Survey of distributed stream processing systems

Aravind Alagiri Ramkumar
CSCI 8980

Abstract

This is a conceptual survey of the state of the art technologies in the area of distributed stream processing systems. This survey will cover a short introduction about the core concepts & architectures of the latest distributed stream processing systems and a detailed theoretical analysis by comparing the implementation methodologies of those core concepts across the latest systems focusing mainly on the aspects of fault tolerance and scalability.

1. Introduction

The availability of data from mobile phones, social media, sensor and other IoT devices has led to the explosion of the volume and velocity of the data. The traditional methods of processing data are inadequate to process this big data and as the big data is most valuable if it is analysed quickly as it is generated. This paper provides an insight over the current stream processing systems by comparing their architecture, implementation methodologies of fault tolerance mechanisms and certain other aspects that are presented later on.

The paper is structured as follows. Section 2 provides a background of the big data processing pipeline. Section 3 explains the basics of the stream processing. Section 4 compares the architecture along with several other aspects in some of the widely used stream processing systems. Section 5 concludes the paper.

2. Background of the big data processing

This section provides a background on the big data processing pipeline and where stream processing comes into picture during the big data processing. Figure 1 gives an overall picture of the big data processing.

- The *Data Sources* component indicates the sources of data which requires real time processing.
- The *Data Collection* will be generally performed by the edge nodes present near the data source using the available communication mechanisms.
- The collected data is sent to the *Messaging Systems* such as Queueing systems like Apache ActiveMQ, Apache RabbitMQ or the Publish Subscribe solutions such as Apache Kafka or managed services like Azure IoT hubs.
- *Data Stream Processing systems* are commonly designed to handle and perform one-pass processing of unbounded streams of data by subscribing or pulling data from the messaging systems.
- The range of *Data Storage* solutions used to support a real-time architecture are numerous, ranging from relational databases, to key-value stores, in-memory databases, and NoSQL databases.

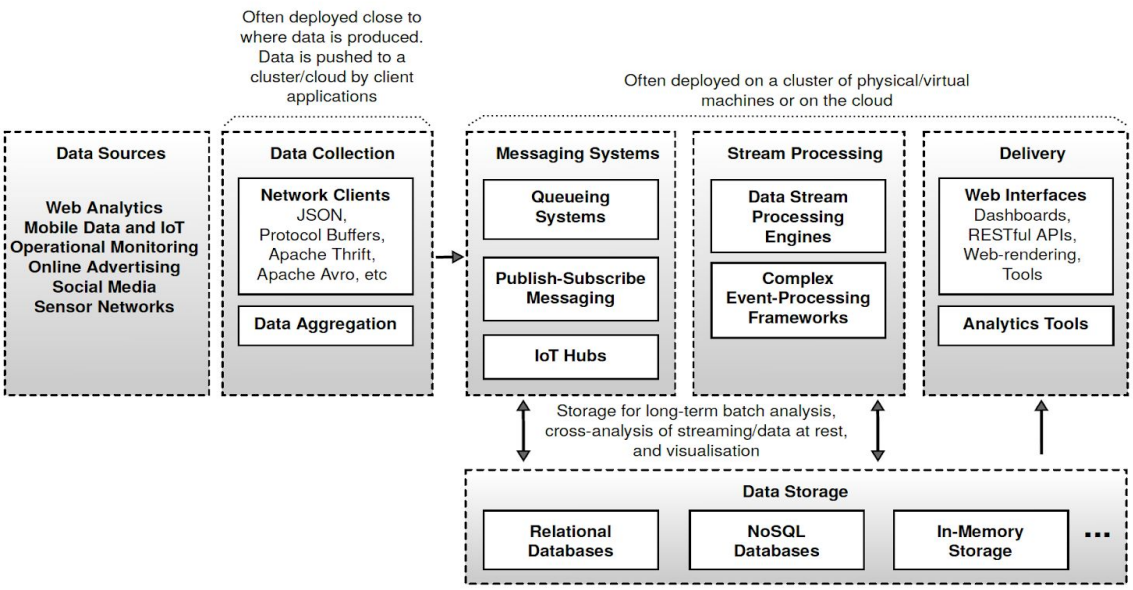

Figure 1: Overview of data processing architecture

3. Basics of Stream Processing

This section explains the basic concepts in the stream processing systems which are necessary for understanding the concepts discussed later on.

