Survey of Distributed Stream Processing for Large Stream Sources

Supun Kamburugamuve
For the PhD Qualifying Exam
12-14-2013

Advisory Committee
Prof. Geoffrey Fox
Prof. David Leake
Prof. Judy Qiu

Table of Contents

1. Introduction

There is a class of emerging applications in which, a 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 so on. 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 sequence of event tuples. Since large volume of data is coming to these systems, the information can no longer be processed in real time by the traditional centralized solutions. A new class of applications called distributed stream processing systems (DSPS) has emerged to facilitate such large scale real time data analytics. For the past few years batch processing in large commodity clusters has being a focal point in distributed data processing. There has being efforts to make such systems work in an online stream setting [**1**]. But the stream based distributed processing requirements in large clusters are significantly different from what the batch processing system are designed. People used to run batch jobs for large scale data analytics problems that require real time analysis due to the lack of tools and still people run batch jobs for those applications. The large scale distributed stream processing is still a very young research area with lot of questions than answers. MapReduce [**2**] 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, the demand for large scale stream processing can expect to grow significantly in the coming years and it is interesting to know what these sources are.

Stream Sources

According to Cisco [**3**], there are about 11 billion things connected to the Internet by 2013. Many studies done by different groups have projected that huge numbers of devices will connect to the Internet by 2020 [**4**] [**5**]. The number of devices projected by these groups for 2020 is around 25 billion to 50 billion. Traditional PCs and Smartphones will only fill about half of this number and rest of it will come from smart sensors and devices connected to the Internet. The estimated world population by that time will be around 7.5 billion people. Most of these smart devices will have machine-to-machine communications to enhance the living conditions of the human beings. There are many application areas where data collected by sensor data can be used for various purposes.

Sensors are used often to track the behavior of objects. Insurance companies provide devices to track the driving patterns of the vehicle owners and adjust the insurance prices accordingly are a good example. Also user generated events can be used to track the behavior of users. A good example is Netflix movie recommendations where Netflix tracks the behavior of the user and provides intelligent recommendations based on the user patterns. Another use of sensors is to monitor the environment to gain more situational awareness. A large number of sensors deployed for monitoring an environment can give decision makers real time information to take the best decisions as possible. Best example is a combination of surveillance cameras, motion detection sensors and audio sensors giving security personal the ability to monitor suspicious activity. Also large shipping companies are using sensors to monitor the traffic patterns to constantly re-route the vehicles to provide efficient transportation. The information gathered using sensors could be used to take optimized decisions in the longer term. For example information gathered from patients over time using the sensors could help the doctors to diagnose the illnesses accurately and do customized treatments that suits the specific patients.

Factories are employing sensors to optimize their process lines by taking inputs from their process lines, analyzing them and adjusting the processes according to the results in real time. Sensors can be used to monitor the resource consumptions and optimize the resource production according to the consumption demands and patterns. For example smart grid projects monitor the energy consumption of the users in real

time to adjust the energy production accordingly. Using sensors to monitor the environment and taking highly important decisions is another key growing area. New cars are equipped with sensors to automatically brake in certain situations to avoid collisions.

These are some of the areas and use cases where sensors and event streams are expected to grow in the future. All the above cases when deployed at mass scale will produce very large amounts of data. Some of the data produced must be processed real time and decisions must be made in real time. Some applications require data to be stored and analyzed by the batch processing systems to get a deeper understanding and discovering of patterns. Also we can envision use cases where the preliminary processing is done in real time and batch processing seeded by the these preliminary analysis of the data.

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

2. Stream Processing Model

Definition - A stream is a sequence of unbounded tuples 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.

Stream processing involves processing data before storing and Hadoop like batch systems provide processing after storing. A processing unit in a stream engine is generally called a processing element (PE). Each PE receives input from their input queues, does some computation on the input using its local state and produce output to their output queues. Stream processing engine creates a logical network of stream processing elements connected in a directed acyclic graph (DAG). An example can be seen in figure 1. A PE is a node in the graph and inputs to a PE and output of a PE are modeled by edges. Tuples of a stream flow through this graph and PEs consume these tuples and emit tuples. A PE does not necessarily emit tuples in response to an incoming tuple. A PE can choose not to emit a tuple or it can choose to emit a tuple in a periodic manner or it can choose to emit tuples after it saw N input tuples. The communication between PEs can happen according to push based or pull based messaging. Each PE executes independently and only communicates through messaging with other elements. Set of PEs running in parallel are not synchronized in general and can run in any speed depending on the available resources.

