Self-organized Architecture for Sharing Data Streams at Large Scale

Nicolás Hidalgo and Erika Rosas

Department of Informatics
Universidad de Santiago
Santiago, Chile
Email: name.lastname@usach.cl

***Abstract*—Stream Processing Engines are designed to deal with real-time computing of massive data streams generated on social networks, news feeding, satellite images, sensor devices, among other sources. For example, in the context of the Internet of Things and Smart Cities, a high volume of data it is expected to be distributed geographically. In this context, the re-use of processed stream enables resource optimization by avoiding re-computation, enabling to provide aggregation and global data visualization. We propose a self-organized architecture to share data streams, which enables resource localization over a scalable, fault-tolerant Distributed Hash Table structure. The Stream Processing Engines are organized into a structured peer-to-peer network and they exploit a Publish/Subscribe system to publish and locate preprocessed streams, possibly in other geographic regions. In order to deal with communication latency problems in the peer-to-peer network, we propose a latency-aware algorithm that estimates distance between the nodes in the system.**

Keywords–Stream Processing; Peer-to-Peer networks; Large Scale Computing; Publish/Subscribe.

I. INTRODUCTION

Large scale streams can be generated in domains like meteorology, finance transactions, remote sensing, software logs, wireless sensor network, social interactions, telecommunications, just to mention a few. In order to process the amount of data generated in these scenarios, the capacity of many machines is required.

In the domain of the Internet of Things (IoT) or Smart City platforms, Wireless Sensor Networks (WSN) are used to monitor gas leaks, parking availability, traffic congestion, pollution levels, the infrastructure's health, and garbage levels [1]. These platforms enable integrating and visualizing data in order to make informed management decisions. The technological improvements and lower costs of these pieces of hardware provide an idea that in the future all the sensor information will be unmanageable in a centralized infrastructure. Moreover, in the domain of social networks and online interactions, there is a continuous stream of events that is used for computing trending topics or word counting. Such an analysis could also be applied to all interactions occurring on the Internet. Clickstream analysis and software logs are two examples of Internet-scale data generated continuously. Considering that massive data processing can be spread across geographically distributed machines, processing could be aggregated using a global large-scale infrastructure in real-time [2].

Stream Processing Engines (SPEs) are designed to deal with real-time processing of high volume data streams. SPEs have evolved from centralized solutions [3], to be able to distribute queries among several nodes [4][5][6], to finally distribute operators (or processing elements) that solve a query across different nodes [7][8]. The latter type is especially interesting since there are cases where a single machine cannot cope with the processing of one operator.

SPEs use a graph-oriented paradigm, where vertices represent operators, also called Processing Elements (PEs), and the edges represent flows of data. An application defines the PEs and their interaction through a graph. The PEs can filter, map, unite, aggregate data, or carry out more complex processing. SPEs can cope with thousands of events per second, however, processing large geographically distributed data requires movement of data across the network, increasing the traffic and compromising the real-time results.

Processing large scale streams requires close to real-time global responses and a highly scalable infrastructure. The volume of data changes over time and the lost of a small amount of data is not critical to the results. However, SPEs do not provide tools that facilitate sharing and reuse of processed stream between clusters that perform the same task. In this work, we propose a model to share streams of geographically distributed data in a scalable manner.

The contribution of this work is a model that organizes SPEs into a Distributed Hash Table (DHT) structure in order to maintain scalable localization of resources. The system uses a Publish/Subscribe system to find and share streams and avoid reprocessing the events. This is a scalable, fault-tolerant and self-organized infrastructure, which maintains low latency using locality aware techniques. The model enables users to estimate latency before deciding whether to use the processed stream found in the system. This is a Quality of Service (QoS) measure, in order to cope with real-time restrictions.

The remainder of this article is organized as follows: Section II presents our system model giving details about each component and Section III details the processing steps. We discuss related work in Section IV and finally, present concluding remarks in Section V.