3.1 Streaming Processing Model

There are two major types of model in which the data is processed in most of the stream processing systems. They are as follows. *Data flow model* where a processing system is continuously ingesting data that is processed at a tuple level by a DAG of operators. *Micro-batch model* in which incoming data is grouped during short intervals, thus triggering a batch processing towards the end of a time window.

Many Stream processing frameworks uses data flow abstraction. The applications in data flow model are structured as Directed Acyclic Graphs of operators. These operators perform functions such as counting, filtering, projection, aggregation or any user defined functions producing another data stream. Frameworks first devise a Logical Plan of data flow through operators. The scheduler maps the operators in logical plan to physical instances as per the parallelism and other factors.

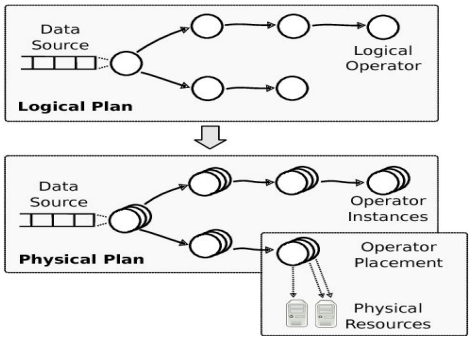

Figure 2: Data Flow Model

3.2 Back Pressure

Backpressure refers to the situation where a system is receiving data at a higher rate than it can process during a temporary load spike. Eg. Garbage Collection Spikes, Surge at the input source

4. Comparison of Stream processing systems

This section will focus on the comparison of the stream processing systems in terms of architecture, message processing semantics, back pressure handling and fault tolerant mechanisms in some of the most widely used stream processing systems namely Apache Spark, Apache Flink, Google Data flow and Twitter Heron.

4.1 Twitter Heron

4.1.1 Architecture

Heron runs topologies. A topology is a directed acyclic graph of spouts and bolts. Spouts generate the input tuples that are fed into the topology, and bolts do the actual computation. [2]

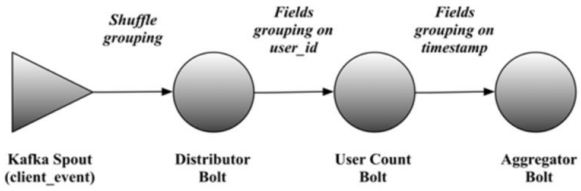

Figure 3: Heron Topology

Users employ the Heron (spouts/bolts programming) API to create and deploy topologies to the Aurora scheduler, using a Heron command line tool. Each topology is run as an Aurora job consisting of several containers, as shown in Figure 4. The first container runs a process called the *Topology Master*. The remaining containers each run a *Stream Manager, a Metrics Manager*, and a number of processes called *Heron Instances* (which are spouts/bolts that run user logic code)