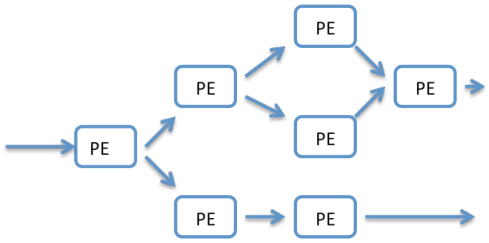

Figure 1 Logical Organization of PE DAG

In the DAG in figure 2, there are three processing elements; each one is replicated once to achieve fault tolerance. PE_1 has two inputs and one output. PE_2 and PE_3 have only one input and one output. Inputs and outputs are connected with a PE using input queues and output queues. If messages flow from PE_1 to PE_2, PE_1 is called upstream node of PE_2 and PE_2 is called a downstream node of PE_1. For the above relationships to

hold PE_1 and PE_2 don't have to adjacent vertices in the graph and child of a PE and parents of a PE are qualified for the above relationships. The actual distribution of PEs and queue management can be quite different in different implementations.

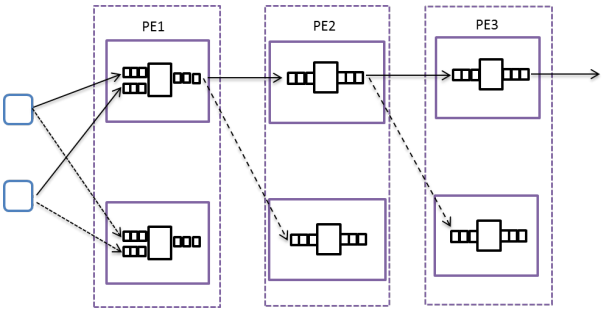

Figure 2 PE Distribution

The PEs connected in a DAG is the logical view of the distribution of processing elements in the system. When such DAG is deployed in a distributed system, each PE can be distributed in parallel among the computing nodes to achieve high throughput and redundancy.

2.1 Physical Distribution of PEs

Two types of distributed stream processing engines can be seen in practice. In the first type there is a central server that orchestrates the distribution of the PEs in to physical nodes. This node is responsible for load balancing the tasks amongst the nodes; schedule the tasks to run on different nodes. In the other type of processing engines there is no central server to orchestrate the work among the nodes and nodes work as fully distributed peer-to-peer system sharing the work equally. In both models a single PE can be running in parallel on different nodes. Now let's look at a possible physical layout for two nodes logically connected as in the following diagram.

Figure 3 Two Logically Connected PEs

Now assume PE1 involves heavy processing and must be deployed in 4 nodes in parallel and PE2 doesn't involve as much work as PE1 and must be run in 2 nodes in parallel. The physical distribution of the PEs is shown in figure 3. The parallel execution of PE1 and PE2 can be seen as two separate logical units. These two units can communicate with each other using different communication patterns.

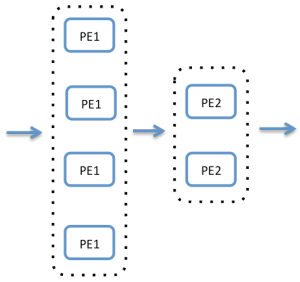

Figure 4 Physical Distribution of PEs

2.2 Stream Processing Engine Requirements

The requirements of a good stream-processing framework are revolving around two important attributes; latency of the system and the high availability of the system. It is important to lower the latency and increase the high availability, but these are competing requirements and one cannot be improved without sacrificing the other. We identify some of the expectations of a distributed stream-processing engine [**6**].

Data mobility – The data mobility is the term used to identify how the data moves through the processing elements of a SPE. High data mobility is a key for maintaining low latency. Blocking operations and passive processing elements can decrease the data mobility.

High Availability & Data processing guarantees – High availability by recovering from failures is critical for a DSPE. Because of the strict requirements on the latency, recovery should be fast and efficient. Processing guarantees like exactly once, at least once or no guarantees are all used by the DSPEs. Processing guarantees are tightly couples with the high availability algorithms and the nature of the data processing operators. Algorithms like active backup, passive backup and upstream backup are used by the DSPEs to provide data processing guarantees amidst the failing components.

Data partitioning – Data should be partitioned and handled in parallel for large volumes of data. The partitioning strategies are a key distinctive feature among the different processing engines. The partitioning algorithms affects how the system handles the data in parallel and how the system can scale.