II. SYSTEM MODEL

In traditional stream processing systems, each application is independent and works isolated from other applications producing data re-processing, which wastes resources. We propose a system model to process massive data streams in a distributed and collaborative manner. Our goal is to provide an infrastructure capable of dealing with the overwhelming amount of data available from diverse sources. Participants may share the pre-processed data streams, in order to avoid reprocessing of same data by other participants. We claim that

 ISBN: 978-1-61208-405-3

this solution can be helpful for building complex applications, which exploits the output data streams of smaller applications. Small applications could provide their results as an input to more complex applications, enabling a more efficient use of processing. This scenario is presented in Figure 1. In the figure, application 2 (App 2) is a complex application, which can be built on top of the data pre-processed and shared by application 1 (App 1) and 3 (App 3).

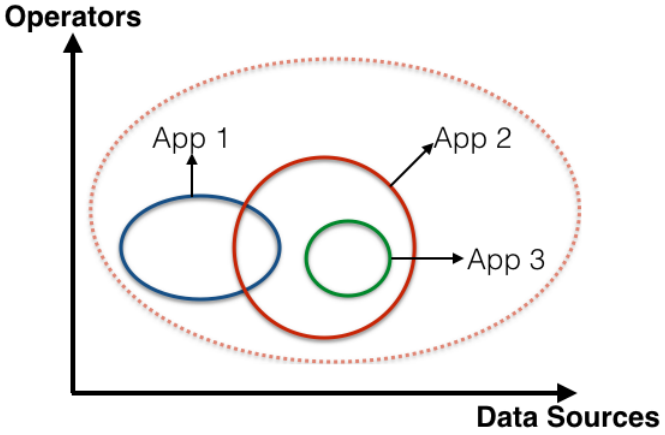

Figure 1. Stream processing and pre-processed data sharing

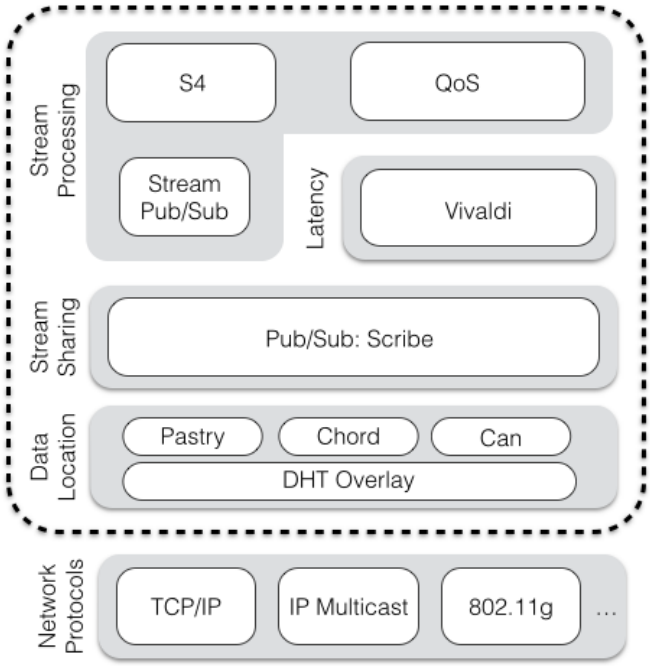

Figure 2. Distributed SPE (DSPE) architecture

The most important challenges of implementing a large-scale SPEs infrastructure are: (1) scalability, the system must be able to process a large amount of data over several geographically distributed SPEs; (2) low latency, due to the real-time nature of SPEs, latency must be minimal despite the SPEs location; (3) fault tolerance, the system must be capable of dealing with failures and changing conditions of the communication network (latency, partitions, etc.).

