Survey of Distributed Stream Processing

Supun Kamburugamuve, Geoffrey Fox
School of Informatics and Computing
Indiana University, Bloomington, IN, USA

1. Introduction

There is a class of applications in which large amounts of data generated in external environments are pushed to servers for real time processing. These applications include sensor-based monitoring, stock trading, web traffic processing, network monitoring, and mobile devices. The data generated by these applications can be seen as streams of events or tuples. In stream-based applications this data is pushed to the system as unbounded sequences of event tuples. Since immense volumes of data are coming to these systems, the information can no longer be processed in real time by the traditional centralized solutions. A new class of systems called distributed stream processing frameworks (DSPF) has emerged to facilitate such large-scale real time data analytics. For the past few years, batch processing in large commodity clusters has been a focal point in distributed data processing. This included efforts to make such systems work in an online stream setting. But the stream-based distributed processing requirements in large clusters are significantly different from what the batch processing systems are designed to handle. People used to run batch jobs for large-scale data analytics problems that require real time analysis due to the lack of tools, and some still run batch jobs for those applications. Large-scale distributed stream processing is still a very young research area with more questions than answers. MapReduce[1] has established itself as the programming model for batch processing on large data sets. Such clear processing models are not defined in the distributed stream processing space. As the exponential growth of the devices connected to the Internet continues, the demand for large-scale stream processing can expect to increase significantly in the coming years.

There are many frameworks developed to deploy, execute and manage event-based applications at large scale, and this is one important class of streaming software. Examples of early event stream processing frameworks included Aurora, Borealis[2], StreamIt[3] and SPADE[4]. With the emergence of Internet-scale applications in recent years, new distributed map-streaming processing models have been developed such as Apache S4, Apache Storm[5], Apache Samza[6], Spark Streaming[7], Twitter's Heron[8] and Nepture[9], with commercial solutions including Google Millwheel[10], Azure Stream Analytics and Amazon Kinesis. Apache S4 is no longer being developed actively. Apache Storm shares numerous similarities with Google Millwheel, and Heron is an improved implementation of Apache Storm to address some of its execution inefficiencies.

We can evaluate a streaming system on two largely independent dimensions. In one dimension there is a programming API for developing the streaming applications, and the other has an execution engine that executes the streaming application. In theory a carefully designed API can be plugged into any execution engine. A subset of modern event processing engines were selected in this paper to represent the different approaches that DPSFs have taken in both dimensions of functionality, including a DSPF developed in academia i.e. Neptune. The engines we consider are:

1. Apache Storm
2. Apache Spark
3. Apache Flink
4. Apache Samza
5. Neptune

We would like to evaluate these five modern distributed stream processing engines to understand the capabilities they offer and their advantages and disadvantages along both dimensions of functionality.

In this report we first identify a general processing model for distributed stream processing systems and identify key requirements of a DSPF that can be used to evaluate such a system. Next we pay special attention to the existing techniques for handling failures in a DSPF. Finally we choose a few existing stream processing system implementations and critically evaluate their architecture and design based on the requirements we have identified.

2. Stream Analysis Architecture

Usually there are hundreds of thousands of heterogeneous stream sources connected to streaming applications. Streaming data from these sources are directed to real-time applications for data analysis. In general the individual sources can come online and go offline at any point in time and can communicate using arbitrary protocols. Such heterogeneous sources are usually connected to a thin gateway layer, which understands these heterogeneous protocols and transfers the data to the streaming applications. This layer is kept as lightweight as possible to reduce the overhead and doesn't involve any heavy processing or buffering. Accepted events are then sent to message queues and routed to the correct destinations for streaming analysis. The message queuing layer acts as an event source to the streaming applications. It provides a temporary buffer and a routing layer to match the event sources and the applications.

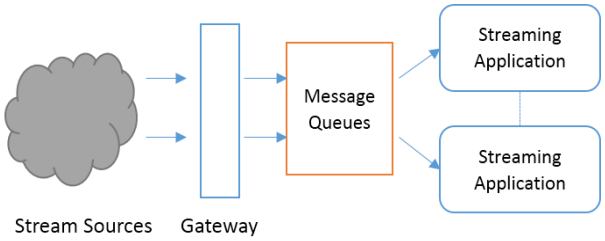

Figure 1 Streaming analysis architecture

This architecture has a strong resemblance to web architectures where stream applications are replaced by a traditional request response web application and message queues are replaced by an enterprise message bus (ESBs) [11]. The gateways match the HTTP load balancers that accept the incoming web connections.

