CHAPTER

8

STREAM PROCESSING IN IoT: FOUNDATIONS, STATE-OF-THE-ART, AND FUTURE DIRECTIONS

X. Liu*, A.V. Dastjerdi*, R. Buyya*,**

**Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia; **Manjrasoft Pty Ltd, Australia*

8.1 INTRODUCTION

The emergence of stream processing is driven by the incompetence of the traditional batch-processing paradigm, when it comes to processing fast data. Nowadays, building a modern information-technology system demands the ability of (1) processing an unprecedented volume of data using possibly distributed resources, and (2) exploring the concealed value of data within a tight time-constraint. Having gained extensive attention from the research and industrial community, the batch-processing model derives a series of techniques to accomplish the first goal. MapReduce, for example, is a highly scalable and widely adopted programing model that is specialized in processing parallelization [1]. It moves the computing power to the vicinity of data so that the enormous processing target can be divided and conquered. Analogously, various NoSQL databases are developed alongside traditional relational databases, which allows for an extent of flexibility in data representation to handle the increasing variety of data formats and to obtain finer control over scalability and availability [2]. By taking the horizontal scaling ability as a principle of design, these batch-based techniques are relatively competent in terms of handling the ever-growing data volume and increasingly complex data format. However, they are all struggling to meet the second goal, in which the strict time-constraint has eliminated the luxury of storing the data somewhere before executing relevant operations against it.

In this context, stream processing is proposed as the antithesis of the batch paradigm that caters to the need of processing continuous data-volume in real-time. Both data aggregation and analysis in the streaming model normally have a strict deadline specified, which means completing the job beyond the deadline is not only considered as degradation of performance, but as a failure to deliver the immediate insights that merit the effort in the first place. Guaranteeing the timeliness of aggregation and analysis is a nontrivial task, the streaming paradigm has to continuously aggregate the target-data elements right after its generation to form possible endless streams over the network. When it comes to the data- processing phase, these streams flow through a computation topology where continuous queries (ie, long-standing queries that usually operate over time and buffer windows) are installed to be processed in a record-by-record manner. In contrast to the batch model where data are persisted for future analysis, the streaming model essentially deals with the dynamic

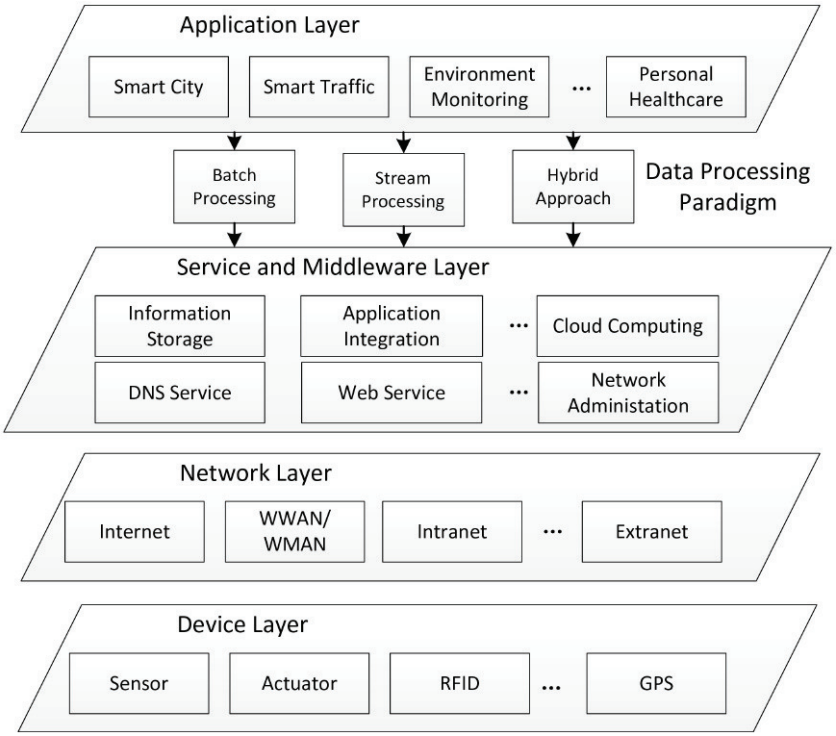

FIGURE 8.1 Stream Processing in the System Architecture of IoT

data-streams that had recently come in, and it incrementally updates the query results. Those data that have passed the processing system cannot be easily retrieved, resulting in an implicit trade-off between the processing accuracy and the real-time promise.

Stream processing has always been an integral part of Internet of Things (IoT) applications, as it offers a scalable, highly available, and fault-tolerant solution to handle a high volume of data in motion. As shown in Fig. 8.1, the architecture of IoT outlines the importance of stream processing and how it is connected with the rest of the system. From the perspective of an application developer, stream processing mainly works as a connecting bridge between the application layer and the service and middleware layer, which allows the upper logic to make appropriate use of the underlying general-purpose services and infrastructures. For example, the streaming paradigm may decide that only the synopsis of incoming data needs to be preserved in the storage system so that no external database is required, or the runtime framework that conducts the data analytics needs be placed in a cloud environment to take advantage of its elasticity feature. In addition to that, the stream-processing paradigm also has a significant impact on the organization of the network layer and the device layer to keep that real-time promise. As a matter of fact, the major part of latency between the data generation and result delivery lies in the data-collection phase rather the processing phase. Therefore, it is essential for an IoT application to properly select the substructures that suit its particular time-sensitivity requirement.