We consider a scenario where multiple SPEs, geographically distributed around the world, collaborate by publishing their processed streams within the community. In our system, SPEs are organized into a DHT structure in order to maintain a scalable localization of resources. The system uses the Publish/Subscribe paradigm to publish and share data streams with remote SPEs. Data streams are identified by a description file, which provides detailed i stream treatment nformation. Publish/Subscribe has become a popular communication paradigm that provides a loosely coupled form of interaction among many publishing data sources and many subscribing data sinks. In Publish/Subscribe paradigm, messages are published into channels or topics asynchronously, without knowing the subscribers. On the other hand, the subscribers state their interest in one or more topics, and receive messages without knowing the publishers. This decoupling of publishers and subscribers enables greater scalability and a dynamic network topology.

A. Layered view

We propose an architecture composed of SPEs organized over a DHT-based P2P network where peers or SPEs share their resources in order to reduce data re-processing. From now onwards, we consider a peer as an instance of a SPE. The proposed architecture is composed of 4 layers: the overlay network, a stream sharing system, the stream processing engine and a latency aware tool. The first component provides efficient data localization. The second is responsible of publishing the available pre-processed streams. The third is able to process streams, and finally, the fourth component is in charge of estimating the latency of the data movement when sharing the streams. Figure 2 presents the proposed architecture, which is detailed below.

B. Overlay Network

The DHT network is implemented using Pastry [9]; however, any other DHT, such as Chord [10], can be used in its place. Pastry is a well-known KBR (key-based routing), which provides scalable and efficient data localization. Pastry routing can efficiently locate data in a logarithmic number of routing hops $logN$, where N is the number of peers in the network. DHT-based overlays like Pastry can manage millions of participants without compromising performance, providing the substrate to build large scale systems.

Every peer in Pastry [9] is assigned a unique node ID in a space of 128-bit identifiers generated using a cryptographic hash SHA-1. The neighbors of a peer in Pastry are stored in the *leafset* that contains the L numerically closest peers, $L/2$ clockwise and $L/2$ counterclockwise. Pastry routing algorithm is a prefix-based algorithm that routes a message to the numerically closest peer of a given key k, we call this peer the *responsible* for k. The Pastry routing table stores on the n^{th} row the IP address of peers whose nodeIDs share the first n digits with the nodeID of the present peer. The algorithm forwards the messages to a peer chosen from its routing table that shares at least one more digit with the key k than the current peer. If no such peer can be found and the current peer does not know any other peer that is numerically closer

 ISBN: 978-1-61208-405-3

to k, then the current peer is responsible for k and the routing ends.

Pastry has the advantage over other DHTs of including a neighbor list, which maintains contacts to peers close in terms of a metric, for example latency. In our case this improves routing and performance.

C. Sharing Streams

Stream sharing is achieved by exploiting a Publish/Subscribe mechanism specially suited for DHT networks, called Scribe. Scribe [11] is a topic-based system built on top of Pastry [9] that creates a multicast tree, which contains all the peers subscribed to a given topic. The multicast tree is essential to notify subscribers about updates on the given topic.

Each topic is referenced to by an identifier and the Pastry node with the closest identifier to the topic becomes its responsible peer. A multicast tree is built for each topic, rooted at the corresponding responsible peer. In Scribe when a new node subscribes to a subject, its subscription is routed by Pastry to the corresponding responsible peer. The nodes in the path towards the responsible peer update the tree structure in order to include the new subscriber in a distributed manner. When an event is published for a subject or topic, a message is routed through Pastry to the peer responsible for that subject. The responsible peer is addressed by the subject's identifier.

DSPEs can publish their streams identifying them using the stream data source. Then any operator or subset of operators related to that data source will be published at the same peer. Subscribers can join the group by performing the subscribe operation using the data source of their interest.

When an event arrives at the responsible peer for a given topic, a matching process among the description files of the streams is performed in order to find streams that match the query. Then, the references to the candidates' streams are returned.

D. Stream Processing

Stream processing has generated the attention of scientific community in the last years, arising as a promissory solution to process the huge amount of data generated nowadays. Many SPEs have been developed, systems like S4 [7], Storm of Twitter [8], TimeStream [12], StreamCloud [13], SEEP [14], D-Stream [15], MillWheel [16], Kinesis [17] among others, are systems proposed to process massive data in real-time. In this work, we focus on the Apache solution, called Simple Scalable Streaming System (S4).