Data Querying – High-level languages and tools for querying stream networks reduce the development cycles required to develop applications on top of stream processing engines. SQL is a widely used standard for developing such queries over data and can be adapted to process time series data in streams. The standard data processing languages promote the wide adoption of the technologies and development of standard algorithms on top of DSPEs.

Deterministic or Non-Deterministic processing – Deterministic processing networks provides the same output when presented with the same input independent of the other factors. When processing elements are non-deterministic the recovery from failures become difficult. On the other hand deterministic operations limit the scope of processing capabilities.

Data storage – Data can be stored for later reference and these stored data can be sent through the stream processing engines for historical data analysis. The ability to process historical data allows the algorithm designers to run their algorithms on historical data for verification and tuning purposes.

Handling Stream Imperfections – The messages in a stream can be delayed, can come out of order, can be duplicated or lost. DSPEs can provide functionalities to handle such imperfections by using delayed processing, storing the data.

3. Fault Tolerance

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 set of such components are working together the probability of one component failing at a given time is non-negligible and in practice failure is the norm rather than the exception. Large-scale batch processing systems like Hadoop has a high latency for operations and this is expected in such systems and delaying a computation because a failure has happened is not considered critical. In Hadoop the expectation is the correct calculations without any loss of information amidst the failing components and to achieve this latency is sacrificed. For a streaming framework latency is of utmost importance they should be able to

recover fast enough so that the normal processing can continue with the minimal effect to the overall system. The recovery methods for stream processing engines range from recovery with no effect to the system without any information loss to recovery with some information loss. Recovery without any information loss and recovery with the minimum latency are at the two extremes of the recovery methods.

These recovery methods have being categorized as **Precise Recovery**, **Rollback Recovery** and **Gap Recovery** [**7**]. In precise recovery there is no effect of a failure visible except some increase in latency. In Rollback recovery the information flowing through the system is not lost but there can be effects to the system other than increase in latency. For example information may be processed more than once when a failure happens. In Gap recovery the loss of information is expected and it provides the weakest guaranteed processing.

There are several methods of achieving fault tolerance 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 least overhead. Other three approaches can be used to provide both precise recovery and rollback recovery. There is some recent work done on stream processing to provide recovery by keeping lineage information about the data products. All these recovery methods except amnesia assumes that there are parallel nodes running in the system and these can take over the responsibility of a failed task.

We will look at the failures from the task level rather than the node level. Task level is more general than node level because a task can fail without the node failing due to communication link failures, resource limitations etc. In the following discussion a task is equivalent to a PE.

3.1 GAP Recovery

In Gap recovery when a task fails another task takes over the operations of the failed task. The new task starts from an empty state and starts processing the inputs directed to it by the upstream processing tasks. Because the new task starts from an empty state, tuples can be lost during the recovery phase. This type of recovery method is named as amnesia.

3.2 Rollback Recovery

As stated earlier Rollback recovery can be achieved using Active Standby, Passive Standby and Upstream backup.

Passive Standby

In this recovery approach the primary task saves its state periodically (checkpoint) to a permanent shared storage and this information is used to create the state of the secondary when it takes over. The secondary receives information about the state of the primary as well as the output queues of the primary. In this approach secondary is always behind the primary. Also the upstream node stores the tuples in its output queue until the tuples are stored in the secondary. When a tuple is stored in the secondary safely, secondary notifies the upstream node about it and then only upstream node deletes those messages.

When a failure happens the secondary takes over and sends the messages saved in its output queues to downstream nodes. Then it asks the upstream nodes to send the messages that it has not seen. Because the secondary is always equal or behind the primary, secondary can send duplicates to the downstream nodes when it takes over. Because of this passive standby provides repeating recovery for deterministic networks and divergent recovery for non-deterministic networks.

Active Standby

At a given time both primary and secondary nodes are given the same input and these two nodes run the same PE in parallel. The primary PE's output is connected to the downstream nodes and the backup node's output is not connected. Once a failure occurs the backup take over the operation and its output is connected to the downstream nodes.

The biggest challenge in Active Standby is in managing the output queue of the backup nodes. Active node and backup node both process data on its own speeds. The primary receives ACKs from the downstream nodes when the tuples are safely stored in the downstream nodes. These tuples are deleted from the output queues of the primary and queue trimming message indicating the messages that secondary can safely remove is sent to the secondary. After getting this message secondary deletes messages that are already seen by the downstream nodes from its output queue. Active standby provides repeating recovery for deterministic networks and divergent recovery for non-deterministic networks.

