turn represent a variety of very different Web Objects providing very different capabilities.
– Sensor Update: Sensor Updates are the unit of data sent by a Web Object to its Service Object. It contains the different synchronously sensed values and a timestamp that is maintained all over the pipelines. A subscription or a query to a Service Object will get the data in this format.

3   Data Processing Pipelines

Service Objects store their associated data in abstractions called *streams*. The unit of data that can be observed for one stream is called a *Sensor Update* (SU). Applications can subscribe to or query data associated to any stream. Streams can be of two different types:

---

[2] https://github.com/servioticy
[3] https://github.com/servioticy/servioticy-vagrant

- Simple data streams store data generated in the physical world by a sensing device, assuming that a device with N sensors will generate N streams of data that will be grouped in a Service Object abstraction that represents the device.
- Composite data streams represent transformations (aggregate, merge, filter or join, among other possibilities) performed on other data sources (either by devices located in the physical world or by Service Objects existing in the ServIoTicy platform). They can be thought about as a virtual (non-physical) sensor of the SO.

From an API perspective there is no difference between a simple stream and a composite stream, as they both support queries and subscriptions. Therefore, the inputs of composite stream can be streams or other composite streams. These chained transformations of SUs are called *Data Processing Pipelines*.

3.1 Data Structures

The structure of a Sensor Update that corresponds to a given stream is basically composed of a series of *Channels* associated to the dimensions of the data represented by the stream (e.g. a geo-location stream may contain two channels representing the latitude and the longitude correspondingly), and a timestamp reported by the data source as the time at which the Sensor Update was generated.

The composite stream structure is similar to the structure of a SU. It contains channels, and each channel contains a so-called 'current-value' field that represents the output value that the composite stream will emit after ingesting a new SU, assuming that the output is not filtered. In a SO document, the content of a 'current-value' field is a string with a JavaScript variable assignment using any mix of basic operator and functions from the Math object, String object, Array object, as well as shorthand conditional expressions (a = b ? true : false). The result of the assignment to 'current-value' will always be numeric, a Boolean, a string or an array of the previous types. It will be stored and emitted to its subscribers.

3.2 Stages of the Processing Pipeline

Once a SU reaches a composite stream as one of its inputs, it goes through a number of stages in order to transform it into a new output SU. This process of ingesting a SU and processing it until a new SU is produced can be summarized as the following set of stages:

1. Subscriber dispatching: A sensor update gets into the processing pipeline, along with its origin information. This stage looks for the subscribers of its origin and if they are composite streams, they are requested and sent to the next stage with the SU.
2. Data Fetching: The composite stream may need access to the data stored by other streams that are inputs involved in the data transformation. In each stage, the sources needed by the stream are queried and their data made available for the rest of the stages, altogether with the original SU. References to fields on the Sensor Updates are made using JSONPaths.

3. Transformation & filtering: Data transformation is performed by taking all the SUs extracted from all the data sources, and operating on their associated data using JavaScript algebraic operations and its Math object functions, String object operations, Array object operations, and boolean operations, to finally obtain a single value for the new SU to emit. Also, before and after the transformation SUs are discarded if a defined filter assertion is false, and no further stages would follow.
4. Store, trigger actions and emit: Finally, the generated SU gets stored and emitted to the stream subscribers. Additionally, in this final stage, actions to be sent back to SOs are triggered. Such actions will end up being sensor actuations that will be driven through the WOs that embed the actual physical objects.

In ServIoTicy, basic physical object actuation is driven through SOs. When a SO gets an action invoked through the SO actions API, the action is initiated on the corresponding WO, that will act as a proxy for the physical actuator. If a user needs to be able to manually request the execution of a composite action (involving multiple SOs), it is necessary to create a SO that includes the desired action and references to the individual SOs representing each of the physical objects to be actuated, so that the composite action can be properly triggered.

3.3   Design Principles

The data processing pipelines introduced in this work are intended to be scalable in accordance with other works found in the literature [15]. In particular, the key design principles considered for the data processing pipelines were:

- Event-driven: A new SU calculation is triggered in a stream when it receives a SU.
- Lock-free: A stream that needs of several different SUs to generate a new one will not lock until all of them are received. It makes use of the received SU, and queries the last SUs from the other needed streams.
- Real-time data processing oriented: Each new SU is processed individually without waiting for a batch.