S4 [7] is a general-purpose, distributed, scalable, event-driven, modular platform that allows programmers to easily implement applications for real-time processing of continuous unbounded streams of data. SPEs like S4 have a graph-oriented programming model where nodes represent operators, also called processing elements (PE), and the edges represent data flows. A query defines the PEs and their interaction. PEs can filter, map, unite, aggregate data, or carry out more complex processing. PEs are the basic computational unit in S4. They consume events on the basis of keys and may generate results as events. PEs are executed on Processing Nodes (PN), which are machines in a cluster. A special type of PEs called *adapters* associate tuples with keys.

S4 can be deployed on commodity hardware achieving low latencies in communication. S4 uses a push model where events are pushed to the next PE as fast as possible. In case a PE becomes overloaded, S4 uses load shedding and, in case of failure, S4 provides state recovery via uncoordinated checkpointing, using a coordinated communication system to detect node failures and notify nodes.

E. Low Latency

In P2P systems, participants are distributed all over the world, experiencing different communication latencies. Neighbors on a DHT can have greater communication latency compared to farther located peers on the DHT.

Due to the online nature of stream processing it is essential to reach low latency responses. For this reason, is important to provide information about pre-processed data latency in order to make the decision of exploiting such information or re-processing it locally.

Applications have to meet different QoS requirements, furthermore the access to remote pre-processed streams experience different latencies. Applications must be able to evaluate the performance of using pre-processed data as an input stream. To cope with this requirement, our model provides a QoS module that estimates latency based on the Vivaldi algorithm [18].

Vivaldi [18] is an algorithm to estimate distance between peers in a fully distributed manner. It is based on the principle of spring relaxation to find minimal energy configurations in the system measuring latencies. Vivaldi presents a fully distributed lightweight algorithm that assigns synthetic coordinates to nodes in such a way that the distance between the coordinates of two nodes accurately predicts the communication latency between the nodes.

Vivaldi does not require a fixed network infrastructure or especial nodes to compute distances. Instead, any node can compute good quality coordinates by collecting latency information from only a reduced number of nodes. To collect information, Vivaldi piggybacks data on communication messages enabling traffic reduction while keeping other nodes informed on latencies experienced. The use of communication messages to spread information enables Vivaldi to scale to a large number of nodes.

Vivaldi can be applied on P2P systems in a straightforward manner. Dabek et al. have applied Vivaldi over a Chord [10] infrastructure to reach a low latency service over a P2P network [19]. Steiner and Bliersack have analyzed Vivaldi's performance [20], reinforcing Vivaldi authors' claims about the accuracy and the ability to scale of the algorithm. However, they also conclude that Vivaldi is not suitable for selecting close-by peers (within the same ISP). Round-trip time is composed of three elements: propagation delay, transmission delay, and queuing delay. On close-by peers RTT is small and become masked by the other components inducing noise to the estimation process degrading the estimation accuracy. This is not the case of our work since our scenario considers worldwide distributed SPEs.

QoS module estimates latency for the candidate streams provided by Scribe. Once latency is estimated, the stream processing layer can decide whether to exploit the remote pre-processed data or to start the reprocessing of data locally.

 ISBN: 978-1-61208-405-3

III. Processing Model

Given a DSPE requiring processing a distributed data stream, the processing model follows 4 steps:

1) **Stream Publication**
DSPEs process data streams generating an output result. This output is a stream that can be shared as pre-processed data to be exploited by other applications. The sharing process relies on publishing data about the output streams in order to allow another SPE to know if this is the data it needs.
The data about the stream to be shared is:
   - *Data source*
   - *Description of the processing*
   - *IP address of the SPE.*

   The identifier of the stream is defined by the *data source* or input stream the application receives. The Publish/Subscribe mechanism sends the stream data to a responsible peer which stores all the streams related to that same identifier or topic, using `SHA(identifier)`. Additional information or metadata could be published in order to facilitate the matching process and also to determine QoS characteristics based on the DSPE localization or bandwidth.
   Figure 3 presents an example of the publication of a pre-processed stream which identifier is the string `twitter` (data source). Scribe computes the `SHA(twitter)` and routes the data of the stream, description and IP address to the peer closest to the result of that computation. The peer uses `Scribe.publish(identifier, IP, description)` in order to publish the stream.
   The responsible peer R multicasts a message with this data to all the subscribers of the data source, to new subscribers and in case of updates.