The distributed streaming applications are deployed in DSPFs running on large clusters. The applications connect to the message queues to obtain the data. By looking at the modern distributed streaming engines, we can identify some common layers. These layers are shown in the figure below.

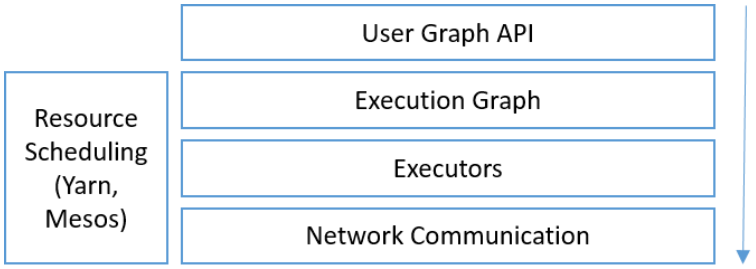

Figure 2 DSPF Layered architecture

The top layer defines the user APIs where a streaming application is defined as a graph. The events flow through the edges of this graph and processing happens at the nodes. The user graph is then converted to an execution graph and components of this are distributed to a set of workers/executors across a cluster of nodes. Network communications are stabilized between the workers to complete the graph. The bottom three layers can work with a resource management layer such as Apache Yarn[12] or Mesos[13] to acquire the necessary cluster resources to run the execution graph.

3. Message Queuing

Distributed Stream Processing has a strong connection to message queuing middleware. Message queuing middleware is the layer that compensates for differences between data sources and streaming applications. Message queuing is used in stream processing architectures for two major reasons.

- It provides a buffer to mitigate the temporal differences between message producing and message consuming rates. When there is a spike in message production, they can be temporally buffered at the message queue until the message rate comes down to normal. Also when there is a slowdown in the message processors, messages can be queued at the broker.
- Messages are produced by a cloud of clients that makes a connection to the data services hosted in a different place. The clients cannot directly talk to the data processing engines because different clients produce different data and

these have to be filtered and directed to the correct services. For such cases brokers can act as message buses to filter the data and direct them to appropriate message processing applications.

Traditional message brokers are designed around transient messages where producers and consumers are online mostly at the same time and produce/ consume messages at an equal rate. Popular message brokers such as RabbitMQ[14] and ActiveMQ are all developed around this assumption. There is an emerging class of storage first brokers where data is stored in the disk before they are delivered to the clients. The best example of such storage first brokers is Kafka[15]. Transient brokers and storage first brokers are suitable for different applications and environments. Situations in which storage first brokers are best suitable include:

1. Large amounts of data are produced in a small time window which the data processing applications cannot process during this time.
2. Consumers can be offline while large amounts of data are produced, leading to it being stored at the brokers.
3. The same data needs to be streamed to different applications at different instances for processing.

Situations in which transient message brokers are best suitable:

1. Low latency message transfer at the brokers.
2. Transactional messaging to ensure guaranteed processing across applications.
3. Dynamic routing rules at the message broker.

Both transient and storage first brokers can be used in Streaming platforms depending on the requirements of the applications. Open source transient message brokers such as ActiveMQ, RabbitMQ, and HornetMQ provide similar functionalities in the context of stream processing. We will use RabbitMQ as an example transient message broker and Kafka as a storage first broker to compare between the two types.

	Kafka	RabbitMQ
Latency	Polling clients and disk-based data storage makes it less friendly to latency critical applications.	In memory storage for fast transfer of messages.
Throughput	Best write throughput with scaling and multiple client writing to same topic. Multiple clients can read from same topic at different locations of message queue at the same time.	Single client writing to the same topic. Multiple consumers can read from the same topic at the same time.
Scalability	Many clients can write to a queue by adding more partitions. Each partition can have a message producer implying writers equivalent to partitions. The server doesn't keep track of the clients, so adding many readers doesn't affect the performance of the server.	Maintains the client status in memory, so having many clients can reduce its performance.
Fault tolerance	Support message replication across multiple nodes	Support message replication across multiple nodes
Complex message routing	No	Supports up to some level, but not to the level of a service bus.

The routing rules at the modern message brokers are primitive compared to systems such as complex event processing systems and enterprise message buses (EMB) that are at the forefront of modern web applications. Routing can be built-in to streaming applications but this introduces extra functionalities to the streaming applications which are not part of their core responsibility.