3.3 Upstream Backup

In upstream backup nodes store the tuples until the downstream nodes acknowledge them. In upstream backup the handling of output queues become complicated because tuples can be consumed by multiple downstream node and because of the non-deterministic nature of the operators. Upstream backup provides repeating recovery for deterministic networks and divergent recovery for non-deterministic networks. From the above three approaches the upstream backup provides the weaker guarantee with lower operational costs and lower network overhead.

3.4 Precise Recovery

Precise recovery can be achieved by modifying the algorithms for rollback recovery. In passive backup scheme after a failure occurs the secondary can ask the downstream nodes for the latest tuples they received and trim the output queues accordingly to prevent the duplicates. For Active backup with deterministic operators the same procedure can be applied. In case of non-deterministic operators, Active backup is complicated to handle failure. Measures have to be taken to make sure both active node and backup node is at a consistent state.

4. Distributed Stream Processing Engines

4.1 Aurora

Aurora [**8**] [**9**] is an early stream processing system developed by Brown University and MIT that has the modern distributed stream processing network structure. At first Aurora was designed as a single site stream-processing engine. Later the Aurora functionality is combined with the Medusa [**10**] system to achieve distributed processing.

Architecture

Aurora employs a DAG based processing model for streams. The model is similar to the generic stream-processing model introduced earlier. Boxes denote the processing elements in Aurora and the edges are the streams. The processing model of Aurora essentially represents a DAG bases workflow system. A stream is viewed as an unbounded set of tuples. Tuples arrive at the input queues of box processor and a scheduler selects which boxes to run. Each aurora box is a stream operator specified in the Aurora query model. After the processing is done the output is again moved to the input queue of the next box in the processing graph. Finally output is presented to the applications. So Aurora employs a push based model for data movement between the boxes and applications.

Aurora employees network queries to process the streams. These queries are called Stream Query Algebra (SQuAl) and is similar to SQL but with additional features like windowed queries. The queries can run continuously on the incoming streams or they can run on stored historical data at the Aurora connection points or by the storage manager. Connection points are the places where Aurora boxes connected together and can be used to add boxes to the processing model dynamically.

Aurora can persist data at the input queues if the data doesn't fit in the memory or it can persist the data to support queries that require long list of historical data. If the system gets overloaded Aurora can shed the load by randomly dropping tuples or by choosing a semantic dropping strategy where it does some filtering before dropping. For example less important tuples can be dropped.

Fault Tolerance
At first Aurora was not designed to support fault tolerance and later with the addition of Medusa, the model is extended to support fault tolerance by upstream backup method [**11**].

4.2 Borealis

Borealis [**12**] is the next version of Aurora stream processing system. Borealis is a distributed stream processing engine which has the core stream-processing model of Aurora and distributed functionality of Medusa. As in Aurora the Borealis stream-processing network is a DAG with boxes representing processing elements and edges representing streams of messages between the boxes. On top of the Aurora functionality Borealis provides few advanced functionalities.

Dynamic revision of query results – In certain stream applications, corrections or updates to the previously processed data is sent to the stream engines. Also data may arrive late and may miss the processing window. Borealis can use these data that are available after processing to correct the results.

Dynamic Query modifications – The Borealis processing elements have command inputs for modifying the behavior of a query during the runtime.

Architecture
Borealis has a fully distributed architecture consisting of Borealis servers running in different sites and these sites are interconnected to take coordinated actions. There is no central controller for the Borealis sites. Each Borealis server runs a query processor, which executes the actual query diagram. There is an admin module in each server that determines whether the queries should run locally or remotely. Each query processor has a box processor that runs the individual processing elements, a storage manager for storing the streams, load shedder for shedding load in case of overloading situations and a priority scheduler that determines the order of box operations depending on the tuple priorities. Each node has a Neighborhood optimizer that uses the local information and remote site information to balance the load among the sites. The High Availability modules at each site monitor each other and take over in case of node failures.

The data movement between different query processing units is done using a pull-based strategy. If the downstream node is overloaded the load can be shredded at the downstream nodes. To avoid load shedding each node tries to offload the work to other nodes if the node is getting overloaded.

High Availability
Borealis uses a variation of active replication (active standby) of the query network for fault tolerance [**13**]. Every node receives heart beat messages from the upstream nodes indicating their availability. The replicas are kept at consistent state with each other by using a technique called *SUnion*. The operations running inside the Borealis processing elements are deterministic but when a PE receives input from two upstream nodes and take the union of the two inputs the, the resulting message order can become inconsistent across replicas

due to timing. Borealis nodes collect the incoming messages in to buckets until markers are received from upstream nodes, which indicate an end of a data collection. After the markers are received the nodes sort the messages according to timestamp values. This operation is named as *SUnion* it can keep the replicas at a consistent state sacrificing the data mobility.