The approach followed by ServIoTicy is an asynchronous model for which only one of the sources needs to issue a sensor update to trigger the processing of the composite stream. It enforces a high rate of updates and avoids locking the generation of new updates because one sensor is idle. This situation would lock an entire pipeline.

3.4   Time, Data Consistency and Efficiency

A composite stream can take as inputs the most recent SU from any stream declared in the platform, either from its own Service Object or from any other Service Object. In the context of a particular data stream, that receives SUs as inputs and stores data associated to its outputs in the platform, some restrictions need to be in place to keep chronological consistency of the data being produced by a given composite stream.

More formally, let S be a composite stream that takes as inputs the SUs generated by N streams. Let $su_i^{t_i}$ be the the most recent SU associated to the ith stream that is a data source for S, where $0 \leq i < N$, and let be t_i the associated timestamp to $su_i^{t_i}$. Also, let $su_s^{t_s}$ be the most recent SU associated to the stream S. Notice that it is possible that $\exists_i$ such that $i = s$ if S consumes its own previously generated data to produce new outputs.

Then we can define $SU_{s,in}^t = \{su_0^{t_0}, su_1^{t_1}, \ldots, su_{n-1}^{t_{n-1}}\}$ as the set of N inputs that S will use to produce one new output $SU_{s,out}^t$ with timestamp t. This output will be defined as a function $SU_{s,out}^t = f(SU_{s,in}^t)$ that is user-defined.

Given these definitions, ServIoTicy needs to guarantee that the function f is calculated (and an output $SU_{s,out}^t$ emitted) only once for the same set set of input values, and that at least one of the SUs in $SU_{s,in}^t$ needs to be updated (with a more recent timestamp) to trigger the computation again. Furthermore, it is necessary that the set $SU_{s,in}^t$ satisfies that $\exists su_i^{t_i} \in SU_{s,in}^t$ such that $t_i > t$ to initiate the computation of f to emit $SU_{s,out}^t$.

This restriction can be enforced by checking all the elements of $SU_{s,in}^t$ everytime that an element of the set is updated. But this approach can result in performing large amounts of costful operations just to decide that the conditions were not satisfied and that no new output needs to be emitted.

To mitigate this problem, ServIoTicy relaxes the previously stated restriction to the form $t_j > t$ where $0 \leq j < N$ and $su_j^{t_j}$ is the actual element in $SU_{s,in}^t$ that triggered the computation. This relaxation is possible because if an element exist in the set other than the one triggering the computation that has a more recent timestamp than t, then this it is very unlikely that this element has been computed before in time, because then t would have to be as recent as its timestamp. Otherwise, if the element with more recent timestamp has not yet triggered the computation, then it means that the SU has been stored for the source stream and it must be awaiting in a queue its time to be processed, and therefore it will trigger the computation soon.

3.5   Execution trees of the Data Processing Pipelines

The structure of a pipeline created using the ServIoTicy subscription model is by definition a directed graph. In practice, though, it behaves more like a set of trees. The reasoning behind this statement is discussed in this section.

When an update reaches a stream, if it is newer than the last generated update, the computation will be triggered. But if the received update is as new as the last generated update, the computation will be discarded. Consider a stream that has several inputs and they originally come from the exact same entry stream to the pipeline (source). When one of the inputs receives an update, at some point all the other inputs will receive an update with the same timestamp and the subsequent computations will always be discarded. Only the first update to reach the stream will trigger the computation.

From this reasoning it can be deduced that the set of paths of the triggered computations from a single source will always end up looking like a tree. For example Figure 1(a) represents the graph of a valid pipeline. The computations that would be generated from the subscriptions $d{\rightarrow}c$ and $h{\rightarrow}e$ are discarded for

the explained reasons. Therefore the execution graphs look like in Figure 1(b), and updates from d to c and from h to e will only be queried.