On the other hand, the applications from the IoT domain have always been the driving force that motivates the development and adoption of the stream-processing paradigm. The primary cause is that

the way of data generation has become increasingly active in the emerging IoT applications. Previously, data in the conventional scenarios resulted from passive reactions to real-world events or user queries, but nowadays IoT data are mostly automatically generated by large-scale sensor networks for monitoring and decision-making purposes. As a consequence, not only has the amount of data being generated soared, but also the places of data production have become much more geographically dispersed than before. In some cases, leveraging the stream-processing model to handle data in motion is the only viable option.

Besides, the value of IoT data has also become increasingly sparse and deeper hidden, which results in a significant change in the relevant processing techniques. Prior to the IoT era, we had intended to collect comparatively a small fraction of data with high precision to quickly perform analysis and get results in time. However, today the data format used by IoT application is known to be heterogeneous, unstructured, and fine-grained. This is a result of numerous factors, including the advancement of mobile and internet technologies, popularity of social media, and widespread deployment of sensors and actuators in a mutable environment. For example, to make the city more desirable and liveable, the smart-city program installed in Rio de Janeiro, Brazil in 2010 has set up an operations center to analyze real-time data 24/7 from 30 municipal institutions. In this program, raw information is collected from various data sources to support the central surveillance and analytics at a single hub, including live video from traffic and public transport, position information from Google Maps, and real-time alarms from the sensor networks on utility and emergency-service readouts [3]. Such an explosion on data dimensionality has made it impossible to meaningfully correlate the small datasets from these sources, regardless of how precise they are. Revealing the true value of IoT data therefore largely depends on the processing of a massive amount of heterogeneous data, where stream processing comes up as a handy tool, as it provides the required availability, high throughput, and real-time support.

This chapter aims to provide a discussion on the concept and general architecture of a stream-processing system in the context of IoT, with a focus on comparing various platforms that are available to process continuous logic according to specific application needs. To this end, we first analyze the characteristics of stream data, and then we present a general stream-processing architecture, which is determined by the associated processing demands of each characteristic. Finally, this chapter concludes with an outlook that explores the future directions and trending topics regarding the development of stream processing in the IoT domain.

8.2 THE FOUNDATIONS OF STREAM PROCESSING IN IoT

There is a considerable ambiguity related to the terms “stream processing” and “stream analytics,” as they have been simultaneously used by a diverse range of research communities. For example, stream processing in the context of parallel processing refers to a computer programming paradigm that allows applications to better exploit computation parallelism using a combination of heterogeneous resources, such as CPU, GPU, or field-programmable gate arrays (FPGAs) [4]. On the other hand, stream processing in the field of connection-oriented communications means to transmit and interpret digitally encoded coherent signals in order to convey data packets for the higher-level network abstraction [5]. In the jargon of the database community, “processing stream” refers to a particular ability owned by active DBMS to handle external updates with reactive behaviors, according to the predefined Event Condition Action (ECA) rules [6]. Nevertheless, these definitions of stream are either overly limited or not directly related to our topic, which is the processing of either distributed events or data items for real-time IoT applications. In order to clarify the scope of this chapter, we first define the terms “stream” and “stream processing” in the IoT context.

8.2.1 STREAM

A stream is a sequence of data elements ordered by time. The structure of a stream could consist of discrete signals, event logs, or any combination of time-series data, but the way of recovering data from one to another must be append-only, resembling a conveyor belt that continuously carries data elements through a processing pipeline. In terms of representation, a data stream has an explicit timestamp associated with each element, which serves as a measurement of data order. Based on this, we formally define the denotation of *stream* in the context of IoT, as a Data Element–Time pair (s, Δ), where

1. s is a sequence of data elements that are made available to the processing system over time. A data element may consist of several attributes, but it is normally atomic, as these attributes are tightly coupled with one another for logical consistency.

   Typical element types include immutable data tuples of the same or similar category, as well as heterogeneous events that come from a variety of sources. Depending on the specific application scenario, data elements can be either regularly generated by sensor networks that have monitoring intervals, or randomly produced by real-world events such as user clicks on a website, updates to a particular database table, and system logs produced by Internet services.
2. Δ is a sequence of a timestamp that denotes the sequence of data elements. Since heterogeneous elements could be aggregated into a single stream out-of-order due to the uncertainty of distributed data-collection and transmission procedures, the use of a timestamp is necessary to reconstruct the logic sequence for the following analytics. In addition, timestamps can be also used to evaluate the real-time property of a stream-processing system, by checking on whether the results have been presented on time.

   Normally, timestamps can be implemented in two forms: (1) as a string of absolute time-values, which consumes more resources to be processed, but makes it easier for developers to devise joint algorithms on separate streams; or (2) as a sequence of positive real-time intervals that only record the relative order of data elements in the same stream. The latter form alleviates the stress of the network by reducing the size of the timestamp, but it is harder to reorder the sequence of events across different streams with only in-stream intervals.

8.2.2 STREAM PROCESSING

Stream processing is a one-pass data-processing paradigm that always keeps the data in motion to achieve low processing-latency. As a higher abstraction of messaging systems, stream processing supports not only the message aggregation and delivery, but also is capable of performing real-time asynchronous computation while passing along the information. The most important feature of the streaming paradigm is that it does not have access to all data. By contrast, it normally adopts the one-at-a-time processing model, which applies standing queries or established rules to data streams in order to get immediate results upon their arrival.

All of the computation in this paradigm is handled by the continuously dedicated logic-processing system, which is a scalable, highly available, and fault-tolerant architecture that provides system-level integration of a continuous data stream. As a consequence of the timeliness requirement, computations for analytics and pattern recognition should be relatively simple and generally independent, and it is common to utilize distributed-commodity machines to achieve high throughput with only sub-second latency.