2) **Resource Discovery**
DSPEs can locate SPEs that process the data of a specific stream subscribing the topic built for the data source. When the peer is subscribed to the data source it can receive new data about SPEs that are working on this data. `Scribe.join(identifier)` subscribes the peer to the correspondent multicast tree.
Each time a new stream is published, subscribed SPEs are notified of the updates in the topic. The node can decide locally if it is interested in one SPE output. Figure 3 shows peer P joining the group of the data source, called `twitter`, using Scribe.

3) **Data Sharing and Processing**
Data processing involves processing data either locally or exploiting pre-processed data from a remote DSPE.
A SPE that process a stream that is required by several others, builds a Scribe multicast tree with the subscribers that need this output stream (`Scribe.create(IP,data_source)`). This same node is the root of the multicast tree and the requesters use the IP address of the peer in order to subscribe to the stream. In this way, the source node does not send the stream directly to all the subscribers, but it uses the multicast tree to balance the load.
Figure 3 shows this step where the peer P joins the multicast tree of the stream generated by DSPE.

4) **Latency Estimation**
Once one stream is selected, the peer estimates the latency that the peer will experience during the retrieval of the stream from that remote DSPE. Latency is critical for online processing, however applications have different QoS requirements which should be considered. If it does not achieve the expected latency compared to the direct use of the data source, then the remote pre-processed data is discarded and the peer P should leave the DSPE group.

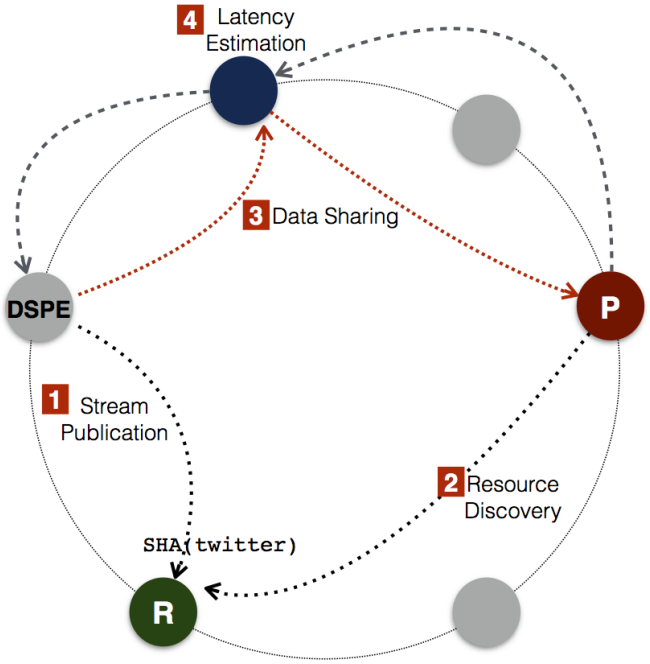

Figure 3. Distributed sharing-based model for stream processing

In Figure 3, the distributed stream sharing process is presented. The first step consists in checking the published streams. Secondly, the matching process between the query and the available streams are performed in order to select the candidate streams. Third, candidate streams are returned to the querying DSPE. Then, latency estimation takes place in order to discard or take preprocessed streams. Finally, the selected candidate stream is retrieved from the remote DSPE.