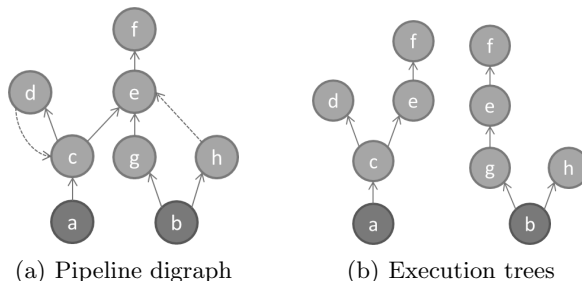(a) Pipeline digraph   (b) Execution trees

Fig. 1. Relation between a pipeline and its execution trees

Another interesting property of a pipeline is the novelty of its generated data, and it is useful for evaluating the quality of a stream. A stream generates novel data when it has an input with a source that no other input of the same stream has. The further a stream is in a path from the last new source addition, the less novel its generated SUs are. For example in Figure 1(a), c, g, h and e are 1 level more novel than f and d. See that e gets data sourced on b from two inputs, but theres also another input sourced on a. On the other hand f and d are one vertex away from the most novel source. At the levels of data novelty of this example, getting data from f or d is not a problem. The problem comes when the distance from the most novel stream is too far away will always take too much time to process an SU that will not add much value to what it is already evaluated, and will generate several discarded computations which will end up being time consumed without a result. Novel data means faster dispatch, less noise in the pipeline and more added value on the data.

3.6 Runtime implementation and user-code injection

The software that dispatches the incoming SUs and executes the pipelines runs on STORM. STORM topologies are static, but the pipelines can easily change over time, add connections between them, and have arbitrary sizes. For this reason the STORM topology in ServIoTicy runs the stages described in Section 3.2, common to all the pipelines to be processed. On the subscribers dispatch stage, the target streams are requested, with the code to be executed in them (previously deployed by the owner of the Service Object using the REST API). In the different execution stages (filters and transformation), the JavasScript code related to it is executed on a JavaScript engine. The JavaScript engine used is Rhino.

4 Evaluation

This section presents a performance evaluation of the implementation of the ServIoTicy Data Pipelines.

4.1   Evaluation Methodology and Infrastructure

In the experiment we explored the performance of several randomly-generated topologies. We present here the average results for all of them and the specific results of one illustrative case. A number of SUs were submitted to the topologies, and we measured the time it took for each SU to be propagated to all the streams that were subscribed directly or indirectly to the SU.

To drive the evaluation we developed a tool to automate the generation and deployment of randomly generated Data Processing Pipelines. The tool provides several control knobs to customize the properties of the topologies being generated. The most relevant controls are the number of streams, the number of composite streams, the number of operands per stream and how the operands are distributed between the streams.

The tests were run on two sets of nodes: one set for running the client emulators and one set for running the servers of the system under test. The 'server' set was composed of 16 two-way 4-core Xeon L5630 @2.13GHz Linux boxes, for a total of 8 cores per node and 16 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 24GB of RAM. The 'client' set was composed of 2 two-way 6-core Xeon E5-2620 0 @2.00GHz Linux boxes, for a total of 12 cores per node and 24 hardware threads because hyperthreading was enabled. Each 'server' machine was enabled with 64GB of RAM. All nodes were connected using GbE links to a non blocking 48port Cisco 3750-X switch. The ServIoTicy data processing runtime was deployed on 2 server machines, and 1 client machine was used to generate the SUs. The REST API used the other nodes to host its components. For the data processing pipelines we used Apache STORM v0.9.2-incubating, Kafka v0.8.2.2 and ZooKeeper v3.4.5.
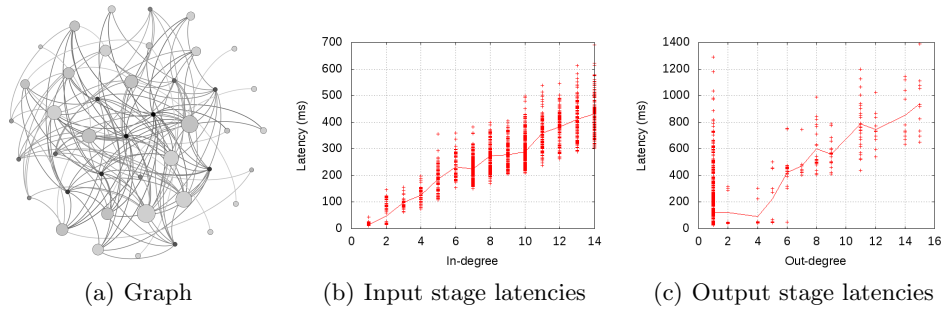