Table 8.1 Comparison of the Stream Model and the Batch Model

Aspects	Stream Model	Batch Model
Management target	Transient streams	Persistent data batch and relations
Amount of data	Possibly infinite	Finite
Processing model	In-memory processing	Store-then-process and in-memory processing
Query model	Continuous and standing-by query	One-time query
Access model	Sequential access	Random access
Result repeatability	Nearly impossible	Easy
Pattern of result update	Incremental update	Global update
Focus of processing	Low latency and high throughput	High accuracy and comprehensiveness

However, there is another subclass of stream-processing systems that follows the microbatch model. Compared to the aforementioned one-at-a-time model, in which it is difficult to maintain the processing state and guarantee the high-level fault-tolerance efficiently, the microbatch model excels in controllability as a hybrid approach, combining a one-pass streaming pipeline with the data batches of very small size. It greatly eases the implementation of windowing and stateful computation, but at the cost of higher processing-latency. Although such a model is called *microbatch*, we still consider it to be a derived form of stream processing, as long as the target data remains constantly on the move while it is being processed. In order to better illustrate the basic idea of stream processing, we compare it to the well-known batch paradigm in Table 8.1. Although these two paradigms share some similarities in terms of the objective and functionality of processing, they differ significantly in the way that data is organized and processed.

When it comes to the application of stream processing, we have identified two utterly different types of use cases. The first one is *Data Stream Management*. The system falling in this category is normally called *Data Stream Management System (DSMS),* analogous to the traditional DBMS, whose goal is also to manipulate a huge amount of available data to constitute data synopsis, schema, or some other mathematical or statistical model that is easy to understand and interpret. Specifically, data streams within the DSMS are joined, filtered, and transformed according to specific application logic with the use of continuous and long-standing queries. In the early days of DSMS, an application developer could easily set up those queries using SQL-like declarative language, whereas the real implementation was left to be transparently handled by the DSMS. However, since the throughput requirement of stream processing has soared during the recent decade, and the corresponding DSMS has become increasingly distributed, sticking to such a declarative model makes it painful to horizontally scale, and even harder to maintain the required availability and fault-tolerance ability. Therefore, the state-of-art DSMS mostly adopts the imperative way to implement long-time queries, by using the provided programming API, where a segment of code is performed upon the arrival of each incoming data element to compose the whole-analysis logic. Additionally, the responsibility of managing the processing state now rests on the shoulders of the application developers, resulting in a nontrivial effort to debug the application, as well as tune the performance on a specific platform. A typical use-case of DSMS includes face recognition from a continuous video stream, and the calculation of user preference according to his or her click history.

The other use case is called *Complex Event Processing (CEP)*, which is essentially tracking and processing streams of raw events in order to derive significant events and identify meaningful insights from them. There are several techniques being used to achieve that goal. The most notable one is to implement and configure the processing logic as a set of inferring rules in the knowledgebase so that they could be used in the decision-making process of identifying complex patterns. To define and preserve the mutual relationship of events, various types of event-processing languages have been proposed to correlate the seemingly independent events with the relationships such as causality, membership, and timing. Besides, CEP systems normally require that the maintenance of state and the preservation of event relationship be provided at the system level rather than the application level, which makes the microbatch model a preferable option compared to the one-at-a-time model.

In contrast to the primary goal of DSMS, which performs stream analytics at a geographically concentrated place, the major concern of CEP is to infer the needed insight from the vast volume of raw events to stream as fast as possible. Therefore, the computation complexity of CEP logic is usually lower than that of DSMS, and it is preferable to make the rule-matching process take place somewhere near the data generation.

For the sake of clarity, we summarize the major differences between DSMS and CEP in Table 8.2.

However, the boundary of CEP and DSMS is not clearly demarcated in terms of the implementation. A CEP system can be built on top of DSMS by implementing event rules with query languages, whereas the functionality of DSMS can be provided by certain CEP systems that have analytic logic integrated into the rule-based knowledgebase. Actually, there is an ongoing trend that a single stream-processing platform is able to serve both the use cases without requiring too much modification. For example, Apache Storm, a prevalent real-time computation framework that receives a lot of attention recently, has the one-at-a-time model at its core, which makes it an ideal platform for data-stream

Table 8.2 Differences Between Two Use-Cases of Stream Processing: DSMS and CEP

Aspects	DSMS	CEP
Processing target	Continuous streams of data	Discrete events
Typical data sources	Video or audio stream, user clicks, social media context.	Sensory information System and service logs
Data variety	Structured, semi/unstructured	Normally structured
Logic implementation	Continuously queries	Event-matching rules or state automaton
Amount of applied logic	Small	Large
Typical application scenario	Quantitative analytics	Qualitative inference
Scalability	Horizontal scale-out	Vertical scale-up
Preferred venue of processing	Collect and aggregate information to a single location to achieve centralized processing	Amortize the processing task throughout the data chain and bring the computation near the data source to relieve the network overhead
Notification of decision	Usually provide analytics result for another system to make a decision	Make decision based on detected insight and inform the outside world as fast as possible

management. But with the built-in Trident abstraction, Apache Storm can easily fulfill the requirement of CEP by using the microbatch paradigm to become a typical event-processing platform that is capable of identifying meaningful patterns from incoming raw events. Such increasing unity has made it possible to propose an abstract architecture of a stream-processing system which generally satisfies the processing needs coming from both the DSMS and CEP domains.

To this end, we first present a detailed analysis on the characteristics of stream data, as well as their relevant processing requirements. Then we investigate the general architecture of a stream-processing system to cater to these particular requirements and shed some light on how an integral data-processing chain is constituted by the independent streaming components.