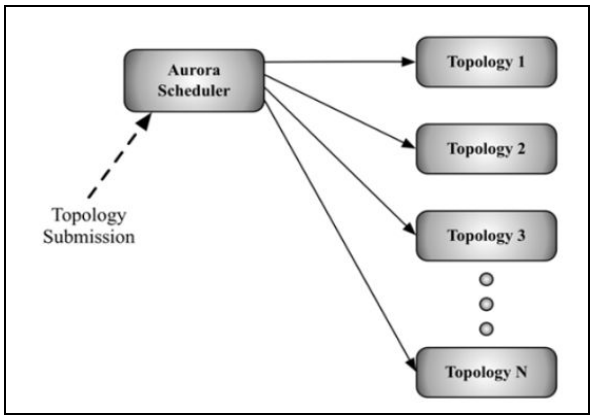

Figure 4: Heron Architecture

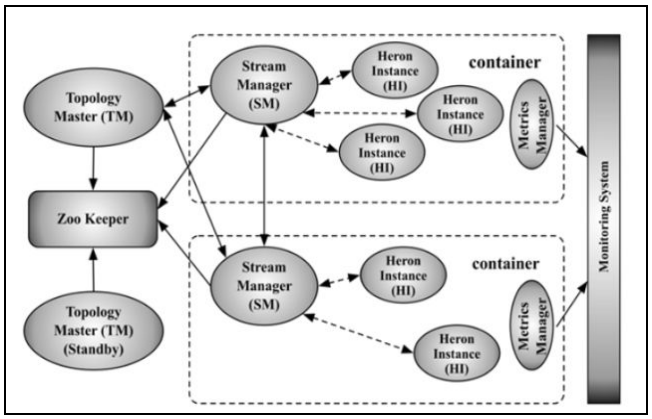

Figure 5: Heron Topology Architecture

The *Topology Master (TM)* is responsible for managing the topology throughout its existence. It provides a single point of contact for discovering the status of the topology. The key function of the *Stream Manager (SM)* is to manage the routing of tuples efficiently. Each *Heron Instance (HI)* connects to its local SM to send and receive tuples. All the SMs in a topology connect between themselves.
When a topology is submitted to Heron, a sequence of steps are triggered. Upon submission, the scheduler (in our case it is generally Aurora) allocates the necessary resources and schedules the topology containers in several machines in the cluster. The Topology Master (TM) comes up on the first container, and makes itself discoverable using the Zookeeper ephemeral node. Meanwhile, the Stream Manager (SM) on each container consults Zookeeper to discover the TM. The SM then connects to the TM and periodically sends heartbeats.
When all the SMs are connected, the TM runs an assignment algorithm to assign different components of the topology (spouts and bolts) to different containers. This is called the physical plan in our terminology. Once the assignment is complete, the SMs get the entire physical plan from the TM, which helps the SMs to discover each other. Now the SMs connect to each other to form a fully-connected network. Meanwhile, the Heron instances (HI) come up, discover their local SM, download their portion of the physical plan, and start executing.

4.1.2 Message Processing Semantics

Twitter Heron supports At least once or At most once message processing semantics depending on the configuration.

4.1.3 Back Pressure Handling

When an SM realizes that one or more of its HIs are slowing down, it identifies its local spouts and stops reading data from them. The affected SM sends a special *start back pressure* message to other SMs requesting them to clamp down their local spouts. When the other SMs receive this special message, they oblige by not reading tuples from their local spouts. Once the slow HI catches up, the local SM sends *stop backpressure* messages to other SMs. When the other SMs receive this special message, they restart consuming data from their local spouts again.

4.1.4 Fault Tolerance Mechanisms

Heron uses a mechanisms of *upstream backup and record acknowledgements* to guarantee that messages are re-processed after a failure. [3]
The process of acknowledgements work as follows. Each record that is processed from an operator sends an acknowledgement that it has been processed. The acknowledgements are sent either to the source operator or to a separate Acker tasks configured to maintain the acknowledgements. The source of the topology keeps a backup of all the tuples it generates. Once a source operator knows that it has received acknowledgements from all operator until the sinks for a particular record, it can safely be discarded from the upstream backup.
At failure, if not all acknowledgements have been received, then the source record is replayed. This guarantees no data loss, but does result in duplicate records passing through the system (hence the term “at least once”)

4.2 Apache Spark

4.2.1 Architecture

Spark uses a master/worker architecture. There is a driver that talks to a single coordinator called master that manages workers in which executors run. A Spark worker can run multiple executors and each executor contains one or more tasks. A conceptual overview of a worker's architecture displayed in figure 6. The Worker Nodes or its executor processes are again responsible for the calculations. Executor processes only work for one program at one time and stay alive until it has finished. A consequence of the first aspect is a complexity reduction with regard to task scheduling, since each application can schedule the tasks of their exclusive executors independently, meaning without considering other programs. The execution scheduling is done by the Driver Program.