When an upstream node failure is detected, the downstream node chooses a replica of the upstream node to get the missing inputs. If it cannot find a suitable replica it can wait, continue processing without the specific failed input.

4.3 Apache Storm

Apache Storm [**14**] is a real time stream-processing framework built by Twitter and now available as an Apache project.

Design Goals of Storm: Guaranteed message processing is a key goal in design of Storm. A message cannot be lost due to node failures and at least once processing is a design goal. Robustness of the system is critical to the

Architecture

There are three sets of nodes in a Storm cluster and they are Nimbus node, ZooKeeper nodes and Supervisor nodes. Nimbus is the main server where user code has to be uploaded and Nimbus distributes this code among the worker nodes for execution. Also Nimbus keeps track of the progress of the worker nodes so that it can restart the failed computation or move the tasks to other nodes in case of node failures. The set of worker nodes in the Storm cluster runs a daemon called Supervisor. The coordination between supervisor nodes and the Nimbus happens through the ZooKeeper. The message flow in the system is done using ZeroMQ [**15**] based transport or Netty [**16**] based transport. The transport layer is pluggable.

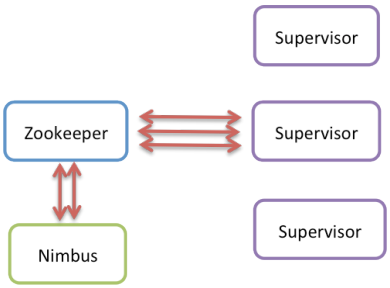

Figure 5 Storm Architecture

Programming Model

Storm doesn't try to fit a specific programming model like MapReduce on top of streams. Storm programming model provides distributed stream partition among the processing nodes. Each processing element process the input as it processes the whole stream. Storm programming model consists of Spouts, Bolts, Topologies and Streams. The Spouts and Bolts are arranged in a DAG called a Topology. A user submits a topology to a Storm cluster to execute. Stream is a set of tuples and these tuples can be a user-defined type or a system defined type. Spouts are the stream sources for the topology. Bolts consume events and emit events after processing. Storm topology starts with a set of spouts and the rest of the layers of the topology consist of Bolts. User can write the storm spouts and bolts in different programming languages like python, java or clojure. A storm job is configured using the Java programming language as a topology object and the storm client is used to submit the job to the Nimbus.

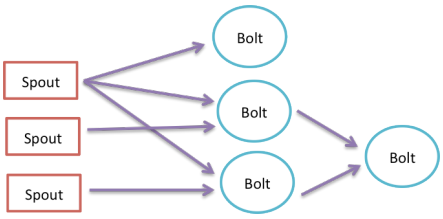

Figure 6 Storm Topology

Data Flow

Stream events are injected into a topology from a spout. A spout can be listening to a HTTP port or it may be pulling messages from a queue. When the topology is created it can specify how many tasks to run in parallel for a spout or a bolt. We can view the Spouts and bolts as logical execution units and tasks as the physical instances of the spouts and bolts. The tasks execute the Spout code or Bolt code in parallel in different nodes. A bolt's input is connecting to the output of a spout or another bolt.

Bolts pull messages from the upstream processing units. This model makes sure that the bolts will never take excess amount of messages that it cannot process. So in Storm only place where messages shredding happens, because the system is running at the full capacity is when they are processed by the spouts which is the beginning of the message origination to the system.

A bolt can have multiple tasks executing the bolt code. This bolt can be connected to another bolt which may be executing with multiple tasks in parallel. When these two bolts are logically connected the messages from the task belonging to the sending bolt can flow to task belonging to the receiving bolt according to some message grouping rules. The simplest of such rules is the Shuffle Grouping. In shuffle grouping the messages between the tasks are delivered such that each task will get an equal number of messages. Another grouping is fields grouping where messages are keyed on an attribute and messages with the same value for the chosen attribute are going to the same task. In all groupings every message is going to the every task of the receiving bolt. There are other groupings like Direct Grouping, Local Shuffle Grouping, Non Grouping and Global Grouping. These groupings can be used based on the application requirements.