8.2.3 THE CHARACTERISTICS OF STREAM DATA IN IoT

As suggested by its name, stream data in IoT constitutes inherently dynamic, continuous, and unidirectional data flows that are normally processed in a one-pass manner. Such a dynamic paradigm has endowed it with several common properties, such as timeliness, randomness, endlessness, and volatility.

8.2.3.1 Timeliness and Instantaneity

Ensuring the timeliness of processing requires the ability to collect, transfer, process, and present the stream data in real-time. As the value of data may vanish over time rather rapidly, the streaming architecture needs to perform all the calculation and communication on the fly with the data that has newly arrived.

On the other hand, the data generation in IoT environments mainly depends on the status of data sources. The amount of data that is generated at low-activity periods can be dramatically less than the number observed at peak times. Usually the stream-processing platform has no control over the volume and complexity of the incoming data stream. Therefore, it is necessary to build an adaptive platform that can elastically scale with respect to fluctuating processing demands, and still remain portable and configurable in order to stay agile in response to the continuously shifting processing needs.

8.2.3.2 Randomness and Imperfection

Randomness and data imperfection are two direct consequences of the dynamic nature of stream data. There could be several unforeseeable factors that affect the processing chain. For example, the data generation process may induce randomness because the data sources are normally independently installed in different environments, which makes it nearly impossible to guarantee the sequence of data arrival across different streams. Besides, the data transmission process can also result in disorder and other defections in the same data stream, as some tuples may be lost, damaged, or delayed due to the constantly changing network conditions. Stonebraker et al. have elaborated on the possible types of data imperfection found in stream data, and list the capability of handling imperfections on the fly as one of the eight requirements of real-time stream processing [7].

8.2.3.3 Endlessness and Continuousness

As long as the data sources are alive and the stream-processing system is properly functioning, newly generated data will be continuously appended to the data channel until the whole application is explicitly turned off. Therefore, processing stream data needs the support of high-level availability to avoid any possible interruption of data flow, which may lead to the accumulation of backlogs, and, finally, the breach of the real-time promise.

Table 8.3 Characteristics of Stream Data and the Corresponding Processing Requirements

Characteristics	Corresponding Requirement
Timeliness and instantaneity	1. Data cannot be detained in any phase of the processing chain, so there should be a comprehensive data-collection subsystem working as a driving force that powers the data in motion once they are generated. 2. For compute-intensive applications, a data aggregation subsystem is needed to gather the collected data for centralized processing. 3. Each phase of the processing chain is preferable to be horizontal scalable in order to keep pace with the fluctuated workload.
Randomness and imperfection	1. For cleansing and coordination purposes, data should be first buffered in a message subsystem before being processed. 2. A declarative or imperative CLPS is responsible for implementing the application logic and handling possible data-stream imperfections.
Endlessness and continuousness	1. The storage subsystem can only be used as an assistance component that preserves the data synopsis or the query results. 2. Ensuring the availability is one of the core design principles due to the continuousness of workload.
Volatility and unrepeatability	1. The data value and insights discovered from the streams should be immediately submitted to other services or presented to users through a presentation subsystem. 2. The fault-tolerance ability is another system design principle, as it is costly or even impossible to replay the incoming stream during the recovery of system failures.

8.2.3.4 Volatility and Unrepeatability

Most of the stream data will be discarded once they have finished traversing through the stream-processing system, which makes the existence of data quite volatile. Even if the data sources are able to replay the data stream upon the retransmit request, the new stream is unlikely to be exactly the same as the previous one. Also, the timeliness of result presentation would be impaired because of the reprocessing.

Table 8.3 summarizes the processing requirements with regard to the corresponding characteristics, where the phrases in italic denote the streaming components that need to be implemented in the different stages of the data-processing chain.

Apart from these common properties, stream data in IoT is known to be highly dynamic and heterogeneous. The dynamism not only refers to the varying data volumes, but also it denotes the constantly changing data quality, credibility, and presentation model that are caused by the dynamicity of the environment. Since there could be a series of resource constraints that confine the ability of data sources and even alter the structure of the data transmission network, the stream-processing system is required to be workload-adaptive and context-aware so it can keep on finding meaningful insights from the ever-changing raw data.

Heterogeneity is another notable characteristic brought on by the IoT context. As an example, smart-city application, a mobile app that automatically searches for empty parking spaces for the car, the driver needs to collect various formats of data from different places to make a comprehensive decision. For instance, the app uses the GPS signal from the driver's personal device to determine

the current position, inquires a vacancy pool to show the possible alternatives, including the location and permitted parking hours, and makes a recommendation among these alternatives, using the traffic conditions from a road- monitoring system. As most of the raw data are extracted from the sensory information through distributed smart-devices and embedded sensors in real time, it is a great challenge for the data collection system to achieve data federation and provide a unified view from the upcoming heterogeneous sets of data and the prior knowledge extracted from the history information.

8.2.4 THE GENERAL ARCHITECTURE OF A STREAM-PROCESSING SYSTEM IN IoT

First of all, we argue that a *stream-processing architecture* should include an integral data-processing chain that covers the whole lifespan of data (from its generation up to its consumption). However, most of the previous research had used this term in a narrower sense, only referring to the organization of a logic-processing system where the relevant analytics are performed.