4. Stream Processing Application Model

Definition - A stream is a sequence of unbounded tuples or events of the form $(a_1, a_2, \dots, a_n, t)$ generated continuously in time. Here a_i denotes an attribute and t denotes the time.

The Stream processing model is a graph of stream processing nodes connected through streams of events. The main objective of a Distributed Stream Processing engine is to provide infrastructure and APIs necessary to create the stream processing graph and execute this graph with a stream of messages continuously. The nodes in this graph process the messages and the edges are communication links.

4.1 User Graph and Execution Graph

Usually the graph defined by the user is converted to an execution graph by the runtime engine. Figure 1 is a user defined graph for stream processing with 3 streaming processing operators connected by streams. Figure 2 shows the same graph converted to an execution graph where 2 instances of the operator S and 3 instances of operator W are running in parallel.

Figure 3 User Graph

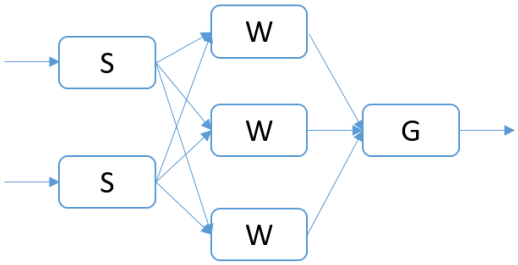

Figure 4 Execution graph

This transformation from user graph to execution graph is implemented differently in different DSPFs. In general one category of DSPFs allows the user to give direct information to convert the user graph to an execution graph. The second category of DSPFs convert the user graph from an execution graph using implicit details. The execution graph is distributed across a set of processes running in a cluster of nodes. With this information in mind we can put the functionality of any DSPF into two layers:

1. The APIs provided to create the streaming processing user graph including the streaming operators.
2. The implementation of the execution graph.

When carefully designed, these two layers can be used as separate components. For example theoretically a user graph defining the API of one DSPF can be executed over another DSPF's execution layer by providing a conversion from the user defined graph to the execution graph.

The underlying infrastructure handles how the nodes of the graph are distributed to multiple machines and how communication happens between the nodes.

The key factors for designing the execution graph are:

1. Fault tolerance
2. Message delivery guarantees
3. Throughput vs. latency
4. Resource management and scheduling of streaming operators
5. Communication between the operators

5. Graph API

The graph is abstracted in different ways in different stream processing engines. Some DSPFs directly allow users to model the streaming application as a graph and manipulate it as such. Others do not allow this function and instead give higher level abstractions which are hard to recognize as a graph but ultimately executed as such. Different DSPFs have adopted different terminologies for the components of the graph.

To further understand how these two types of APIs differ, let's take two examples from Storm and Flink.

```
1: TopologyBuilder builder = new TopologyBuilder();
2: builder.setSpout("spout", new RandomSentenceSpout(), 5);
3: builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
4: builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
5: StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
```

```
1: StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
2: DataStream<Tuple2<String, Integer>> dataStream = env.socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .groupBy(0)
                .sum(1);
3: env.execute("Socket Stream WordCount");
```

In Apache Storm the graph is directly created by the user. The operators are defined as implementations of classes. These operators are combined by streams and the APIs define how the streams connect the different operators. For example, line 3 defines an operator called split sentence with its implementation defined in the SplitSentence class and connects this operator to the spout operator using a load balancing grouping (shuffleGrouping). Also the same line instructs Storm to create 8 instances of this operator. The load balancing group means the messages will be distributed to the 8 instances from the spout operator in a round robin fashion.

In Flink the operators are connected using the data. Every operator creates a DataStream and the next operator works on this DataStream. For example the flatMap creates an instance of a DataStream and groupBy operator acts on this DataStream object. The edges of the graph are not defined by the user and are created by Flink based on the data and the operators defined on them.

The table below shows how the different DSPFs name the components of the graph and how it is represented.