Fault Tolerance

Storm employs a variation of upstream backup algorithm for fault tolerance. In Storm spouts keep the messages in their output queues until they are being acknowledged. The acknowledgement happens after the successful processing of an event by the topology. If an acknowledgement comes for a message within a reasonable amount of time spouts clear the message from output queue. If an acknowledgement didn't come within a predefined period (30 second default) the spouts replay the message again through the topology. This mechanism along with the pull based approach in bolts guarantees that the messages are processed at least once inside Storm.

The node failures are handled by the Nimbus. The Supervisor nodes send heartbeats to nimbus periodically. If nimbus doesn't receive the heartbeats in a timely fashion from a supervisor it assumes that the supervisor is no longer active and move the workers in that failed supervisor to another supervisor node. Message failure and node failure are taken as two orthogonal events because message failure can happen due to other reasons like software bugs, intermittent network failures. Because of this handling of failed messages and moving of workers to other nodes in case of node failures are done in two separate ways without any correlation between the two. This design makes the system more robust for failures.

Storm Query Model
Trident [**17**] is the high level abstraction for creating Storm topologies. Trident provides high-level data operators like joins, filters, aggregators, grouping and functions. Trident has a Java programming language based API for user to configure the system. A high level abstraction for Trident is still not developed. At the runtime the Trident jobs are converted to storm topologies and deployed on the cluster.

4.5 Apache S4

S4 [**18**] stands for Simple Scalable Streaming System and it was developed by Yahoo and donated to Apache Software Foundation in 2011.

S4 is a fully distributed real time stream processing framework. It employs the Actors model for computations. The processing model is inspired by MapReduce and uses key based programming model as in MapReduce.

S4 Architecture
S4 creates a dynamic network of processing elements (PEs) and these are arranged in a DAG at runtime. PEs are the basic computational elements in S4 and it is user defined code specifying how to process events. Instances of PEs are created and arranged in a DAG structure at runtime. A runtime instance of a PE is identified using the PE's code and configuration, the events consumed by the PE, key attributes for the events, and value of the keyed attributes.

A new instance of a PE is created for different values of the key attribute. PE can output events to other PEs in the system. There are special classes of PEs that are not keyed on any attribute. Such a PE usually take external inputs into the S4 framework. One of the biggest challenges in PE architecture described above is that, for key attributes with very large value domains there can be large number of PEs created in the system at a given time. So there must be intelligent mechanisms to discard the PEs to prevent the system from becoming unusable.

The PEs in the S4 system are assigned to Processing Nodes. Processing nodes are logical hosts to the PEs and run the PEs by accepting input events and dispatching them to PEs running on them. The events in the S4 are routed to Processing Nodes by doing a hashing function on the key attribute values of events. So events with the same value for the key attributes are always routed to the same processing node.

The coordination between the processing nodes and the messaging between nodes happens through a communication layer. This layer provides the functionality to failover the nodes in case of failures. The actual underlying stream messaging is pluggable providing flexibility for user to write their network protocols if required. The coordination between the nodes is done through Apache ZooKeeper.

Fault Tolerance
When a S4 node fails the system detects the failure using ZooKeeper and distribute the tasks assigned to the failed node to other nodes. S4 doesn't give message delivery guarantees in case of failures and uses Gap recovery for recovering from failures. Snapshots of the state of the processing nodes are saved time to time and these are used to create a new instance of a Processing node when one fails. Also because of the push model of events in the system, S4 can drop events due to high load. Because of these two reasons event lost is possible in the system.

Programming Model
S4 applications are written using the Java programming language. The stream operators are defined by the user code and can be deterministic or non-deterministic. The configuration for a job is done through the

Spring Framework based XML configuration. A job configuration consists of the definition of the included PEs and how they are configured.

4.5 Apache Samza

Apache Samza [**19**] is a stream-processing framework developed by Linkedin and donated to Apache Software Foundation.

Samza Processing Model

Samza message stream is an immutable unbounded collection of messages of same type. A stream can be read by many consumers in the system and messages can be added to a stream or deleted from a stream. Streams are always persisted by the Samza in its brokering layer. A job in Samza is a logical collection of processing units that act on a stream of messages and produce output streams. A job creates the network topology that processes the messages. Streams are partitioned into sub streams and distributed across the processing tasks running in parallel. A partition of a stream is a totally ordered sequence of messages. A task is the processing element in the Samza network. A task act on one partition of a message stream and produce a message stream. A task can consume multiple partitions from different streams.