For example, a widespread survey written by Gugola et al. broke down the general architecture of an information-processing system into five major components: the *receiver*, *decider*, *producer*, and *forwarder* that manipulate data streams according to the designated logic, and a *knowledgebase* that assists the *decider* during the decision-making process [8]. This usage implicitly assumes that the incoming data has already been shaped as continuous streams and can be readily obtained by the receiver, so that the counting of processing latency should start from the time at which the data streams enter the system, rather than the time when data is generated. However, this assumption regarding the triviality of data collection and aggregation is tenable only when the research purpose is to evaluate the correctness and competence of a particular logic-processing subsystem. When it comes to building stream applications for real-world scenarios, such an assumption is poorly suited because collecting and aggregating data from geo-distributed data sources are inherently costly procedures. There are a series of development and deployment hurdles to be overcome by the use of dedicated streaming components.

Fig. 8.2 presents a general architecture of a stream-processing system that is tailored to the IoT peculiarities. This architecture breaks down the whole data-processing chain into several stages according

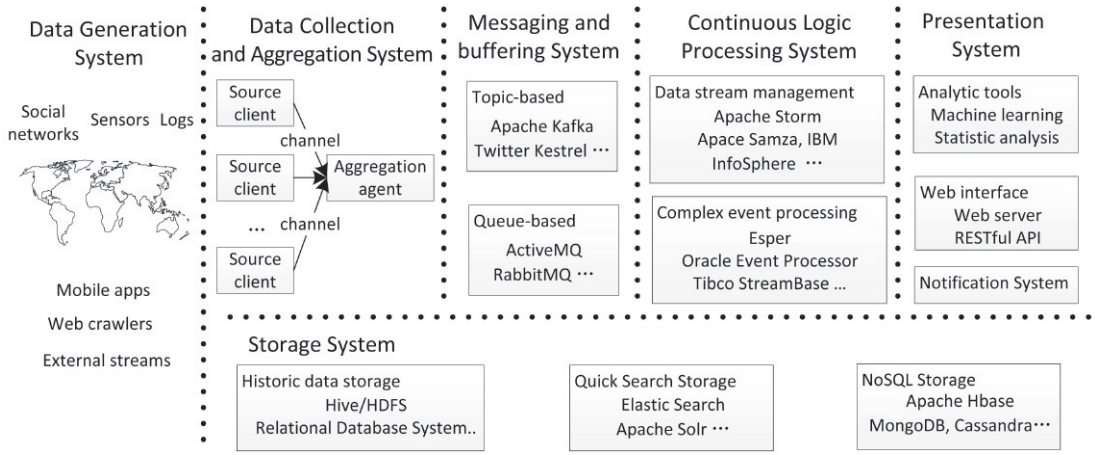

FIGURE 8.2 General Architecture of a Stream Processing System in IoT

to the functionality and target; we have identified six separate streaming components which are responsible for data generation, collection, buffering, logic processing, storage, and presentation, respectively.

The data-generation system denotes the spectrum of data sources that continuously produce raw information for the data-processing chain. There are a lot of entities that can fulfill this definition, which makes a full enumeration nearly impossible. However, we can still categorize the generated data into three types, in accordance with their modalities.

The first type, *static data*, refers to the long-term information that has already been stored in on-premise infrastructures or remote locations. As these data are mostly derived from the validated knowledge and are not frequently updated, they are usually fetched by the stream-processing system on a regular basis, serving as reference information during the analytic procedure. The second type, *centralized stream data*, is a special type of stream that only comes from a single centralized data source. Data of this type sometimes even demands to be processed right in the same place where it is generated, so there would be no need for aggregating data to achieve a unified data-view. However, this type of data is also not the mainstream input for IoT stream applications, for the reason that it is rather rare that one data source can generate all the information that is required for the analytic process. Apart from these two, *Distributed stream data* is the most common data type used in IoT applications. Data of this type dynamically come from various distributed places in heterogeneous formats, such as sensory information from sensor networks, personal preferences from mobile devices, and social-media streams from Internet services. The volume of distributed stream data and the time sensitivity of its application actually determine the performance requirement for a particular stream-processing system.

However, no matter which form of data is being produced, the data sources have to generate a unique timestamp associated with it to denote the time of generation. These timestamps are used to build the continuous processing logic and further evaluate the timeliness of execution.

The *Data Collection and Aggregation System* combining with the *Messaging and Buffering System* plays the role of a message broker in the whole data-processing chain. To collect and aggregate different types of data, various forms of source clients are independently installed to drive the newly generated data in motion, while several aggregation channels are provided to gather these stream data into a centralized buffer, using hierarchical aggregation agents. There are two types of message buffers in terms of implementation: some are topic-based, which support a higher-level programmability, whereas the others are queue-based, and thus mainly optimized for performance concerns.

The *storage system* and *presentation system* are two supportive components for a stream-processing architecture. Keeping all the historical data in the storage system is neither feasible in implementation nor necessary in terms of the processing requirement. Therefore, data that need to be stored are either established knowledge, which can guide the future processing, or meaningful data synopsis, which might arouse the future interest of users. On the other hand, the presentation system serves as an interface of the stream-processing system, wherein it immediately hands over the data value to the higher-level analytic tools, or directly delivers the results or notifications to the end users. It is also responsible for receiving search-command or query updates from the external environment so that it can make the stream-processing system more adaptive and responsive.

As the core of the data-processing chain, the *Continuous Logic Processing System* (CLPS) deserves to be separately reviewed in the next section. As suggested by the name, it is responsible for processing aggregated data according to the designated continuous logic, which could either come from the Data Stream Management or Complex Event Processing background.

8.3 CONTINUOUS LOGIC PROCESSING SYSTEM

In particular, we thoroughly discuss the history of the CLPS from an evolutionary perspective, and then outline the differences among some state-of-the-art CLPS implementations.