(a) Graph   (b) Input stage latencies   (c) Output stage latencies

Fig. 2. Topology number 3 and its related experiments results

4.2   Experiment

For this experiment, we generated six different testing topologies for ingesting data produced by a Service Object. The characteristics of these topologies are
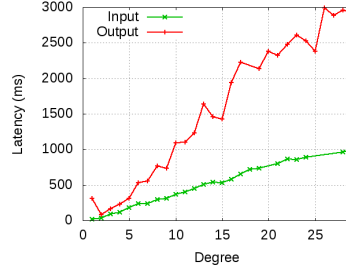
Fig. 3. Stage latency by degree

summarized in Table 1. They can be grouped based on their size (small, medium or large), and we randomly produced 2 samples of each complexity level. Based on our experience, topologies 1 and 2 emulate two realistically sized situations. Topologies 3 and 4 are large cases. Finally, topologies 5 and 6 are extreme cases. A graphical representation of topology number 3 is shown in Figure 2(a). In this figure, dark nodes indicate a high out-degree and big nodes represent high in-degree. The in and out degree related properties are also very relevant for this experiment, as they have a big impact on the metrics taken.

Table 1. Pseudo-random topologies

Type	Small		Medium		Big	
Id	1	2	3	4	5	6
Max in-degree	9	8	14	16	29	24
Mean in-degree	1.42	1.94	3.54	3.51	5.28	6.18
In-degree std. dev.	2.22	2.63	4.36	5.05	7.43	7.38
Max out-degree	4	7	15	15	25	28
Mean out-degree	1.42	1.94	3.54	3.51	5.28	6.18
Out-degree std. dev.	1.07	2.14	4.59	4.44	7.71	9.48
Edges	30	37	149	151	423	458
Nodes	21	19	42	43	80	74
Sources	11	9	17	18	30	24
Sinks	4	7	15	15	25	28
Density	0.14	0.21	0.17	0.16	0.13	0.16
Connectivity	1	1	1	1	1	1
Edge-connectivity	1	1	1	1	1	1

For each data source, 10 Sensor Updates were sent to the platform in sequence: a new update was generated only after the previous pipeline computation was finished. During the topology execution, two metrics were measured for each stream. The first metric is the execution time to perform all the data queries required to complete the processing, named the *input stage*. This metric measures the effect of using several inputs to generate a new update. The second

metric is the time difference between the instant at which a new update is emitted and the time at which all subscribers have received it: this metric measures how the topology processing time is affected by the number of subscribers at each stage of the processing pipeline. This is named in this section as the *output stage*.

Other stages were also measured, such as the injected code processing time or the time an update remained unaccessed in Kafka. The function to generate a new update was always a summation of the inputs, and so had complexity $O(n)$, being n the in-degree. However, these measures resulted on negligible times and have not been included in the discussion.

Figures 2(b) and 2(c) show all the latencies measured for topology number 3. Each dot in the plot represents one execution of a topology node with a given in- or out-degree that corresponds to the value in the X-axis. The average latency for each degree is also drawn in both charts as a solid line. As it can be observed, latency grows linearly with the degree level as some sequential operations are required for each operation. Although the communication is made asynchronous, the stages need to be closed before jumping to the next step for the topology, and therefore it is necessary to wait for all on-the-fly operations to complete at some point, what results in a waiting time that is proportional to the number of initiated operations and therefore the degree of the stage.

Finally, Figure 3 shows the average latency on the input and output stages for every related degree, across all six topologies. As it can be observed, the latency of both the input and output stages grow linearly, but in a higher pace in the output stage. While the in-degree latencies look almost the same to Figures 2(b), the out degree grows faster. The reason for this worse performance is that this Figure reports average values that are affected by the higher latencies of the bigger topologies. Therefore, the time of the output stage not only depends on the out-degree, but also on the total size of the topology. And in particular, the topology length is the most important factor that affects the performance of the topologies. The larger the topology is, the more operations are run in parallel in the topology and therefore the largest the response times of the components, resulting in a slightly higher latency to complete the processing of an update.