Samza Architecture

Samza relies on Apache Yarn [**20**] for the distributed resource allocation and scheduling. It uses Apache Kafka [**21**] for the distributed message brokering. Samza provides an API for creating and running stream tasks on a cluster managed by Yarn. In this cluster Samza runs Kafka brokers and Stream tasks are consumers and producers for Kafka streams. Kafka provides a distributed brokering system with persistence for message streams. The system is optimized for handling large messages and provides file system persistence for messages. In Kafka a stream is called a Topic and topics are distributed across the brokers using a partitioning scheme. The partitioning of a stream is done based on a key associated with the messages in the stream. The messages with the same key will always belong to the same partition. Samza uses the topic partitioning of Kafka to achieve the distributed partitioning of streams. A Samza task can be producing messages to a Kafka topic or can be consuming messages from a Kafka topic.

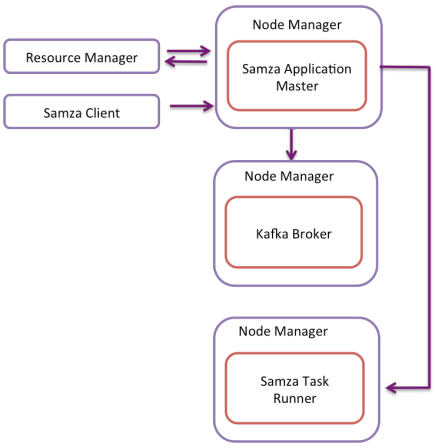

Figure 7 Samza Architecture on Apache Yarn

In Samza architecture there is a message broker between two adjacent message processing tasks along the same path. A task puts message to the broker's stream partition and broker persists the message to the file system. A downstream task poll the message broker to receive the messages. By default Kafka stores all the messages in the file system and doesn't delete them for a configured amount of time. This allows the downstream tasks to consume messages at arbitrary points if they need to.

Fault Tolerance

Samza provide at least once message delivery guarantees using upstream backup techniques for handling message failures. When a node fails and new node takes over, the new node start reading from the upstream broker topic from the maker that the failed node set. Because of this Samza achieves repeating recovery for deterministic networks and divergent recovery for non-deterministic networks. The upstream backup recovery method works only for task level failures in Samza. If a broker node fails, Samza loses messages persisted in the file system and these cannot be recovered. There has been work done to replicate the messages in the file system to avoid such failures.

Programming Model

Samza tasks are written using Java programming language. This code specifies how to process the messages. The actual job is configured through a properties file. The properties files along with the compiled code are submitted to the Yarn Samza cluster.

5. DSPE Comparison

Table 1 gives a summary of the available features in the streaming processing engines considered in this report.

Property	Aurora	Borealis	Storm	S4	Samza
Data Mobility	Push	Push based / Data stored for SUnion	Pull based, no blocking operations	Push based	Pull Based, data stored at the message broker file storage
HA & Message Processing Guarantees	Highly available rollback recovery with upstream backup	Highly available with rollback recovery with active backup recovery	Highly available with rollback recovery using upstream backup. At least once processing.	Highly available with Gap Recovery	Highly available with rollback recovery. Data lost when broker failure happens
Data Partition and Scaling	None	Handled automatically by the system	Handled by user configuration and coding	Based on key value pairs	Based on the topic partitioning/ message keys
Data Querying	SQL Based	SQL Based	Trident	None	None
Deterministic or Non-Deterministic	Most of the operators are deterministic	Deterministic	Doesn't specify	Doesn't specify	Doesn't specify
Data Storage	Data can be stored and analyzed	Data is persisted at each node and can analyze the stored data	None	None	Data stored at the brokers and can use these for later processing
Stream Imperfection Handling	None	Yes	User has to implement	None	Can use stored data at the brokers for such cases

Table 1 DSPE Comparison

6. Related Work

Complex Event processing

Complex event processing [**22**] is a widely used technique in enterprises to analyze the data. There is an overlap between complex event processing and stream event processing. Both systems work on events and produce results based on the properties of the events. In complex event processing the incoming events are not belonging to a single stream of events. Instead these events belong to an event cloud. An event cloud contains events from different sources and the causal relationships among the individual events are not reflected in the order of the events. It is the responsibility of the CEP engines to figure out the relationships among the events and do the processing accordingly. In Stream processing the events are coming from a single event stream. In an event stream the events are ordered according to time they are produced and they are coming from the same source. Causal relationships between the events are preserved in an event stream. The stream processing is more streamlined and can be done in efficiently. Stream processing engines require very little memory compare to complex event processing engines because the events don't have to be remembered to discover the causal relationships and ordering.