The origin of the CLPS dates back to the beginning of this century. As shown in Fig. 8.3, the first generation of CLPS, pioneered by NiagaraCQ [9] and STREAM [10], is merely several prototypes from the research community and only suited for certain processing scenarios in which only a small amount of data are generated. In addition to that, the types of operations supported by these prototypes are also limited, which means that they are usually used as functional extensions of the existing Data Base Management Systems (DBMS).

On the other hand, these prototypes are ground-breaking explorations in the new area of stream processing. NiagaraCQ [9], for example, defines a simple command-language to create and drop continuous queries over XML files on the fly. It also supports grouping continuous queries based on their structures, and performs incremental evaluations of each group by considering only the changed portion of the targeted XML file. Besides, this command language adopts a declarative syntax to make it developer-friendly, which can help the existing queries written in transitional SQL to be transplanted into the new stream-processing platform.

In contrast to NiagaraCQ, the focus of STREAM [10] developed by Stanford is to transfer from persistent relations to transient data streams with window-based data processing and approximate query answering. STREAM directly supports SQL-like query language so that it can be regarded as a functional extension of traditional DBMS. With the lessons learned from STREAM, the authors also discuss models and issues in managing data-stream systems [11].

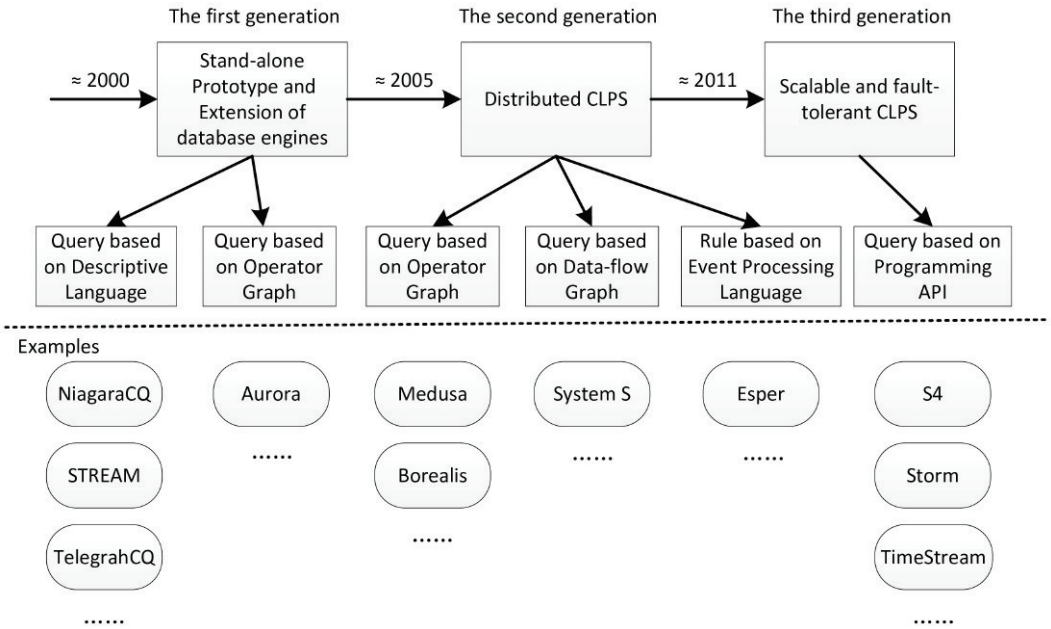

FIGURE 8.3 Evolutionary History of CLPS

There are also some CLPSs that are directly built on top of existing databases. TelegraphCQ [12] is an example of a system falling into this category. It is developed on PostgreSQL to cope with the high-value and diverse data streams, and enables the possibility of adaptive querying.

Aurora [13] is the last breakthrough founded in the first generation of CLPS. Within the help of the "boxes and arrows" paradigm, the continuous queries are implemented by explicit operator graphs rather than declarative query languages such as SQL. As a result, the performance of stream processing is significantly improved at the cost that the query implementations and internal processing mechanisms are no longer transparent to the developer.

Around 2005, the research front advanced to distributed stream processing, where the CLPS is able to take advantage of a set of distributed hosts to achieve better scalability and fault-tolerance. The project Medusa [14] is an extension to Aurora, which leads to a scalable and QoS-oriented architecture. As a result of distribution, the logical entities of Medusa are no longer tightly coupled and they have to communicate with each other through a naming, discovering, and message-passing process. In addition to that, the problem of load balancing and resource management also emerged as a great challenge in distributed CLPSs, for example, as an operator node could be split into several atomic units and re-mapped to participating machines, it is important to design a dynamically partitioned operator network to improve the resource utilization.

Borealis [15] engine was developed on top of Medusa in order to integrate some advanced capabilities, including dynamic query modification, result revision, and flexible monitoring. Apart from that, Borealis also introduces the concept of a replicated processing node, and defines several new tuple types, such as punctuation tuples and priority tuples, to gain finer control over the fault-tolerance and manageability of the distributed platform.

In contrast to the academic projects such as Aurora, Medusa, and Borealis, whose queries are mostly implemented based on an operator graph, System S [16], a proprietary CLPS developed by IBM, proposed a query model based on data-flow graph to hide the implementation details as much as possible. The core objective is to achieve highly scalable, resource-efficient processing through a user-oriented declarative abstraction with a balanced resource-allocation mechanism. Afterward, the developers in IBM also introduced an intermediate language called SPADE [17] to grant the users more flexibility by allowing them to design the data-flow graph and the associate stream operators on their own.

The aforementioned systems all fall into the subcategory of DSMS. On the other hand, the project Esper chooses a different evolutionary path in terms of the implementation of continuous logic. By using the event processing language (EPL) and a pluggable runtime library, Esper is suitable for distributed event processing that has different types of events defined [18]. Some commonly used operations like joint splitting and filtering can be easily expressed by EPL with a very similar syntax to SQL, so that the programming burden of the developer is also significantly relieved.