5   Related Work

In the last years several stream processing platforms have emerged, being Apache Storm [2] the most popular and it is used in this contribution as a platform runtime. Storm is a distributed, reliable, and fault-tolerant stream processing system, which was open sourced by Twitter after acquiring BackType and now distributed by the Apache Software foundation. ZeroMQ or Netty are the messaging interfaces between the computation units. In the last versions multi-tenancy was added in terms of several tenants deploying isolated topologies. This topologies are always in memory whether are being used or not, and there is not data subscription between tenants. Also open-source and distributed by the Apache Software Foundation are Apache Samza [10] and Apache Flink [1] and Apache S4 [12]. Apache Samza uses Kafka for the whole messaging between the computation units and YARN for resource management. Apache Flink is a streaming

dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. It has two APIs, one for data streams and another for data sets or batch processing. Flink also bundles libraries for domain-specific use cases like complex event processing and machine learning. Apache S4 is an already deprecated project started by Yahoo with a very similar topology based philosophy to Storm and an architecture resembling the Actors model. Microsoft Research developed a proprietary solution for complex event processing called StreamInsight [7]. It also leverages a programing model for temporal data streams, operator algebra and continuous queries. Other relevant foundations on stream processing in real-time from Microsoft come the CEDR [9] project. It is centered in the problem of keeping time consistency on event streaming. Other well known research related projects on data streams are Aurora [6] and its forks Medusa [8] and Borealis [5]. None of this projects are maintained anymore. From the perspective of data stream sharing, StreamGlobe [11] offers a Grid Computing solution using a P2P approach. It consist then in stream sharing between machines but not multi-tenancy.

Data Centric view of the IoT is not something new for ServIoTicy as it was widely covered in the survey presented in [14]. What ServIoTicy uniquely provides is an open source solution that challenges the features of commercial solutions such as Xively [4] and Evrythng [3], while extending their capabilities with the ability to inject user-defined code into its stream processing runtime.

6   Conclusions

In this paper we have introduced a multi-tenant data stream processing mechanism on top of Apache STORM that enables the tenants to share data streams between them. STORM provides auto-scaling capabilities that make it particularly suitable for cloud deployments. The ServIoTicy runtime allows for users to deploy custom service codes inside Service Objects in the form of composite streams, and subscribe those streams to multiple sources of data (either outside the platform on real-world devices or in other streams defined in the ServIoTicy platform by other users). The user-code will be automatically injected in the STORM topology and executed when a unit of data is generated from a source to which the composite stream is subscribed. The runtime is designed to be highly scalable, following a lock-free model that combines operations triggered by new data being generated inside or outside the platform, with queries performed over historic data logged for existing Service Objects. The design imposes some restrictions mainly related to the timestamps of the updates being processed, and some optimizations are applied to improve the scalability of the platform. A basic evaluation of the runtime is included in this work, showing how acceptable response times of less that 100ms can be delivered by basic composite streams, and that for most realistic pipelines can be processed in the range of less than a second. The work presented in this paper is, to our knowledge, the first multi-tenant IoT data processing platform for the Cloud.

7 Acknowledgments

References

1. Apache Flink official website, `http://flink.apache.org`
2. Apache Storm official website, `http://storm.apache.org`
3. evrythng official website, `evrythng.com`
4. Xively official website, `xively.com`
5. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: CIDR. vol. 5, pp. 277–289 (2005)
6. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB JournalThe International Journal on Very Large Data Bases 12(2), 120–139 (2003)
7. Ali, M., Chandramouli, B., Goldstein, J., Schindlauer, R.: The extensibility framework in microsoft streaminsight. In: Data Engineering (ICDE), 2011 IEEE 27th International Conference on. pp. 1242–1253. IEEE (2011)
8. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Load management and high availability in the medusa distributed stream processing system. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 929–930. ACM (2004)
9. Barga, R.S., Goldstein, J., Ali, M., Hong, M.: Consistent streaming through time: A vision for event stream processing. arXiv preprint cs/0612115 (2006)
10. Kleppmann, M., Kreps, J.: Kafka, samza and the unix philosophy of distributed data
11. Kuntschke, R., Stegmaier, B., Kemper, A., Reiser, A.: Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures. In: Proceedings of the 31st international conference on Very large data bases. pp. 1259–1262. VLDB Endowment (2005)
12. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: Data Mining Workshops (ICDMW), 2010 IEEE International Conference on. pp. 170–177. IEEE (2010)
13. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: iserve: a linked services publishing platform. In: The 7th Extended Semantic Web Ontology Repositories and Editors for the Semantic Web Workshop. vol. 596 (June 2010), `http://oro.open.ac.uk/23093/`
14. Qin, Y., Sheng, Q.Z., Falkner, N.J.G., Dustdar, S., Wang, H., Vasilakos, A.V.: When things matter: A data-centric view of the internet of things. CoRR abs/1407.2704 (2014), `http://arxiv.org/abs/1407.2704`
15. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Record 34(4), 42–47 (2005)