7. Discussion

In this report we identified key attributes of distributed stream processing systems and evaluated several leading distributed stream-processing engines based on those attributes. The early stream processing engines were monolithic systems that run the stream processing DAG as a single entity. The monolithic systems were replicated as a whole to gain the fault tolerance and high availability. The fault tolerance aspects of the systems are handled at the node level rather than the task level. The node failure handling and message guarantees were taken as a single cohesive operation. The new stream processing engines view a job as set of tasks running on different nodes. The tasks run individual stream operations and can be replicated for fault tolerance at task level. The task failures and message failures are taken as two unrelated operations and handled as two separate concerns. The new model favors deployment in large commodity clusters.

There are three different models introduced for stream partitioning, work distribution among nodes and parallelization of tasks in Storm, S4 and Samza. Each model has its drawbacks and advantages. None of these models have taken a clear lead in the stream processing space and it is too early to tell about their success. The query languages for this large scale distributed stream-processing engines are almost non-existing today and we can expect them to grow as the stream processing engines mature. With the growth of the Internet of Things in the coming years the demand for large scale distributed stream processing will also increase and it is important to find formal stream processing models to lay a foundation to build standard data analytics algorithms on such systems.

8. References

[1] Tyson, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears Condie, "MapReduce Online," , 2010, pp. In NSDI, vol. 10, no. 4, p. 20.

[2] Jeffrey, and Sanjay Ghemawat Dean, "MapReduce: simplified data processing on large clusters," in *Communications of the ACM 51.1 (2008): 107-113*.

[3] Cisco. Cisco the network. [Online]. http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342

[4] Cisco. The Internet of Things. [Online]. http://share.cisco.com/internet-of-things.html

[5] gsma. GSMA ANNOUNCES THE BUSINESS IMPACT OF CONNECTED DEVICES COULD BE WORTH

US$4.5 TRILLION IN 2020. [Online]. http://www.gsma.com/newsroom/gsma-announces-the-business-impact-of-connected-devices-could-be-worth-us4-5-trillion-in-2020

[6] Michael, Uğur Çetintemel, and Stan Zdonik Stonebraker, "The 8 requirements of real-time stream processing," in *ACM SIGMOD Record 34, no. 4 (2005): 42-47.*

[7] J-H., Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik Hwang, "High-availability algorithms for distributed stream processing," in *In Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pp. 779-790*, 2005.

[8] Daniel J., Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik Abadi, "Aurora: a new model and architecture for data stream management," in *The VLDB Journal—The International Journal on Very Large Data Bases 12, no. 2 (2003): 120-139.*.

[9] Mitch, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B. Zdonik Cherniack, "Scalable Distributed Stream Processing," in *In CIDR, vol. 3, pp. 257-268*, 2003.

[10] Ugur Cetintemel, "The aurora and medusa projects," in *Data Engineering 51 (2003): 3.*.

[11] Jeong-Hyon, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik Hwang, "A comparison of stream-oriented high-availability algorithms," in *Technical Report TR-03-17, Computer Science Department, Brown University*, 2003.

[12] Daniel J., Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner et al. Abadi, "The Design of the Borealis Stream Processing Engine," in *In CIDR, vol. 5, pp. 277-289*, 2005.

[13] Magdalena, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker Balazinska, "Fault-tolerance in the Borealis distributed stream processing system," in *ACM Transactions on Database Systems (TODS) 33, no. 1 (2008): 3*.

[14] Apache Software Foundation. Storm Project Incubation Status. [Online]. http://incubator.apache.org/projects/storm.html

[15] ZeroMQ. [Online]. http://zeromq.org/

[16] Netty Project. [Online]. http://netty.io/

[17] Nathan Marz. Github. [Online]. https://github.com/nathanmarz/storm/wiki/Trident-tutorial

[18] Leonardo, Bruce Robbins, Anish Nair, and Anand Kesari Neumeyer, "S4: Distributed stream computing platform.," in *In Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, pp. 170-177*, 2010.

[19] Apache Software Foundation. Samza. [Online]. http://samza.incubator.apache.org/

[20] Apache Software Foundation. (2013, Oct.) Apache Software Foundation. [Online]. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[21] Jay, Neha Narkhede, and Jun Rao Kreps, "Kafka: A distributed messaging system for log processing.," in *In Proceedings of the NetDB*, 2011.

[22] Alejandro, and Boris Koldehofe Buchmann, "Complex event processing," in *it-Information Technology 51, no. 5 (2009): 241-242*.