However, the advent of Web 2.0 and IoT applications has brought the previous CLPSs to their knees. Around the year of 2011, CLPS evolved into the third generation, which is inherently scalable and fault-tolerant, and designed for a large-size cluster composed of commodity machines.

Among others, S4 [19] developed by Yahoo! is generally perceived as the first CLPS that meets the criteria of fully scalable and fault-tolerant. It offers intuitive programming API similar to MapReduce that can be used to develop streaming applications. However, S4 does not guarantee the correctness of processing, so streaming data could be either lost or repeatedly executed during the processing process.

Fortunately, the emergence of the Storm project [20] gracefully handles this requirement by introducing the anchor mechanism. With anchor, Storm is able to process a tuple with either an exactly once,

at least once, or at most once, guarantee. Besides, it supports almost arbitrary programming languages such as Clojure, Java, Ruby, and Python to implement the spouts and bolts, which are the logical operations in Storm. It also greatly enhances its fault-tolerant ability with finely grained task-level parallelization, as when a host fails, all of the running tasks on it could be transferred to other healthy hosts.

TimeStream [21] is another scalable CLPS written in C# by Microsoft. According to the authors' evaluation, it is able to handle an advertising aggregation data-source with a data generation speed of 700,000 URLs per s, 1 per a 6-node commodity cluster, all within 2 s. Similar to Storm and S4, TimeStream adopts a task DAG to denote the sequence of logical operators. To make the system adaptive and autonomous, there is a resilient substitution mechanism to dynamically adjust and reconfigure task DAG in accordance with the changes to incoming streams or in the presence of any failed nodes.

There are also some other state-of-the-art CLPSs that are available to be inserted into the data-processing chain. As shown in Table 8.4, we compare these alternatives in terms of:

- *System architecture*: it outlines internally how a CLPS is organized and coordinated
- *Data transmission*: the way that streaming data feeds the processing system. *Pull-based* means that CLPS is responsible for actively fetching data, whereas *push-based* means passive message- reception
- *Development language*: which languages are being used to develop the CLPS
- *Programming*: which components need to be programmed to apply the continuous logic
- *Partitioning and parallelism*: how data is partitioned to achieve processing parallelization
- *Accurate recovery*: whether the CLPS is able to accurately reproduce the same processing result when failures occur to the system
- *State consistency*: whether the system is able to ensure the consistency state for all of the participating components during the processing procedure

8.4 CHALLENGES AND FUTURE DIRECTIONS

The current stream-processing systems have been greatly improved to cater to the emerging needs of IoT applications. A state-of-the-art stream-processing system now should satisfy the following criteria: (1) horizontal scalability to accommodate different sizes of processing needs, (2) easy to program and manage while concealing the tedious low-level implantation from its users, and (3) capable of dealing with possible hardware faults with graceful performance degradation rather than sudden termination.

However, there is still a long way to go before the stream-processing systems achieve their full maturity. The following aspects summarize the challenges that still need to be further addressed, and also point toward the possible research directions that should attract more attention from the research community.

8.4.1 SCALABILITY

Scalability does not just refer to the ability to expand the system to catch up to the ever-increasing data streams, so that the promise of the Quality of Service (QoS) or Service Level Agreement (SLA) could be honored. Elasticity, the ability to dynamically scale to the right size on demand, is the future and advanced form of scalability. An efficient resource-allocation strategy should be adopted, by which the stream-processing system can start running with only limited resource usages, especially when the data sources are temporarily idle during the application-deployment phase. Afterward, as the workload of IoT may fluctuate and the user requirement may change over time, the system should dynamically

Table 8.4 Comparison Between State-of-the-Art CLPS Implementations

Aspects	Apache Storm	S4	Spark Streaming	Apache Samza	Apache Flink	Esper
System architecture	Master–slave	Symmetric	Master–slave	Master–slave	Master–slave	Master–slave
Data transmission	Pull-based	Push-based	1. Push-based with flume 2. Pull-based with a custom sink	Pull-based	Push-based	Pull-based
Develop language	Clojure	Java	Scala, Java	Java	Java, Scala	Java, C#
programming	Spouts and Bolts	Processing elements	Distributed datasets	Samza job	Data stream and transformations	EPL
Partitioning and parallelism	Sending to different tasks	Based on key-value pairs	Sending to different tasks	Sending to different tasks	Sending to different tasks	Partition based on context
Accurate recovery	Yes, with trident	No	Yes	Yes	Yes	No
State consistency	No	No	Yes, with state DStream	Yes, with embedded key-value store	Yes, with asynchronous distributed snapshots	No

provision new resources by taking into account the characteristics of the available hardware infrastructure, and free up some of them when they are no longer needed. Such an awareness of underlying infrastructure can help the system to perform more reasonable elastic operations, and is also useful for scheduling task loads in case of hardware failures.

8.4.2 ROBUSTNESS

Fault-tolerance is a commonplace topic when it comes to the design and implantation of stream-processing systems, especially when considering that its availability is one of the most crucial prerequisites to guarantee the correctness and significance of real-time processing. The previous research and practice on fault-tolerance mostly rely on either system replication or state checkpointing, which are both not flexible enough to tailor to the robustness for operations in accordance with the trending fault-types. Designing a hybrid and configurable fault-tolerance mechanism that is capable of recovering the system from unforeseeable failures is an open research-question left to be answered.