Spark uses micro batching model - groups data into batches at the input source forming D-streams[6] which is a sequence of RDD (Resilient Distributed Datasets)[5]. Each RDD contains data for certain stream interval. All the operators operate on these dstreams in Spark.

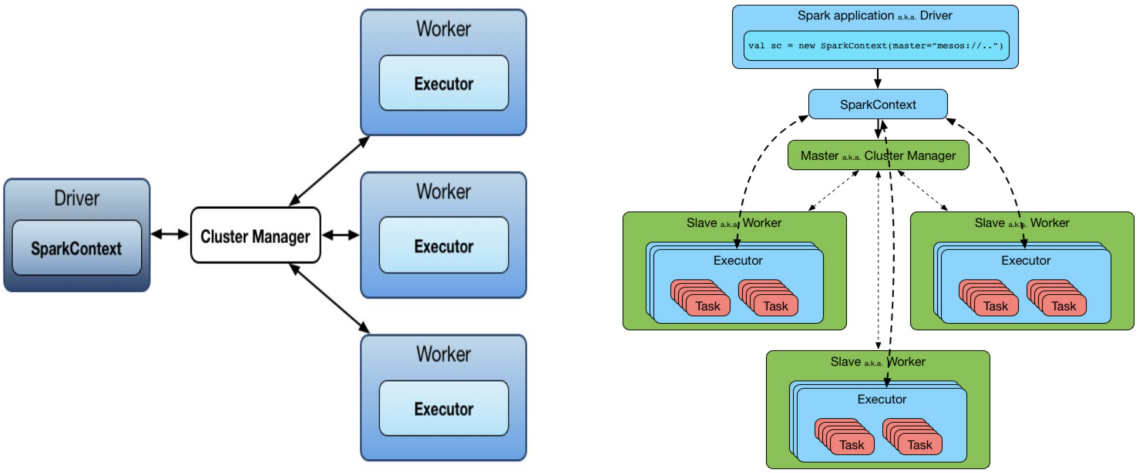

Figure 6: Spark Architecture

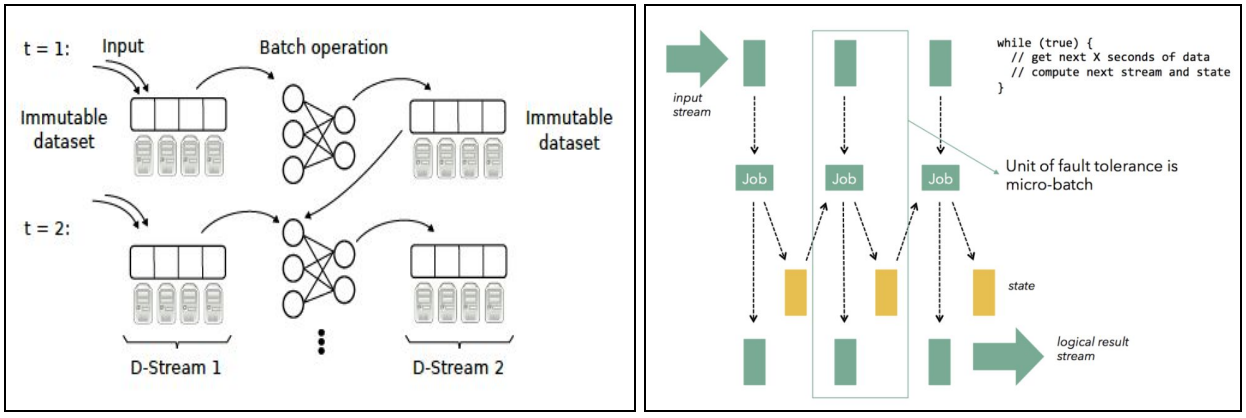

Figure 7: D-streams processing & fault tolerance

4.2.2 Message Processing Semantics

Apache Spark provides Exactly once message processing semantics.