	Storm	Spark	Flink	Samza	Neptune
Graph Node	Spout or Bolt	An operator on a RDD	Operator on a DataStream	Task	Stream Sources and Stream Processors
Graph Edge	Stream	Defined implicitly by the operators on RDDs	Defined implicitly by the operators on Data Stream	Kafka Topic	Links
Graph is directly created by user	Yes	No	No	Yes	Yes
Message abstraction	Tuple	RDD	Data Stream	Envelope	Stream Packet
Primary operator implementation language	Java	Java/Scala	Java/Scala	Java	Java
Name of Graph	Topology	Stream processing job	Stream Processing job	Samza Job	Stream Processing Graph

Storm and Neptune create the graph explicitly by defining the graph nodes, including how they are distributed and connected. For Spark and Flink the graph is created implicitly by the underlying engine based on the operators applied on the data.

6. **Execution Model**

The graph defined by the user is converted to an execution graph by the runtime engine. Other than Spark all the DSPF implementations distribute the nodes of the execution graph into machines running in the cluster and run them as continuous operators that react to the events streaming in through the communication links. Usually a running instance of the execution graph node is called a task. The objective of the underlying engine is to schedule these tasks across a cluster of machines and connect them via network communications.

Spark is not a true stream engine and overlays the streaming capability on top of its core batch processing engine by creating small batches of streaming data and executing them as batch programs. This model executes small batch jobs on the available resources.

The execution model has three major aspects. 1. Task scheduling 2. Task execution 3. Communication.

6.1 Task Scheduling

The runtime engine gets a resource allocation before scheduling the tasks. The resource allocation can be static or dynamic using a separate resource allocator. The most popular resource allocators are Yarn and Mesos. For a streaming system once the resources are obtained they are used by the system for an indefinite period until the streaming application is shutdown unlike a batch processing allocation which completes within a certain time.

The task scheduling is important both for handling fault tolerance and performance. For example if the tasks are sparsely distributed in a large cluster, one node failing in the cluster can have a little effect on the streaming application but the performance will be reduced because of the TCP communications required. If the tasks are densely allocated in nodes, a node failing can bring down most of the application but the performance will be high. Most streaming systems allow the user to specify custom scheduling for tasks.

6.2. Task Execution

Tasks are instances of user defined operators that run continuously reacting to events streaming in from the communication links. Most DSPFs use a worker based approach for running tasks. A worker can host multiple tasks and communications are happening between workers reducing the number of connections needed. Having multiple tasks in a single process can be troubling because a single bad behaving task can affect the other tasks. Also having tasks that does different things in one process makes it harder to reason about the performance of a topology. Heron and Samza deviates from shared task model by using a separate processes for each task.

6.3 Communication

Communications involve serializing the objects created in the program to a binary format and sending them over TCP. Different frameworks use different serialization technologies and this can be customized. The communications can do optimizations such as message batching to improve the throughput sacrificing latency. Usually the communications are peer to peer and the current DSPFs doesn't implement advanced communications optimizations. Communications can be either pull based or poll based while pull based providing the best latency. Poll based systems can have the benefit of not taking messages that cannot process at a node.

Flow control is a very important aspect in streaming computations. When a processing node becomes slow the upstream nodes can produce more messages than the slow node can process. This can lead to message build ups in the upstream nodes or message losses at the slow node depending on the implementation. Having flow control can prevent such situations by slowing down the upstream nodes and eventually not taking messages from the message brokers to process. Heron the successor to Storm provides flow control unlike in the Storm implementation.

Samza is different in communications to other DSPFs because it uses Kafka message broker for communications adding an extra hop between every edge in the graph. This has some advantages such as flow control automatically built in to it. But this added layer makes it much slower than the rest of the DSPFs.

	Storm	Spark	Flink	Samza	Neptune
Data serialization	Kryo serialization of Java objects	RDD serialization	Data Stream Serialization	Custom serialization	Java Objects
Task Scheduler	Nimbus, can use resources allocated by Yarn	Mesos, Yarn	Job Manager on top of the resources allocated by Yarn and Mesos	Yarn	Granules
Communication framework	Netty	Netty	Netty	Kafka	Netty
Message Batching for High throughput	Yes	Yes	Yes	Yes	Yes
Flow control	No	Yes	Yes	Yes	Yes
Message delivery	Pull	Pull	Pull	Poll	Pull

The difference between Spark and Flink is that Flink can work as a native stream processing engine and Spark uses batching to mimic streaming computations. As far as underlying implementation goes, Flink and Storm are similar but the data abstractions remain different. Storm and Neptune appear to be architecturally similar although Neptune tries to avoid some of the bad designs of Storm by using multiple queues and threads in a worker for message processing and improving some of the inefficiencies of Storm such as object creations. As mentioned in several places Heron is an improved version of Storm where it implements pluggable scheduling to resource managers, task isolation using processes and flow control. Heron has much better performance compared to Storm.