IV. Related Work

Recently, several platforms have built on top of SPEs to provide more functionality, differing from sharing stream of data among clusters of nodes. One of them is Trident [21], which is a high level abstraction on top of Storm that simplifies the process of building topologies using a micro-batching processing model. Spark Streaming [22] is a framework that similarly to Trident, that uses microbatch processing. Spark receives data from different sources and includes stateful operators to the SPE. Kafka [23] is a publish/subscribe system that provides log functionality to SPEs, which is designed for

 ISBN: 978-1-61208-405-3

real-time activity. The tuple Kafka-Storm or Kafka-Spark has been proposed in order to guarantee fault tolerance. Below, we discuss related work comparing their main characteristics with our model.

Synergy [24] is a middleware for distributed stream processing systems that uses an overlay mesh network for communication. The distributed processing systems can use the whole architecture and different processing elements can be found at different nodes. They proposed the use of a DHT structure to store and share stream, for this goal we use a Publish/Subscribe system, which allows easy localization of stream about the same topic. For QoS, Synergy uses a process called *impact projection* in order to find a candidate set for processing. We build our system using Vivaldi [18] in order to maintain locality-aware stream processing.

SBON [25] is a layer between a stream processing system and the physical network that manages operator placement for stream processing. SBON uses space coordinate distance between two nodes to represent the overhead of query placement and the cost of routing data between them. In this case, the participants in the distributed systems can place operators of its own application to other nodes in the system. This is a different scenario than the one targeted in our work. We consider that different applications are communicated through a DHT and share their streams through a Publish/Subscribe system. Applications may belong to different owners and the sharing process does not use more resources in processing.

SensWeb [26] is an infrastructure for geocentric exploration of sensor data stream, which allows sharing data streams across multiple applications. We follow this same goal, however we aim at sharing processed streams produced as output of SPEs. SensorWeb is focused on map visualization, which is achieved through a coordinator and an indexing engine. Our work is focused on distributed scalable infrastructure achieved with a DHT and Publish/Subscribe middleware.

GATES [27] is a system that uses a grid middleware for processing distributed data stream. GATES system uses Open Grid Services Architecture (OGSA) to provide self-resource discovering. Our work does not maintain grid boundaries, and follows a P2P architecture to achieve the same.

In the grid category, we also found StreamGlobe [28], a system that classifies peer as super-peers and thin-peers to be able to manage and optimize large networks. Users register subscriptions and data stream at these interfaces. The StreamGlobe scheme uses a hierarchical architecture and uses the same framework as GATES to achieve resource discovery.

Branson et al. propose CLASP [29], a middleware that enables autonomous stream analysis systems to interoperate, providing them with opportunities for data access. In CLASP, applications that seek to cooperate, build virtual organizations that formalize permissible interoperation, called common interest policies (CIP). CIP specifies resources to share and each virtual organization defines a manager, a planner and coordinator process that support collaboration functions. CLASP can complement our work, since it defines a collaboration protocol and a system association. However, CLASP does not cover the distributed infrastructure, or the efficient resource discovery.

V. Concluding Remarks

In this paper, we identified the need of a distributed infrastructure to cope with the huge amount of data stream generated by diverse streaming data sources. We propose a distributed architecture, able to manage the data stream processing in a scalable way. Our architecture relies on a DHT network in which the SPEs communicate and coordinate their actions in order to cooperate to process data. Cooperation is done by sharing pre-processed data streams based on a Publish/Subscribe mechanism.

It is well known that DHT infrastructures have to deal with changing network conditions, affecting their communication latency. We tackle this problem by providing information about access latency to the pre-processed data resources. Such information allows the remote processing engine to decide whether to exploit the remote data stream or reprocess it locally.

We have generated a prototype of the architecture proposed in this article in the context of a Fondef IDEA grant, project code CA12i10314. Our future work is mainly focused on applying this architecture on data stream processing in the context of disaster scenarios.

Acknowledgment

The authors would like to thank the University of Santiago research project PMI-USA 1204 and STIC-AmSud project RESPOND 13-STIC 11. Erika Rosas thanks to CONICYT PAI/ACADEMIA 791220011.