4.2.3 Back Pressure Handling

When using Spark streaming integration with kafka, the easy way to control the max flow rate -- a configuration called spark.streaming.kafka.maxRatePerPartition. According to the documentation, it is the maximum rate (in messages per second) at which each Kafka partition will be read by the API. It prevents micro-batches from being overwhelmed when there is a sudden surge of messages from the Kafka producers.[7]

4.2.4 Fault Tolerance Mechanisms

D-streams are designed handle faults and slow nodes [6]. D-Streams structure a streaming computation as a series of stateless, deterministic batch computations on small time intervals. During a short time interval, D-Streams stores the received data and once the interval elapses, the deterministic computations are performed over the D-streams. These transformations yield new D-Streams, and may create intermediate state in the form of RDDs.

Finally, to recover from faults and slow nodes, both D- Streams and RDDs track their lineage, that is, the graph of deterministic operations used to build them. Spark tracks this information at the level of partitions within each distributed dataset, as shown in Figure 7. When a node fails, it recomputes the RDD partitions that were on it by re-running the tasks that built them from the original input data stored reliably in the cluster. The system also periodically checkpoints state RDDs to prevent infinite recomputation, but this does not need to happen for all data, because recovery is often fast: the lost partitions can be recomputed in parallel on separate nodes. In a similar way, if a node slows down, Spark speculatively executes the copies of its tasks on other nodes, because they will produce the same result.

4.3 Apache Flink

4.3.1 Architecture

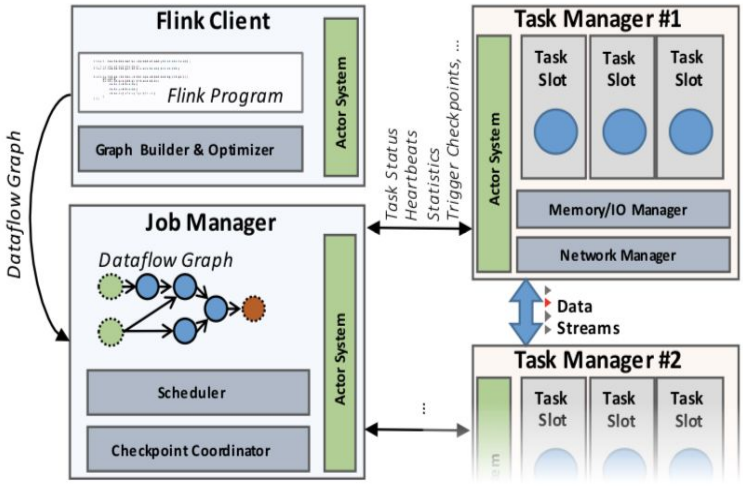

Figure 8: The Flink Process model

A Flink runtime program is a DAG of stateful operators connected with data streams. The client takes the program code, transforms it to a dataflow graph, and submits that to the JobManager. The *Job Manager* coordinates the distributed execution of the dataflow. It tracks the state and progress of each operator and stream, schedules new operators, and coordinates checkpoints and recovery. A *Task Manager* executes one or more operators that produce streams, and reports on their status to the JobManager.
The dataflow graph as depicted in Figure 9 is a directed acyclic graph (DAG) that consists of: (i) stateful operators and (ii) data streams that represent data produced by an operator and are available for consumption by operators. The intermediate data streams are the core abstraction for data-exchange between operators. An intermediate data stream represents a logical handle to the data that is produced by an operator and can be consumed by one or more operators.