7. Processing Guarantees

In large-scale distributed computing, failures can happen due to node failures, network failures, software bugs, and resource limitations. Even though the individual components have a relatively small probability of failure, when large sets of such components are working together the probability of one component failing at a given time is present and in practice failure is the norm rather than the exception. For a streaming framework, latency is of utmost importance and it should be able to recover fast enough so that normal processing can continue with minimal effect to the overall system.

Stream processing engines in general provide three types of processing guarantees. They are 'exactly once', 'at least once' and 'no guarantee'. These processing guarantees are related to the fault recovery methods implemented by the system. These recovery methods have been categorized as Precise Recovery, Rollback Recovery and Gap Recovery[16]. In precise recovery there is no evidence of a failure afterwards except some increase in latency, hence this provides 'exactly once' semantics. In Rollback recovery the information flowing through the system is not lost but there can be effects to the system other than an increase in latency. For example, information may be processed more than once when a failure happens. This means rollback recovery can provide 'at least once' semantics. In Gap recovery the loss of information is expected and provides no guarantees about message processing.

There are several methods of achieving processing guarantees in streaming environments. The more traditional approaches are to use active backup nodes, passive backup nodes, upstream backup or amnesia. Amnesia provides gap recovery with the least overhead. The other three approaches can be used to offer both precise recovery and rollback recovery. All these methods assume that there are parallel nodes running in the system and these can take over the responsibility of a failed task.

Before providing message processing guarantees, systems should be able to recover from faults. If a system cannot recover automatically from a fault while in operation, it has to be manually maintained in a large cluster environment, which is not a practical approach. Almost all the modern distributed processing systems provide the ability to recover automatically from faults like node failures and network partitions. Now let's look at how the five frameworks we examine in this paper provide processing guarantees.

	Storm	Spark	Flink	Samza	Neptune
Recover from faults	Yes	Yes	Yes	Yes	No
Message processing guarantee	At least once	Exactly once	Exactly once	At least once	Not available
Message guarantee mechanism	Upstream backup	Write ahead log	Check-pointing	Check-pointing	Not available
Message guarantee effect on performance	High	High	Low	Low	Not available

Flink provides the best recovery while Neptune has no fault recovery implemented at the moment.

8. Streaming applications

We have identified several types of events and their processing requirements that can present different challenges to a stream processing framework.

1. Set of independent events where precise time sequencing is unimportant, e.g. independent search requests or smartphones or wearable accessories from users.
2. Time series of connected small events where time ordering is important, e.g. streaming audio or video; robot monitoring.
3. Set of independent large events where each event needs parallel processing with time sequencing not being critical, e.g. processing images from telescopes or light sources with material science.

4. Set of connected large events where each event needs parallel processing with time sequencing being critical, e.g. processing high resolution monitoring (including video) information from robots (self-driving cars) with real time response needed.
5. Stream of connected small or large events that need to be integrated in a complex way, e.g. streaming events being used to update models (clustering) rather than being classified with an existing static model which fits category a).

Apart from supporting these events and processing requirements, DSPFs can provide high level application functions integrated at the core level to provide better functionality to users. Time windowing, handling of multiple sources and parallel execution are such high level functionalities we have identified.

8.1 Multiple sources

For streaming applications, events are generated by many sources and the event processing order for each source is important in use cases 2 and 4. A streaming application usually handles events from many sources within a single application. When events from multiple sources are connected, the streaming application has to keep track of the event source by having internal states about the processing for that source. Modern engines don't support automatic isolation of event sources within a single application and the user has to implement such capabilities on their own. It is important to have multiple source capabilities at the framework level because user level handling of such sources doesn't account for functionalities like:

- Automatic scaling of the resources when increasing/decreasing the number of sources
- Scheduling of resources for the applications with respect to the number of sources

8.2 Time windows

Streaming processing in time windows[17, 18] is a heavily used approach for streaming data processing. Having sliding time window functionality as a first class citizen in a streaming framework can greatly benefit the application. Google MillWheel and Flink provides windowing as a built in functionality at the DSPF layer. With other engines, the user needs to define the time windowing functionalities in the applications.