References

[1] "Libelium world." [Retrieved: May, 2015] http://www.libelium.com/smart_cities/

[2] S. Madden and M. van Steen, "Guest editors' introduction: Internet-scale data management," IEEE Internet Computing, vol. 16, no. 1, 2012, pp. 10–12.

[3] D. J. Abadi et al., "Aurora: a new model and architecture for data stream management," VLDB J., vol. 12, no. 2, 2003, pp. 120–139.

[4] D. J. Abadi et al., "The design of the borealis stream processing engine," in Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA, January 2005, pp. 277–289.

[5] S. Krishnamurthy et al., "Telegraphcq: An architectural status report," IEEE Data Eng. Bull., vol. 26, no. 1, 2003, pp. 11–18.

[6] A. Arasu et al., "STREAM: the stanford stream data manager," IEEE Data Eng. Bull., vol. 26, no. 1, 2003, pp. 19–26.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in Proceedings of the 2010 IEEE International Conference on Data Mining Workshops, ser. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177.

[8] "Storm." [Retrieved: May, 2015] https://github.com/nathanmarz/storm/wiki

[9] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in Middleware, ser. Lecture Notes in Computer Science, R. Guerraoui, Ed., vol. 2218. Springer, 2001, pp. 329–350.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[11] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," IEEE J.Sel. A. Commun., vol. 20, no. 8, Sep. 2006, pp. 1489–1499.

[12] Z. Qian et al., "Timestream: reliable stream computation in the cloud," in Proceedings of the 8th ACM European Conference on Computer Systems, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 1–14.

 ISBN: 978-1-61208-405-3

[13] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, 2012, pp. 2351–2365.

[14] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 725–736.

[15] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in SOSP, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 423–438.

[16] T. Akidau et al., "Millwheel: Fault-tolerant stream processing at internet scale," in Very Large Data Bases, 2013, pp. 734–746.

[17] "Amazon kinesis." [Retrieved: May, 2015] http://aws.amazon.com/kinesis/

[18] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: a decentralized network coordinate system," SIGCOMM Comput. Commun. Rev., vol. 34, no. 4, Aug. 2004, pp. 15–26.

[19] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a dht for low latency and high throughput," in Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, ser. NSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 85–98.

[20] M. Steiner and E. Biersack, "Where is my peer? evaluation of the vivaldi network coordinate system in azureus," in NETWORKING 2009, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5550, pp. 145–156.

[21] "Trident." [Retrieved: May, 2015] https://storm.apache.org/documentation/Trident-API-Overview.html

[22] "Spark streaming." [Retrieved: May, 2015] https://spark.apache.org/streaming/

[23] "Kafka." [Retrieved: May, 2015] http://kafka.apache.org/

[24] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware component composition for distributed stream processing systems," in Middleware, ser. Lecture Notes in Computer Science, M. van Steen and M. Henning, Eds., vol. 4290. Springer, 2006, pp. 322–341.

[25] P. R. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. I. Seltzer, "Network-aware overlays with network coordinates," in 26th International Conference on Distributed Computing Systems Workshops (ICDCS 2006 Workshops), 4-7 July 2006, Lisboa, Portugal. IEEE Computer Society, 2006, p. 12.

[26] L. Luo, A. Kansal, S. Nath, and F. Zhao, "Sharing and exploring sensor streams over geocentric interfaces," in 16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2008, November 5-7, 2008, Irvine, California, USA, Proceedings, 2008, p. 3.

[27] L. Chen, K. Reddy, and G. Agrawal, "Gates: A grid-based middleware for processing distributed data streams," in HPDC. IEEE Computer Society, 2004, pp. 192–201.

[28] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser, "Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures," in VLDB, K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, Eds. ACM, 2005, pp. 1259–1262.

[29] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye, "Clasp: Collaborating, autonomous stream processing systems," in Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 348–367.

 ISBN: 978-1-61208-405-3