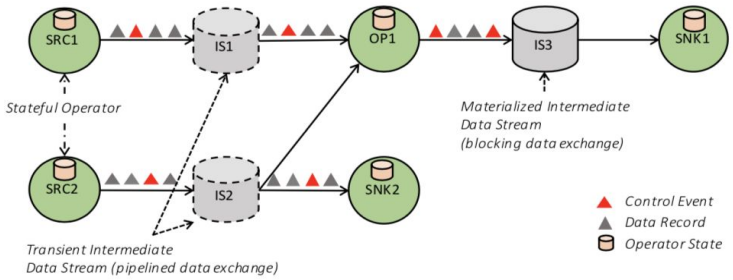

Figure 9: A simple dataflow graph

4.3.2 Message Processing Semantics

Flink provides Exactly once message processing semantics. If needed, it can be configured to At least once message processing model.

4.3.3 Back Pressure Handling

All the data transfers happens through buffers. A managed buffer pool is maintained for every producer and consumer stream. In order for records to progress through Flink, buffers need to be available. You take a buffer from the managed set of buffer pool, fill it up with data, and after the data has been consumed, you put the buffer back into the pool, where you can reuse it again. [9]
Consider a scenario in which task 1 and task 2 runs as the producer and the consumer respectively. If the task 1 wants to send the to task 2, it takes a buffer from the managed buffer pool and fills the data and sends it to the task 2. It is recycled as soon as task 2 has consumed it. If task 2 is slower than 1, buffers will be recycled at a lower rate than task 1 is able to fill, resulting in a slow down of task 1.

4.3.4 Fault Tolerance Mechanisms

Flink offers reliable execution with strict exactly-once-processing via *checkpointing and partial re-execution*. The checkpointing mechanism of Apache Flink builds on the notion of distributed consistent snapshots called *Asynchronous Barrier Snapshotting* [10]. The algorithm is similar to *Chandy Lamport Algorithm.*
Flink's checkpointing mechanism is based on stream barriers that flow through the operators and channels. Barriers are first injected at the sources (e.g., if using Apache Kafka as a source, barriers are aligned with offsets) as control events, and flow through the DAG as part of the data stream together with the data records. A barrier separates records into two groups (as indicated in Figure 8): those that are part

of the current snapshot (a barrier signals the start of a checkpoint), and those that are part of the next snapshot.

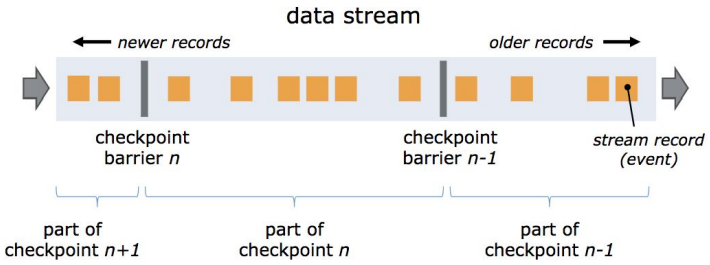

Figure 10: Flink Checkpointing mechanism

The core challenge lies in taking a consistent snapshot of all parallel operators without halting the execution of the topology. Stream barriers are injected into the parallel data flow at the stream sources. An operator first aligns its barriers from all incoming stream partitions (if the operator has more than one input), buffering data from faster partitions. When an operator has received a barrier for snapshot n from all of its input streams, it checkpoints its state (if any) to durable storage and it emits a barrier for snapshot n into all of its outgoing streams. Once a sink operator (the end of a streaming DAG) has received the barrier n from all of its input streams, it acknowledges that snapshot n completed to the checkpoint coordinator.