8.3 Parallel Execution

To process events requiring parallel processing it is important to have parallel processing communication primitives such as group communications among the parallel tasks. To the best of our knowledge none of the streaming engines provide APIs to do parallel processing of an event at the task level. To enable parallel processing of an event, synchronization primitives and communication primitives must be implemented at the parallel task level.

	Storm	Spark	Flink	Samza	Neptune
Time Window	None	Batches are for time windows	Yes	None	None
Multiple Source scaling	None	None	None	None	None
Parallel Processing primitives	None	None	None	None	None

9. Discussion and Future Directions

Distributed stream processing is rapidly evolving with more and more cloud services moving from batch to real time data analytics. The distributed stream processing has evolved from early systems such as Aurora and Borealis to modern systems like Flink and Heron. Flink and Heron provides the best performance combined with fault tolerance and processing guarantees.

The graph model of stream processing has been accepted by the community and adopted by all the frameworks. Two main programming paradigms have evolved to create this graph model as discussed in the earlier sections, both having advantages and disadvantages over each other.

The popular execution model is to distribute the operators in to a multiple nodes and connect them by communication links. Most modern systems have adapted this model except Spark which is not a true streaming system. The implementation details of how the tasks are distributed and executed and the way communication links are created differs in different frameworks each providing their advantages and disadvantages. Flow control is a major aspect of any streaming framework. Having a high performance streaming engine along with flow control is challenging and these are discussed in the Heron[8] architecture. We believe more work has to be done to get better flow control of messages. There is much work that needs to

be done in scheduling of streaming tasks. Scheduling can affect performance of a streaming analytics application depending on the task work load, weather tasks are scheduled in different machines etc. Having adaptive scheduling policies that can change during run time can be a great addition. Heron has switch to a model where each task is given a separate process instead of the Storm model of having multiple tasks in a single task. Different types of applications may benefit from both these models of execution. Different frameworks have adopted different methods to achieve message processing guarantees with different algorithms for achieving the guarantees. Having fast communications within the tasks running in a node such as shared memory communications and optimizations for fast network transports such as InfiniBand[19] can improve the performance of streaming applications and these have to be investigated.

Rich APIs on top of the basic streaming graph model is lacking in modern streaming engines. Flink and MillWheel have implemented time windowing functionality on to its APIs. More research has to be done to integrate rich communications APIs to allow parallel processing and multiple source scaling especially for applications where some state needs to be maintained at the streaming frameworks. SQL on top of streaming frameworks are an interesting direction to provide interactive querying of data.

At the moment there are no standard libraries available for solving common distributed streaming problems. Such libraries can greatly help to accelerate the development of applications on top of streaming frameworks. To facilitate such libraries it is important work on universal APIs for streaming applications which require a community effort.

10. Conclusions

We have identified the architecture for streaming applications and the required software systems for scalable processing of large stream sources. We looked at the role of message brokers in streaming processing, then examined the streaming applications by considering five modern distributed stream processing systems. We first identified the general processing model for streaming processing as a graph execution model. We divided the functionality of a streaming engine into clear layers and evaluated the DSPFs according to functionality in each of the areas. There is more work that needs to be done in areas such as communications, execution and scheduling. Also higher level abstractions are required on top of the graph APIs to provide better functionality to the application developers.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107-113, 2008.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, *et al.*, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005, pp. 277-289.

[3] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*, 2002, pp. 179-196.

[4] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: the system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1123-1134.

[5] Q. Anderson, *Storm Real-time Processing Cookbook*: Packt Publishing Ltd, 2013.

[6] A. S. Foundation. Available: http://samza.apache.org/

[7] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, 2012, pp. 10-10.

[8] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, *et al.*, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239-250.

[9] T. Buddhika and S. Pallickara, "NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments."

[10] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, *et al.*, "MillWheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, pp. 1033-1044, 2013.

[11] D. Chappell, *Enterprise service bus*: " O'Reilly Media, Inc.", 2004.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, p. 5.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *NSDI*, 2011, pp. 22-22.

[14] A. Videla and J. J. Williams, *RabbitMQ in action*: Manning, 2012.

[15] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011.
[16] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, pp. 779-790.
[17] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Moment: Maintaining closed frequent itemsets over a stream sliding window," in *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*, 2004, pp. 59-66.
[18] B. Babcock, M. Datar, and R. Motwani, "Sampling from a moving window over streaming data," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002, pp. 633-634.
[19] I. T. Association, *InfiniBand Architecture Specification: Release 1.0*: InfiniBand Trade Association, 2000.