8.4.3 SLA-COMPLIANCE

How to negotiate the SLA for stream-processing systems has been rarely discussed in the previous research. It also depends on which platform the system is running on, and how stakeholders are involved. But an inherent requirement is to achieve cost-efficiency, which translates to minimizing the monetary cost for the users, as well as reducing the operational cost for the provider (possibly data centers). Achieving SLA-compliance requires the stream-processing system to be equipped with the ability to trade-off between the justifiable metrics, such as performance and robustness, with the running cost, the balance of which should be left for the user to decide when signing up for SLA.

As the stream data from the IoT background tends to be more dynamic and bursty, it would also be interesting to investigate the possibility of providing probabilistic SLA guarantees rather than traditional rule-based promises.

8.4.4 LOAD BALANCING

Currently, the applied load-balance schemes are very simplistic, the major target of which is to normally improve the performance of the system, especially by maximizing the throughput. However, the importance of load balance goes far beyond performance optimization. A wrong balancing decision may lead to unnecessary load-shedding, dropping arrived messages when the system is deemed to be overloaded, which ultimately impairs the veracity of the processing result. It is challenging to take the low-level metrics such as task capacity or lengths of thread-message-queues into consideration during the load-balancing process, but the perspective is very promising, as currently the system utilization rate is still moderate; even the stream-processing system is already saturated, where the inefficient load-balance mechanism is the culprit to blame.

8.5 CONCLUSIONS

To summarize, we have presented the emergence of stream processing as a complement to the batch paradigm, which is especially suited to the IoT context. We discussed the relationship between IoT and stream processing in the introduction, and then outlined the formal definition of stream data as

well as the associate stream-processing concept in the following section. We have also identified the unique characteristics of stream data in IoT and investigated how the processing requirements of them would affect the organization of a stream-processing system. Based on the aforementioned analysis, we presented a general architecture for such a system, and explained in detail about the history and comparison of different continuous logic-processing subsystems. The challenges and open questions for stream processing in IoT are also discussed in this chapter.

It can be concluded that the research on utilizing the stream-processing paradigm to build real-time IoT applications is gradually arousing a storm of hype. Ultimately, the prevalence of such applications requires the development of adaptive and autonomous stream-processing systems to better uncover the connotative value that is hidden within the huge volume of volatile streams.

REFERENCES

[1] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM 2008;51(1):107–13.
[2] Cattell R. Scalable SQL and NoSQL data stores. SIGMOD Rec 2011;39(4):12–27.
[3] Kitchin R. The real-time city? Big data and smart urbanism. GeoJournal 2013;79(1):1–14.
[4] Humphreys G, Houston M, Ng R, Frank R, Ahern S, Kirchner PD, Klosowski JT. Chromium: a stream-processing framework for interactive rendering on clusters. In: Proceedings of the twenty-ninth annual conference on computer graphics and interactive techniques. New York, NY, USA; 2002. p. 693–702.
[5] Taylor MG. Phase estimation methods for optical coherent detection using digital signal processing. J Light Technol 2009;27(7):901–14.
[6] McCarthy D, Dayal U. The architecture of an active database management system. In: Proceedings of the 1989 ACM SIGMOD international conference on management of data. New York, NY, USA; 1989. p. 215–224.
[7] Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. SIGMOD Rec 2005;34(4):42–7.
[8] Cugola G, Margara A. Processing flows of information: from data stream to complex event processing. ACM Comput Surv 2012;44(3):15:1–15.
[9] Chen J, DeWitt DJ, Tian F, Wang Y. NiagaraCQ: a scalable continuous query system for Internet databases. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data. New York, NY, USA; 2000. p. 379–390.
[10] Arasu A, Babcock B, Babu S, Cieslewicz J, Ito K, Motwani R, Srivastava U, Widom J. Stream: the Stanford data stream management system; 2004.
[11] Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, New York, NY, USA; 2002. p. 1–16.
[12] Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden SR, Reiss F, Shah MA. TelegraphCQ: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data. New York, NY, USA; 2003. p. 668–668.
[13] Abadi DJ, Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S. Aurora: a new model and architecture for data stream management. VLDB J 2003;12(2):120–39.
[14] Cherniack M, Balakrishnan H, Balazinska M, Carney D, Cetintemel U, Xing SY, Xing Y, Zdonik SB. Scalable distributed stream processing. CIDR 2003;3:257–68.
[15] Abadi DJ, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang JH, et al. The design of the borealis stream processing engine. CIDR 2005;5:277–89.

[16] Wu KL, Hildrum KW, Fan W, Yu PS, Aggarwal CC, George DA, Gedik B, Bouillet E, Gu X, Luo G, Wang H. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system S. In: Proceedings of the thirty-third international conference on very large data bases. Vienna, Austria; 2007. p. 1185–1196.
[17] Gedik B, Andrade H, Wu HL, Yu PS, Doo M. SPADE: the system S declarative stream processing engine. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. New York, NY, USA; 2008. p. 1123–1134.
[18] Anicic D, Fodor P, Rudolph S, Stojanovic N. EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of the twentieth international conference on World Wide Web. New York, NY, USA; 2011. p. 635–644.
[19] Neumeyer L, Robbins B, Nair A, Kesari A. S4: distributed stream computing platform. In: 2010 IEEE international conference on data mining workshops (ICDMW); 2010. p. 170–177.
[20] Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, Bhagat N, Mittal S, Ryaboy D. Storm@Twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. New York, NY, USA; 2014. p. 147–156.
[21] Qian Z, He Y, Su C, Wu Z, Zhu H, , Zhang T, Zhou L, Yu Y, Zhang Z. TimeStream: reliable stream computation in the cloud. In: EuroSys; 2013.