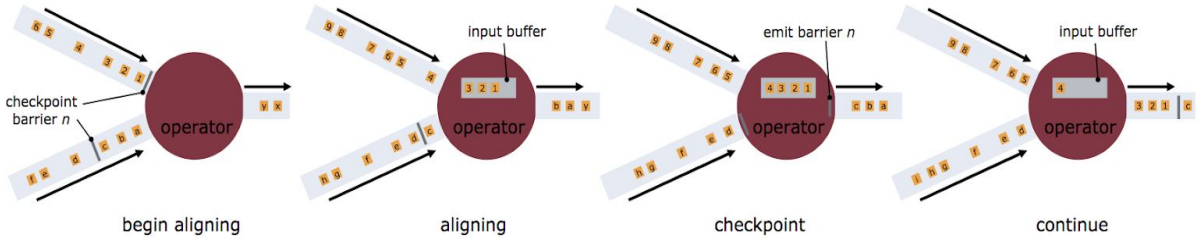

Figure 11: Flink's checkpointing process across multiple streams

4.4 Google Cloud Dataflow

Google cloud dataflow is built on the belief that all the models and other stream processing systems focus on input data (unbounded or otherwise) as something which will at some point become complete. Google dataflow believe this approach is fundamentally flawed when the realities of today's enormous, highly disordered datasets clash with the semantics and timeliness demanded by consumers.

4.4.1 Architecture

The Google Cloud dataflow engine is built on top of the MillWheel streaming engine [11] and the FlumeJava batch Engine, with an external reimplementation for Google Cloud Dataflow, including an open-source SDK. It provides the formal model for the system and explain why its semantics are general enough to subsume the standard batch, micro-batch, and streaming models, as well as the hybrid streaming and batch semantics of the Lambda Architecture. The architecture is not published as it a managed service exposed by google and not open sourced.

4.4.2 Message Processing Semantics

The Google cloud dataflow engine provides an exactly once processing semantics.

4.4.3 Back Pressure Handling

In Dataflow, shuffles are streamed and results do not need to materialize. This gives low latency a natural flow control mechanism as intermediate buffers mitigate back pressure until it reaches the sources (and pull-based sources such as Kafka consumers can deal with this problem).

4.4.4 Fault Tolerance Mechanisms

Fault tolerance in data flow architecture works as follows. Each intermediate record that passes through an operator, together with the state updates and derived records generated, creates a commit record that is atomically appended to a transactional log or inserted into a database. In the case of failure, part of the database log is replayed to consistently restore the state of the computation, as well as replay the records that were lost.

Property	Twitter Heron	Apache Spark	Apache Flink	Google Dataflow
Guarantee	At least once	Exactly once	Exactly once	Exactly once
Streaming Model	Data flow	Micro Batching	Data flow	Data flow
Fault tolerance mechanism	Upstream backup & Acks	Micro batch recomputation	Distributed snapshots	Transactional updates
Over head of Fault tolerance	High (due to acks)	Low	Low	Low (depends on the throughput data store)
Separation of application logic from fault tolerance	No	No (Dstream size matters)	Yes	Yes

Table 1: Comparison of stream processing systems

5. Conclusion

This paper presented the architecture and the key concepts in fault tolerance mechanisms of the widely used stream processing system. It also compared and contrasted the implementations of the fault tolerance mechanisms along with the advantage and disadvantages of each method. This paper also explained how the fault tolerance mechanisms relate to the message processing semantics of the systems. This paper can be taken as a reference for the comparison against the widely used stream processing systems and the users can choose the systems that satisfies their processing requirements.

References

1. STREAM: The Stanford Data Stream Management System (To gain knowledge of basic concepts of stream processing)
2. Twitter Heron: Stream Processing at Scale
3. http://storm.apache.org/releases/current/Guaranteeing-message-processing.html
4. Spark: Cluster Computing with Working Sets
5. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
6. Discretized Streams: Fault-Tolerant Streaming Computation at Scale
7. https://www.linkedin.com/pulse/enable-back-pressure-make-your-spark-streaming-production-lan-jiang/
8. Apache Flink™: Stream and Batch Processing in a Single Engine
9. https://data-artisans.com/blog/how-flink-handles-backpressure
10. https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink
11. MillWheel: Fault-Tolerant Stream Processing at Internet Scale
12. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing