Journal of Big Data

RESEARCH Open Access

Composing high-level stream processing pipelines

Tanmaya Mahapatra[1,2*]

*Correspondence: tanmaya.mahapatra@tum.de
[2] Technical University of Munich, School of Medicine, Institute of Medical Informatics, Statistics and Epidemiology (IMedIS), Ismaninger Straße 22, Munich 81675, Germany
Full list of author information is available at the end of the article

Abstract

The growing number of Internet of Things (IoT) devices provide a massive pool of sensing data. However, turning data into actionable insights is not a trivial task, especially in the context of IoT, where application development itself is complex. The process entails working with heterogeneous devices via various communication protocols to co-ordinate and fetch datasets, followed by a series of data transformations. Graphical mashup tools, based on the principles of flow-based programming paradigm, operating at a higher-level of abstraction are in widespread use to support rapid prototyping of IoT applications. Nevertheless, the current state-of-the-art mashup tools suffer from several architectural limitations which prevent composing in-flow data analytics pipelines. In response to this, the paper contributes by (i) designing novel flow-based programming concepts based on the actor model to support data analytics pipelines in mashup tools, prototyping the ideas in a new mashup tool called aFlux and providing a detailed comparison with the existing state-of-the-art and (ii) enabling easy prototyping of streaming applications in mashup tools by abstracting the behavioural configurations of stream processing via graphical flows and validating the ease as well as the effectiveness of composing stream processing pipelines from an end-user perspective in a traffic simulation scenario.

Keywords: Flow-based programming, Graphical pipelines, Mashup tools, Graphical stream processing, Stream analytics, End-user programming, Data Analytics as a Service (DAaaS), Data Analytics Applications (DAAs)

Introduction

The steady growth of the Internet of Things (IoT) devices has resulted in a massive pool of sensing data. These data need to be processed both close to their sources (IoT devices or nearby servers) and remote data centres to derive insights that can help formulate important, often business-critical or socially impactful, decisions. For instance, sensing and processing the location and speed of cars in a city can help understand the mobility pattern of users in a city and provide services that monitor and prevent traffic congestions in the city.

Currently, data processing is embodied in data analytics applications (DAAs) that are developed and operated via a variety of tools and approaches. The IoT-centric approach emphasizes the heterogeneity of devices and software stacks in the IoT domain and addresses those by offering domain-specific languages and tool-kits for developing and deploying DAAs in a hassle-free way. This approach has seen relative success in the use of graphical mashup tools such as IBM's Node-RED [2, 3] for composing DAAs by specifying the data flow between pre-existing sensor, actuator, and service components, all in a graphical environment that even non experts can use. On the contrary, the Big Data-centric approach emphasizes the need for scale and flexibility and addresses those by offering cluster-based computational frameworks such as Spark [4, 5], Flink [6–8], and Kafka Streams [9]. While it lifts some of the current constraints of data flow programming tools (such as difficulty of specifying parallel computations), this approach makes it hard for non experts to develop DAAs. In particular, although some of graphical tools for developing Spark and Flink applications have started to emerge [10–14], they have seen so far limited adoption.

In our work, we aim to combine the intuitiveness and easy of use, brought by IoT mashup tools, while not sacrificing the flexibility needed to develop non-trivial DAAs, brought by Big Data frameworks. To this end, in the past years, we have investigated the paradigms and concepts of both IoT mashup tools and Big Data languages and frameworks. We have found that the widespread IoT mashup tools, although usually good enough for prototyping DAAs, have a number of limitations that prevent their broader adoption in real-life scenarios where DAAs are needed [15, 16]. At the same time, we have found that Big Data frameworks would benefit from a graphical frontend, similar to an IoT mashup tool, for specifying DAAs.

As a result, we have designed and implemented a new mashup tool for developing DAAs called *aFlux*. aFlux tries to strike a balance between ease of use and flexibility and has been designed to allow non experts to specify real-life DAAs via a graphical frontend tool. It offers support for both specifying blocking and parallel computations, and offers stream processing constructs such as processing windows (present in Big Data frameworks such as Flink and Spark Streaming). aFlux can be used to specify both actor-based Java applications that can run on an IoT device or on a server and Spark and Flink jobs that can run on a remote cluster. It therefore provides a common abstraction layer for DAAs.

In this paper, we describe the main concepts behind aFlux and focus on the way it supports IoT stream processing. We also provide an evaluation of its ability to model real-life DAAs related to a traffic analytics case. For a description of how aFlux can be used in combination with Spark and Flink, we refer the interested reader to our previous papers [17–21].

Structure The rest of the manuscript is structured in the following way: Sect. "Results" summarises the important results of the manuscript. Section "Background and related work" summarises the background information and the state-of-the-art. Section "aFlux concepts and architecture" describes the concepts necessary to overcome the limitations prevalent in the state-of-the-art to support high-level composition of stream processing pipelines. Section "Experimental" evaluates the practicality of the built-in parametric stream processing concepts via realistic use cases in real-time traffic control of highways. Section "Discussion" discusses the results of the experiments. It also compares and contrasts the new concepts discussed in the manuscript with the existing state-of-the-art. Finally, the manuscript ends with concluding remarks described in Sect. "Conclusion".

Results

The main contribution of this paper is to propose a novel tool concept to integrate IoT mashups and scalable stream processing, based on the actor model. We show that several new concepts to control synchronous versus asynchronous communication and parallelism are essential for this. Furthermore, we show that several parameters, such as the window type and size, impact the effectiveness of stream analytics. Importantly, such parameters impact not only the performance of stream processing (e.g. whether data of certain size can be processed within a specific time-bound), but also determine the functional behaviour of the system (e.g. whether the logic that is based on stream analytics is effective or not).

To summarise, this paper introduces essential concepts to support data analytics in flow-based programming paradigm and caters to the research contributions by:

1 Designing of new graphical flow-based programming concepts based on the actor model with support for concurrent execution semantics to overcome the prevalent architectural limitations in the state-of-the-art mashup tools and thereby supporting in-built user-configurable stream processing capabilities for simplified in-flow stream analytics. The new concepts have been thoroughly compared with all relevant existing solutions in Sect. "Discussion".
2 Parametrizing the control points of stream processing in the tool enables non-experts to use various stream processing styles and deal with the subtle nuances of stream processing effortlessly. The effectiveness of parametrization in simplifying quick prototyping of stream analytics applications has been validated in a real-time traffic use case and discussed in Sect. "Experimental".

In comparison to the paper published at VL/HCC'18 [1], this paper extends the previous work by the following additional contributions:

1 The experimental section has been extended to include a new stream processing method for analysis of mean speed and contains additional analysis of shoulder-lane state for all stream processing methods discussed (Sect. "Experimental").
2 The working concepts for aFlux described very briefly in the conference paper have been described in detail. Additionally we describe the state-of-the-art and do an extensive qualitative comparison with state-of-the-art here (Sect. "Discussion").

Background and related work

This section describes the concepts necessary to understand the work. It is divided into three sub-sections. Sub-section "Mashups and mashup tools" describes mashups and mashup tools while Sub-section "Stream processing" describes the fundamentals of stream processing. Sub-section "Related work" lists the existing relevant works and solutions which aim to support easy development of stream processing applications.

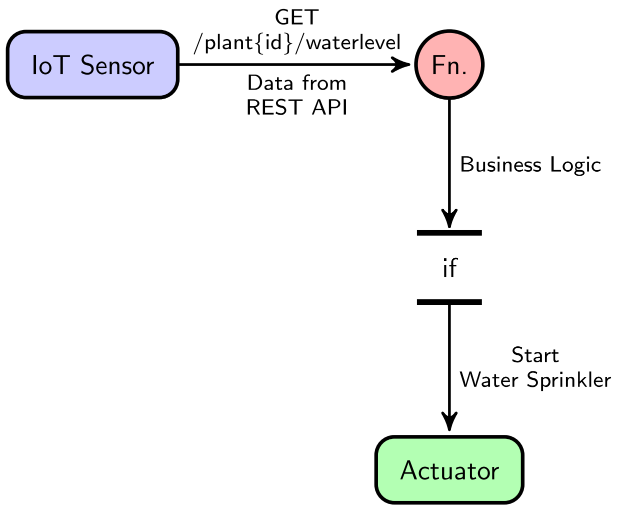


Fig. 1 Skeleton of a mashup application

Mashups and mashup tools

Mashups are defined as a conglomeration of several accessible as well as reusable components on the web. The individual conglomerated components in a mashup are known as the mashup components while the orchestration of control-flow between the components is known as the mashup logic. Typically, the control-flow from one component to the next succeeding component invariably includes data-flow after subjected to data transformations during the execution of a specific business logic abstracted within the mashup component. The components are the building blocks of a mashup, and they can provide either logic/functionality in the form of reusable algorithms or data-sets fetched from Web APIs or even reusable User-Interface parts to be used in the mashup application. A description of how typical mashup functions will make things clear. For instance, consider that the soil moisture data is available from an IoT device with the help of REST APIs. A user wants to get this data, apply some transformation, check the moisture level of the soil and maybe switch on the water sprinkler if it is below a particular threshold value. The mashup depicting the flow for this scenario is given in Fig. 1. The "Fn." block in the figure contains code (business logic, illustrated by the "if" block) which accomplishes the data transformations as well as houses the business logic. The orchestration of 3 components namely data from REST API, a function block and a connection to an actuator, i.e. water sprinkler block clearly depicts how the control flows through them in a coordinated fashion to achieve specific objectives. These components are generally represented by GUI blocks in a mashup tool which must be connected suitably to represent the entire business logic. Typically, control flow from one component to the next in a mashup also includes flow of data subjected to transformations.

Certain mashup tools provide simulation tools and also interoperability for messaging between different platforms [22]. Two prominent IoT mashup tools are WoTKit Processor [23] and Node-RED [2, 3]. WoTKit is a multi-user system for running data flow programs in the cloud while Node-RED is a tool-kit for developing data flows on devices and servers. There are many IoT platforms which include a mashup tool for service

Table 1 Batch vs stream processing

Parameters	Batch processing	Stream processing
Amount of data	Big data	Small + big data
Type of computation	Complex (must access all data)	Simple
Focus	Latency	Throughput
Processing time	Collect and process later	(Near-) real time
Used by	Hadoop mapreduce, spark	Spark streaming, flink
Use cases	Daily total turnover	Instant fraud detection

orchestration including glue.things [24], Thingstore [25], OpenIoT [26], ThingWorx [27], and Xively [27].

Stream processing

In stream processing, data is not stored; rather, it is analysed as soon as it produced. It is the analysis of real-time data-sets called streams. Streams are produced due to a number of reasons like user transactions, continuous sensor readings, social media feed etc. Stream analytics has a significant business impact in-contrast to batch/traditional data analytics. In the case of batch analytics, data is stored and later analysed, thereby allowing organisations to react to a past event. In contrast to this, in the case of stream analytics, the data is processed as soon as it is produced. This allows to quickly identify potential problems and mitigate them. Additionally, certain application scenarios are feasible only with stream analytics, e.g. updating a driver on the road with current traffic situation and suggesting new routes to a destination. Apache Spark and Apache Flink are the most popular stream analytics solutions available and leveraged as of today. Table 1 summarises the main differences between batch and stream processing.

The focus of this paper is working with the core concepts of stream processing like creation of windows, different types of windows, buffer management strategy etc. which are explained in detail in Sub-section "In-flow stream processing". The paper does not focus on the workings of stream analytics suite like Apache Flink or Apache Spark Streaming.

Related work

We did not find any mashup tool or other research works which support generic high-level programming for stream processing to be used in the context of IoT, i.e. ingest data produced from IoT sensors and run analytics on them. Nevertheless, we list the relevant solutions aimed to support easy development of stream processing applications but are difficult for less-skilled programmers to use due to interaction with low-level internals.

IBM Infosphere Streams: IBM Infosphere®Streams is a platform designed for Big Data stream analytics and uses IBM Streams Processing Language (SPL) as its programming language [28]. It is designed to achieve high throughput as well as shorter response times in stream analytics. The key idea is to abstract the complexities in developing a stream processing application and the aspects of distributed computing by allowing the user to use a set of graph operators. The application developed can be translated automatically to C++ and Java. SPL treats the application flow as a streams graph where the edges represent continuous streams and vertices represent stream operators. Stream operators are either transformers, sources or sinks. It is not designed as a visual-flow based language though it models the application in the form of a graph for reasons of expressiveness [29]. It is a complete language and not a stream processing library within a non-streaming language to have improved type checking and optimisation. SPL has 2 main elements in its language construct, i.e. streams and operators. Operators without any input streams are called as sources while operators without any output streams are called sinks. The operators have their own threading and get executed when there is at least one data item in their input stream, i.e. in the edge of the stream graph. The data-items leave an operator in the same sequence in which they had arrived after processing. SPL is issued to develop streaming applications as well as batch applications because both the computing paradigms of Big Data are implemented by dataflow graphs. Additionally, SPL allows defining composite operators to support programming abstraction and enable the development of application involving thousands of operators. It has a robust static type system and minimises implicit type conversions. Every operator can specify the behaviour of its ports, i.e. port mutability. For instance, an operator can define that it does not modify data items arriving on its input port but may permit a downstream operator to alter the same. SPL also makes use of control ports in addition to input and output ports which are used in the feedback loop. There are three main paradigms for stream processing:

Synchronous data-flow (SDF): In this paradigm, every operator has a fixed rate of data-output items per input data-items, i.e. both the cardinality of input and output sets are static and well-known. Examples include StreamIt [30] and ESTEREL [31]. This paradigm is not efficient in real-world scenarios as the input and output sets cannot be known in advance, which leads to optimization issues.
Relational Streaming: This paradigm models the relational model from databases and allows to use operators like select, join, aggregate etc. on data-items. Examples include TelegraphCQ [32], the STREAM system underlying CQL (Continuous Query Language) [33], Aurora [34], Borealis [35, 36], StreamInsight [37] and SPADE (Stream Processing Application Declarative Engine) [38].
Complex Event Processing (CEP): This paradigm treats input streams as raw events. It produces output streams as inferred events, i.e. uses patterns to detect and gather insights. Examples include NiagaraCQ [39] and the SASE (Stream-based and Shared Event processing) [40].

SPL is not based on the SDF paradigm as it allows dynamic input and output rates for each operator. It is based on the relational model and can support CEP with the inclusion of a CEP library within an operator.

StreamIt: StreamIt [30] is a dedicated programming language for writing streams application following the paradigm of synchronous data-flow stream processing. It allows modelling the application in the form of a graph where vertices are operators and edges represent streams. The most basic operator is a *'Filter'* which has one input and one output port. The rate of data ingestion, as well as data production, is static and pre-defined before the execution, which is one of the major disadvantages and restricts its usage to real-world scenarios. Additional programming constructs like *'Pipeline'*, *'SplitJoin'* and *'FeedbackLoop'* are used in conjunction to a 'Filter' to form a communicating network. A 'Pipeline' is used to define a sequence of streams, while a 'SplitJoin' is used to split and join streams. Similarly, the 'FeedbackLoop' operator is used to specify loops in a stream. Representation of a stream application via arbitrary graphs, i.e. a network of filters connected via channels is difficult to visualize and optimize. The main advantage is that it imposes a well-defined structure on streams which ensures a well-defined control flow within the stream graph. It follows the constructs of flow-based programming paradigm. Still, it does not offer any high-level graphical constructs to write stream applications.

The QualiMaster Infrastructure Configuration (QM-IConf) tool:

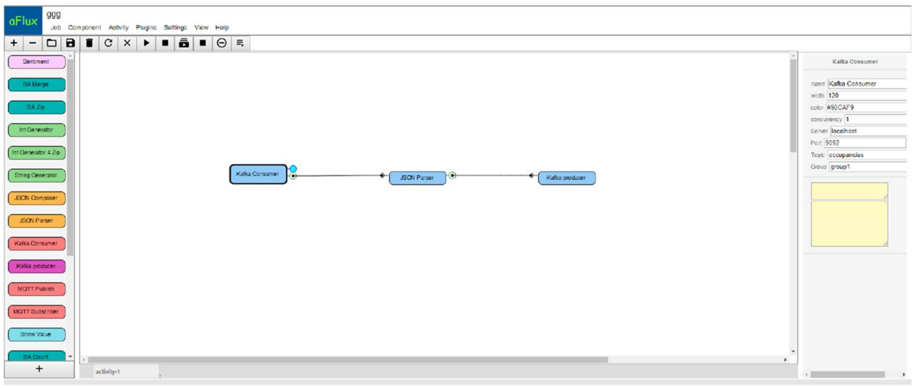
The QM-IConf tool [11] supports the model-based development of Big Data streaming applications. It introduces a high-level programming concept on top of Apache Storm [41]. It features a graphical-flow based modelling of the streaming application in the form of a dataflow graph where vertices are stream operators, and edges represent valid dataflow paths. A valid dataflow path from vertex $v1$ to $v2$ ensures that $v2$ can consume the data produced by $v1$. The dataflow model, consisting of data sources, sinks and operators, is translated into an executable Storm code. Nevertheless, it does not validate its claimed generic modelling approach against other streaming frameworks like Flink or Spark Streaming. Additionally, it supports only a specific subset of stream analytics operators to be used in the pipeline.

IBM SPSS Modeller: IBM SPSS Modeller provides a graphical user interface to develop data analytics flows involving simple statistical algorithms, machine learning algorithms, data validation algorithms and visualisation types [42]. SPSS Modeller provides machine learning algorithms developed using Spark MLlib library which can be launched on Spark cluster by simply connecting them as components in a flow. Although SPSS Modeller is a tool built for non-programmers to perform data analytics using pre-programmed blocks of algorithms, it does not support writing new custom data analytics application.

We compare and contrast these existing solutions with our developed concepts in Sect. Discussion.

aFlux concepts and architecture

Existing mashup tools allow users to design data flows which have synchronous execution semantics. This can be a significant obstacle since a data analytics job defined within a mashup flow may consume a considerable amount of time, causing other components to starve or get executed after a long waiting time. Hence, *asynchronous execution patterns* are essential for a mashup logic to invoke an analytics job (encapsulated in a mashup component) and continue to execute the next components in the flow. In this case, the result of the analytics job, potentially computed on a third party system, should be communicated back to the mashup logic asynchronously. Additionally, mashup tools restrict users in creating single-threaded applications which are generally not sufficient to model complex repetitive jobs.



name Kafka Consumer
width 120
color #90CAF9
concurrency 1
Server localhost
Port 9092
Topic occupancies
Group group1

Fig. 2 aFlux: Graphical User Interface

Requirements

The requirements behind aFlux, i.e. coming up with improved design concepts for flow-based programming tools (a.k.a. mashup tools), is to support the following concepts:

1 support creation of multi-threaded applications (sect. Programming paradigm for multi-threaded applications)
2 support asynchronous execution of components (sect. Asynchronous execution of components)
3 concurrent execution of components in flows (sect. Concurrent execution of components)
4 support for modelling complex flows via flow hierarchies (sub-flows) (sect. Logical structuring units)
5 support inbuilt stream processing (sect. In-flow stream processing)
6 model Big Data analytics via graphical flows and translate the flows to native Big Data programs [17, 18, 21]

aFlux has been designed to meet the above requirements that offer several advantages compared to existing solutions. It primarily aims to support in-flow Big Data analytics when graphically developing services and applications for the IoT. Figure 2 shows its graphical front-end used to create services and applications.

In the rest of this Section, the concepts and architectural decisions to support requirements 1-5 are reviewed in turn.

Programming paradigm for multi-threaded applications

Based on previous analysis, we decided to go with the *actor model* [43, 44], a paradigm well suited for building massively parallel [45, 46], distributed and concurrent systems [47, 48]. Actors communicate with each other using asynchronous message passing [49]. The actor model was originally a theoretical model of concurrent computation [50]. The actor model is one of the ways of realizing the dataflow programming paradigm, which is a special case of flow-based programming [51].

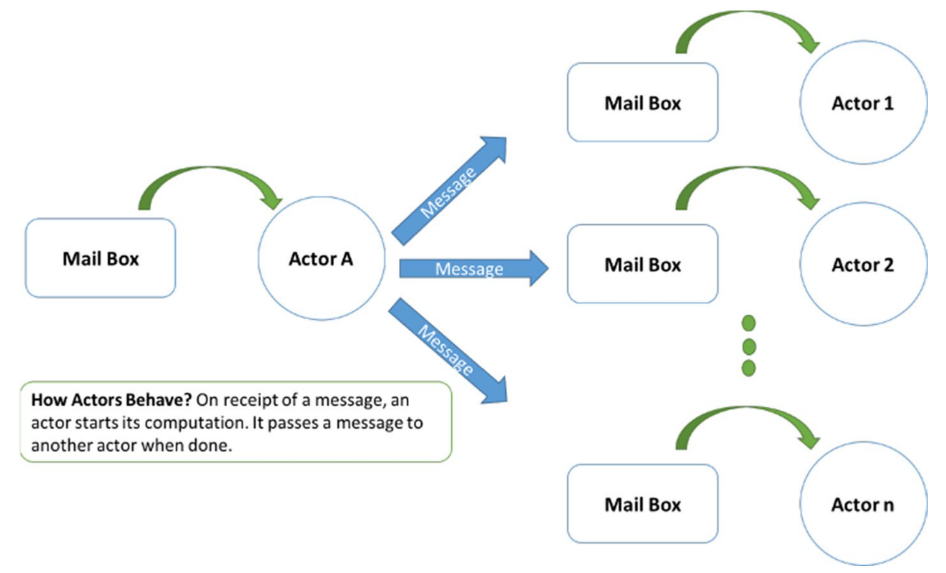

Mail Box
Actor A
Message
Message
Message
Mail Box
Actor 1
Mail Box
Actor 2
Mail Box
Actor n
How Actors Behave? On receipt of a message, an actor starts its computation. It passes a message to another actor when done.

Fig. 3 Actor model: working, as in [52]

In the actor model (Fig. 3), *an actor is the foundation of concurrency* or rather like an agent which does the actual work. It is analogous to a process or thread. Actors are very different from objects because, in an object-oriented programming paradigm, an object can interact directly with another object, i.e. changing its values or invoking a method. This causes synchronization issues in multi-threaded programs, and additional synchronization locks are necessary to ensure the proper functioning of the program [50]. In contrast to this, the actor model provides no direct way for an actor to invoke or interact with another actor. Actors respond to messages. In response to a message, an actor may change its internal state, perform some computation, fork new actors or send messages to other actors. This makes it a unit of static encapsulation as well as concurrency [53]. *Message passing between actors happens asynchronously.* Every actor has a mailbox where the received messages are queued. An actor processes a single message from the mailbox at any given time, i.e. synchronously. During the processing of a message, other messages may queue up in the mailbox. A collection of actors, together with their mailboxes and configuration parameters, is often termed an *actor system*. To summarise, the actor model is based on the following principles:

1 There is no shared, mutable state between actors. Actors exchange immutable messages to communicate with each other. Only an actor has access and can control its own state.
2 Each actor has a queue (mailbox) where the incoming messages arrive. The actor picks and processes the messages from its queue one-by-one (i.e. synchronously) and responds by sending immutable messages to other actors.
3 Messages are passed between actors asynchronously. This means that the sender does not wait for the message to be sent, and it can continue its execution. Messages

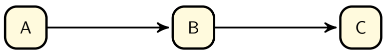


Fig. 4 A typical mashup flow, as in [52]

exchanged between actors provide no guarantees on the sequence of arrival and execution.

4 Communications between actors are asynchronous and decoupled. This allows the actors to run and execute their tasks in different threads. Therefore, they provide a concurrent and scalable model.

The main intuition is that *when a user designs a flow, the flow is modelled internally in terms of actors* i.e. an actor is a basic execution unit of the mashup tool. For instance, the flow depicted in Fig. 4 corresponds to three actors namely A, B and C with the computation starting with actor A. On completion, it sends a message to actor B and so on.

In the realization of aFlux, Akka [54], a popular library for building actor systems in Java and Scala, has been used. Since Akka can be configured in many different ways for parallel and distributed operations and governs how the actors would be spawned and executed. This shields the actors from worrying about synchronization issues.

Flow execution

In aFlux, a user can create a mashup flow called *flux*. A flux is analogous to a flow in IBM Node-RED. The only requirement for designing a flux is that it should have a start node and an end node. A flux by default is tied down to a logical unit called *job*. Every job can have one or more fluxes. When a job containing a flux like in Fig. 4 is designed in aFlux, the control flows through several parties before final execution. Firstly, the job must be saved, which allows the mashup tool to parse the flux diagram created on the front-end by the user. The parsing involves creating and saving a graph model for the job—the *Flux Execution Model*. The parser does not care how many fluxes are present in the job because it scans for special nodes in it. These special nodes are *start nodes* i.e. *specialized actors which can be triggered without receiving any message.* Other nodes are normal actors which react to messages. On detection of all start nodes in an activity, the graph model is built by simply traversing the connection links between the components as designed by the user on the front-end. A flux execution model of a job contains as many graphs as the number of fluxes present in it.

On deployment, the control flows from the front-end to the controller responsible for starting the actual execution of the job. This involves invoking the runner, which fetches the flux execution model of the job. For every flux in the job, the runner environment proceeds to:

1 Identify the relevant actors present in the graph.
2 Instantiate an actor system with the actors identified in step 1.
3 Trigger the start nodes by sending a signal.

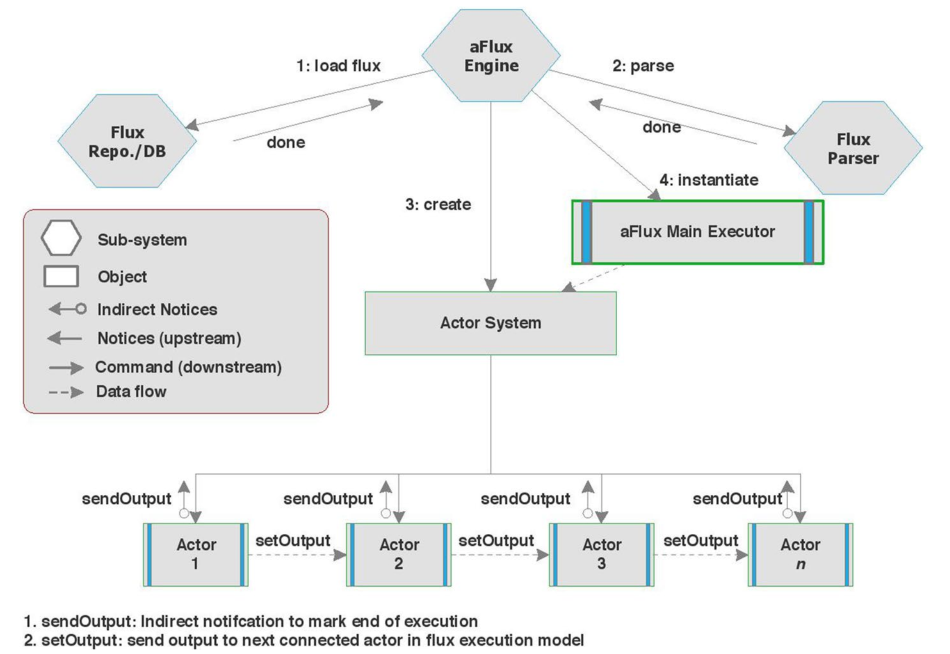


Fig. 5 Flow execution in aFlux: IDAR representation, as in [52]

After this, *the execution follows the edges of the graph model*, i.e. the start actors upon completion send messages to the next actors in the graph, which execute and send messages to the subsequent actors and so on.

The diagram depicted in Fig. 5 is an *Identify, Down, Aid, and Role* (IDAR) *graph* [55] which summarizes the execution of a flux consisting of components i.e. actors within aFlux. IDAR graphs offer a more simple and understandable way of representing how system components communicate and interact in comparison to Unified Modelling Language (UML) [55]. In an IDAR graph, objects typically communicate either by sending a *command message* (control messages) or a *non-command message*, which is called notice. The controlling objects always remain at a higher level in the hierarchy compared to the objects being controlled. An arrow with a bubble (circle) on its tail stands for an indirect method call while a dotted arrow indicates data-flow. Other subsystems having their own hierarchy are represented with hexagons denoting the subsystem manager.

The execution of a flux typically follows the following sequence:

1 When the user executes a flux, the main component in the back-end called as *'aFlux Engine'* sends a command to the *'Flux Repository'* subsystem which reads the stored *'flux execution model'* and returns it.
2 *'aFlux Engine'* sends a command to the *'Parser'* subsystem for parsing. The *'flux execution model'* is checked for consistency, i.e. if the first component in the flux contains an actor whose *'method of invocation'* property has been defined as *'triggered by system'*. The relevant actors used in the flux are identified, and after completion of this operation, the *'aFlux Engine'* is notified.
3 The *'aFlux Engine'* creates an actor system where the actors used in a flux would be executed.

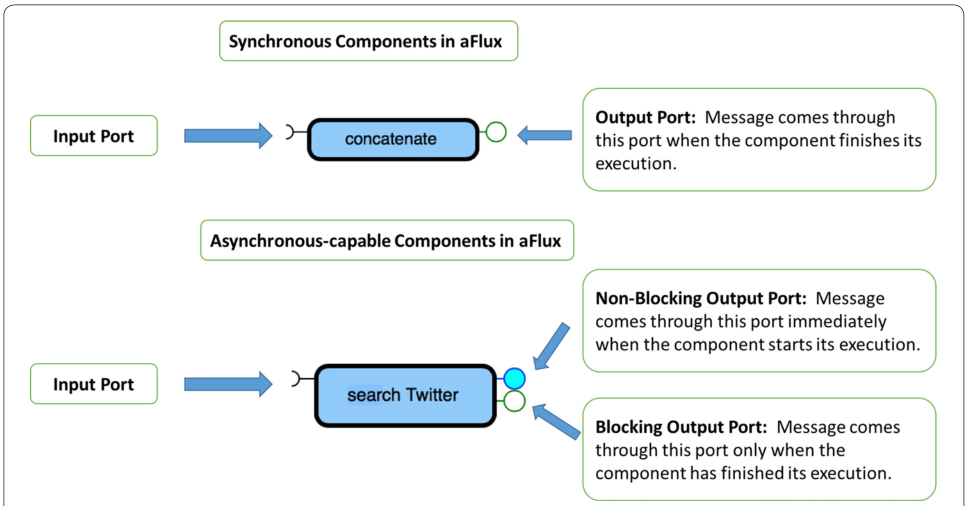

Synchronous Components in aFlux
Input Port
concatenate
Output Port: Message comes through this port when the component finishes its execution.
Asynchronous-capable Components in aFlux
Non-Blocking Output Port: Message comes through this port immediately when the component starts its execution.
Input Port
search Twitter
Blocking Output Port: Message comes through this port only when the component has finished its execution.

Fig. 6 Executable components in aFlux

4 The *'aFlux Engine'* instantiates *'aFlux Main Executor'* by passing the set of actors to be executed.
5 This data flows from *'aFlux Main Executor'* to *'Actor System'* where the relevant actors are instantiated and the first actor is triggered by the *'Actor System'*.
6 The first actor completes its execution, notifies to the *'Actor System'* about its completion of execution by sending a notification via an indirect method call and at the same time sends its output to the next connected actor.
7 This process is repeated until the last actor. When it notifies the *'Actor System'* about its completion of execution, then the *'Actor System'* removes all inactive actors and frees up memory.

Asynchronous execution of components

Components within aFlux are of two types: *synchronous* components and *asynchronous capable* components. Synchronous components block the execution flow, i.e. when they receive a message on their input port, they start execution and pass the message through their output ports upon completion. On the other hand, asynchronous-capable components have two different types of output ports, namely blocking and non-blocking ports (Fig. 6). When these components receive a message on their input port, they immediately send a message via the non-blocking port (at most one per component) so that components connected to the non-blocking port (i.e. components that do not require the computation result of the active component) can start their execution. When the component finishes its execution, it sends messages via its blocking ports; components connected to these ports can then start their execution. This non-blocking execution paradigm helps the end-user to asynchronously execute time-consuming parts of the mashup flow while ensuring other components do not get starved from execution for a longer time period.

Concurrent execution of components

Every *component* in aFlux has a special *concurrency parameter* attached to it which can be configured by the user while designing a flux. The idea is that in an actor system, every actor processes one message at a time. During its processing, new messages are queued on their arrival. To avoid this and facilitate faster processing, every component in aFlux can be made to execute concurrently by specifying the upper threshold value of concurrency. If a component has concurrency level of *n*, messages arrive quickly, and the component takes quite some time to process a message, then the actor system can spawn multiple instances of that component to process the messages concurrently up-to *n* or up-to-the global threshold value defined in the system, whichever is minimum. Beyond that, the messages are queued as usual and processed whenever any instance finishes its current execution. This specification of concurrency parameter is applicable to individual components as well as sub-flows in aFlux and is decided by the user creating a flux. In the case of sub-flows, basically, all the components used within it adhere to the concurrency limit of the sub-flow which means that the actor system can spawn multiple instances of every component used inside the sub-flow as the need arises during run-time.

Logical structuring units

To abstract away independent logic within a main application flow, the system supports logical structuring units called *sub-flows*. A sub-flow encompasses a complete business logic and is independent of other parts of the mashup. A good candidate for a sub-flow is for example a reusable data analytics logic which involves specifying how the data should be loaded and processed and what results should be extracted. Sub-flows in the system are like normal asynchronous-capable components. They have input and two sets of output ports (i.e. blocking and non-blocking). They encompass within themselves a complete flow of graphical components.

In-flow stream processing

The flow-based structure of mashup tools, i.e. passage of control to the succeeding component after completion of execution of the current component is very different from the requirements of stream processing where the component fetching real-time data (aka the listener component) cannot finish its execution. It must listen continuously to the arrival of new datasets and pass them to the succeeding component for analysis. Also, the listener component has many behavioural configurations which decide when and how to send datasets to the succeeding component for analysis.

In aFlux, *the actor model has been extended to support components which need to process streaming data.* The implementation of streaming components relies on the Akka streams library, an extension of the Akka library. Applications based on Akka streams are formulated as building blocks of three types: *source*, *sink*, and *flow*. The source is the starting point of the stream. Each source has a single output port and no input port typically. Data is fetched by the source using the configuration parameters specified, and it comes out from its output and continues to the next component that is connected to the source. The sink is basically the opposite of the source. It is the

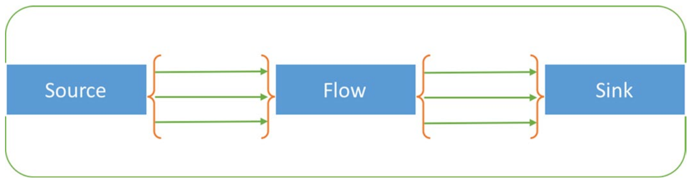


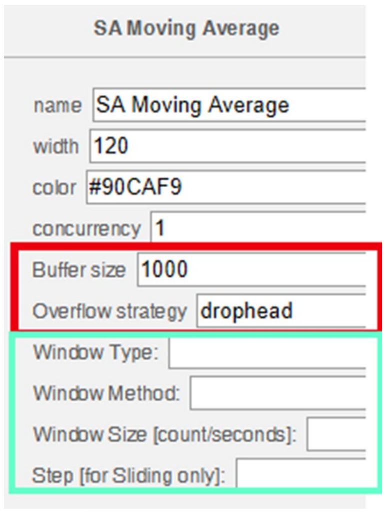
Fig. 7 Runnable flux with streaming actors, as in [52]

endpoint of a stream and therefore consumes data. Basically, it is a subscriber of the data sent or processed by a source. The third component, the flow, acts as a connector between different streams and is used to process and transform the streaming data. The flow has both inputs and outputs. A flow can be connected to a source, the outcome of which results in a new source or even after a sink which creates a new sink. A flow connected to both a source and a sink results in a runnable flux (Fig. 7), which is the blueprint of a stream.

Each streaming component in aFlux offers a different stream analytics functionality (e.g. filter, merge) and can be connected to other stream analytics components or to any common aFlux component. The stream analytics capabilities make use of three categories of components, i.e. *fan-in*, *fan-out* and *processing* components. *Fan-in* operations allow joining multiple streams into a single output stream. They accept two or more inputs and give one output. *Fan-out* operations allow splitting the stream into substreams. They accept one stream and can give multiple outputs. *Processing* operations accept one stream as an input and transform it accordingly. They then output the modified stream, which may be processed further by another processing component. The transformation of the stream is done in real-time, i.e. when the stream is available on the system for processing and not when it is generated at the source. Every component is internally composed by a source, a flow and a sink. When a component is executed by aFlux, a blueprint that describes its processing steps is generated. The blueprints are only defined once, the very first time the component is called, e.g. create a queue where the new incoming elements of the stream get appended for a component to process.

Every stream analytics component has some attributes that can be adjusted by the user at run-time. For example, for the processing components, the user can optionally define windowing properties such as window type and window size. The internal source of every stream analytics component has a *queue* (buffer), the size of which can be defined by the user (default is 1000 messages). The queue is used to temporarily store the messages (elements) that the components receive from its previous component in the aFlux flow while they are waiting to get processed. Along with the queue size, the user may also define an *overflow strategy* that is applied when the queue size exceeds the specified limit. Figure 8 shows the interface of aFlux where the user can define buffer size and overflow strategy. The overflow strategy determines what happens if the buffer is full, and a new element arrives. It can be configured as:

drop buffer:	drops all buffered elements to make space for the new element.
drop head:	drops the oldest element from the buffer.
drop tail:	drops the newest element from the buffer.
drop new:	drops the new incoming element.


SA Moving Average

name SA Moving Average
width 120
color #90CAF9
concurrency 1
Buffer size 1000
Overflow strategy drophead
Window Type:
Window Method:
Window Size [count/seconds]:
Step [for Sliding only]:

Fig. 8 aFlux GUI to specify buffer size, overflow strategy & window parameters, as in [52]

The internal flow part of a streaming component describes its logic and defines its behaviour. This is where the whole processing of messages takes place. The source sends the messages directly to the flow when it receives them. As soon as the processing of a message has finished, the result is then passed to the sink. By default, the analysis of messages is done in real-time, and each message is processed one-by-one (e.g. count how many cars have crossed a given junction). However, the user can also select windowing options.

Figure 8 shows the interface of aFlux where the user can define windowing properties. The implementation supports *content-based* and *time-based* windows. For both of these types of windows, the user can also specify a *windowing method* (tumbling or sliding) and also define a *window size* (in elements or seconds) and a sliding step (in elements or seconds).

In a nutshell, a window is created as soon as the first element that should belong to this window arrives, and the window closes when the time or its content surpasses the limit defined by the user. A window gathers all messages that arrive from the source until it is closed completely. Finally, the component applies the required processing on the data in the window. It passes the result(s) to the sink. The first thing is to choose whether the window should be content or time-based. A content-based window has a fixed size of a number of elements n. It collects elements in a window and evaluates the window when the n^{th} element has been added. On the other hand, a time-based window groups elements in a window based on time. The size of a time window is defined in seconds. For example, a time window of size 5 seconds will collect all elements that will arrive in 5 seconds from its opening and will apply a function to them after 5 seconds have passed.

In stream analytics, there are different notions of time like:

processing time: windows are defined based on the wall clock of the machine on which the window is being processed.

event time: windows are defined with respect to timestamps that are attached

	to each element.
hybrid time:	combines processing and event time.

In the current implementation, the processing time is used to interpret time in our processor. For instance, a time window of size 60 seconds, will close exactly after 60 seconds. After deciding on using content or time windows, the user has to decide how to divide the continuous elements into discrete chunks. Here the user has the following two options. The first is tumbling window, where stream elements are divided into non-overlapping parts, and each element can only belong to a single window. The second option is a sliding window, which is parametrized by length and step. These windows overlap, and each element may belong to multiple windows. Windows can be either tumbling or sliding. A tumbling window tumbles over the stream of data. This type of window is non-overlapping, which means that the elements in a window will not appear in other windows. A tumbling window can be either content-based (e.g. "Calculate the average speed of every 100 cars") or window-based (e.g. "Find the count of tweets per time zone every 10 seconds") whereas a sliding window slides over the stream of data. Due to this reason, a sliding window can be overlapping, and it gives a smoother aggregation over the incoming stream since it does not jump from one input set to the other, but it slides over the incoming data. A sliding window has an additional parameter which describes the size of the hop. A sliding window can as well be either content-based (e.g. "For every 10 cars calculate the average speed of the last 100 cars") or time-based (e.g. "Every 5 seconds find the count of tweets per time zone in the last 10 seconds"). Thus if the sliding step is smaller than the window size, elements might be assigned to multiple successive windows. The tumbling window can be conceived as a special case of a sliding window, where the window size is equal to the sliding step. Therefore, it does not make any sense to define a sliding step for a tumbling window.

The sink is the final stage of a stream analytics component. The sink gets the results from the flow and decides the final outcome. In this case, the results need to be sent to the next component in the flux because the components should be able to pass messages to each other.

Experimental

The experimental has been designed to capture that writing a performant stream analytics application is always use-case specific and takes a few iterations. We aim to simplify this by not requiring the user to write any code, select parameters easily for stream analytics and thereby support quick prototyping. An open-source traffic simulation software by the name SUMO [56] has been used to demonstrate the stream processing capabilities of aFlux. The data generated from the system is random, making it a perfect fit for real-time analytics and the results of the analytics affect the system performance, i.e. traffic congestion in SUMO. For the evaluation purposes, the traffic of A9 highway[1] near Munich has been used for simulation. TraCI [57], a python-based interface, has been used for data-exchange between SUMO and Kafka [58–60].

[1] A9 public GitHub project, available at https://github.com/iliasger/Traffic-Simulation-A9.

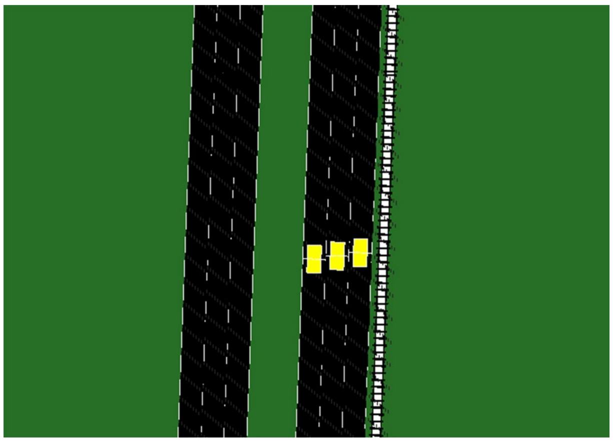


Fig. 9 The 4 lanes used in the experiment. Each of the 3 left lanes possess a loop detector. The 4*th* lane is initially closed

Scenario: In the scenario, all cars run on a straight line on the A9 highway and in the same direction from south to north. At a certain point on the highway, there are four lanes, three of which possess a loop detector (see Fig. 9). Loop detectors measure the occupancy rate (0-100) on the lane, i.e. how long was a car placed on the loop detector during the last tick (one tick equals to one second of simulation time in SUMO). A high occupancy rate signals a more busy lane and therefore, the possibility of traffic congestion. The fourth lane of the highway, further referred to as shoulder-lane, is initially closed, which means that no cars can run on it. However, if the total average of the occupancy rates of the three other lanes exceeds the threshold of 30, the shoulder-lane opens to reduce the traffic. When the average of the occupancy rates falls below 30, the shoulder-lane closes again. On the 500*th* tick of the simulation, it is assumed that a car accident happens. A lane, ahead of the four previously mentioned lanes, gets closed at the same moment and remains closed for the rest of the experiment. This builds up congestion on the highway, causing the occupancy rates of the loop detectors to increase and makes it meaningful to open the shoulder-lane at some point to alleviate the congestion.

Goal: The goal of this experiment is to compare different stream processing methods on data coming from the simulation environment. Notably, the one-by-one method for processing data, tumbling window processing with three different window sizes (50, 300 and 500) and sliding window processing are compared. The user can define the method of data processing and change various associated parameters on aFlux UI. The loop detector occupancy, lane state, mean speed of cars and time values from SUMO are captured via TraCI and published to Kafka. The lane state is a binary value that indicates the state of the shoulder-lane at the current tick(0 means closed and vice-versa). Mean speeds are used as an indicator of traffic congestion, i.e. a low mean speed on a lane indicates traffic congestion. The average mean speeds of the three lanes are plotted to demonstrate the effect of the shoulder-lane in the relief of traffic congestion. Finally, time measures the duration the shoulder-lane state takes to reach 1 for the first time and the duration it needs to reach 0 again for the last time. This factor indicates the responsiveness of each method on traffic changes, e.g. how fast the system perceives and reacts

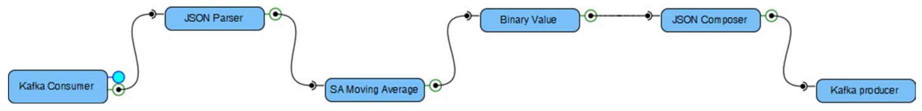


Fig. 10 aFlux flow used in the experiment - subscribes to a Kafka topic that publishes the occupancy rates of loop detectors and calculates their moving average in real-time

to traffic congestion. All runs of the experiment are based on the same conditions. The routes of the cars and the way that they are simulated in the simulation have the same randomness for all runs; therefore do not impact the experiment results. The only factor that influences the results of the experiments is the decision to open and close the shoulder-lane.

Flow-based data analytics: For the reliability of the results, the experiment has been run twice for every stream processing method. In order to make decisions to change the state of the shoulder-lane based on the occupancy of the loop detectors, a flow in aFlux has been designed (Fig. 10). The first component of the flow is a Kafka subscriber that listens to the topic where TraCI publishes the occupancy rates of each loop detector on every tick. The data is parsed using a JSON parser, and the results are passed to the moving average component. The moving average component receives the occupancy values and calculates their average on real-time and based on the user-specified method (windowing or simple processing). The results are then passed to the binary value component, which outputs 0 if the average is below the user-defined threshold (e.g. 30) or 1 otherwise. Finally, the result is transformed into a JSON file and published to a Kafka topic, where TraCI listens, to decide whether to open or close the shoulder-lane.

Evaluation parameters: The data coming from SUMO is analysed in several stream processing methods via a number of configurable parameters. The result of such analytics affects the performance of the system, i.e. SUMO. To measure the effect on system performance, the following aspects are considered:

Responsiveness	indicates how fast the system can detect traffic congestion and open the shoulder-lane to alleviate it.
Settling time	refers to the time the system needs to reach a steady state [61]. In the experiment, the shoulder-lane may open and close successively. We define the settling time as the time the shoulder-lane needs to reach a steady-state after a change occurs. It is estimated based on the shoulder-lane state parameter.
Stability	refers to the ability of the system to reach a stable state without overshoots when a change occurs. Overshoot occurs when the system exceeds a certain target point before convergence [61]. In our case, stability is tested when the shoulder-lane changes state. Stability is in inverse proportion to settling time, i.e. short settling time infers to higher system stability.

The occupancy rates of the loop detectors have not been considered for result-analysis since they are used to make decisions in aFlux, and the focus is to examine the impact of these decisions to other factors in a traffic system.

Analysis of Mean Speed:

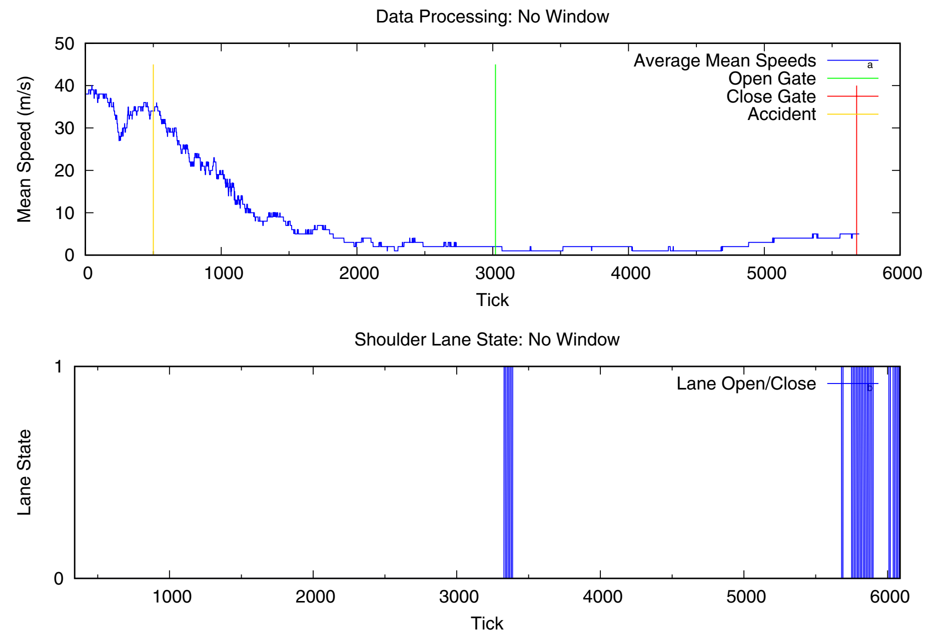


Fig. 11 Average mean speed of cars moving on the 3 lanes. (a) Data analysis without windows (one-by-one), (b) Shoulder-lane state without windows (one-by-one)

First, for every processing method, the average of the mean speeds of the cars moving on the 3 previously mentioned lanes, per tick is analysed. The mean speed of the vehicles running on a lane at a certain point of time discloses information about the current congestion of this lane. Through this analysis, the responsiveness of each method to changes and their effectiveness to solve a problem is determined; in this case, to alleviate the traffic congestion.

From Fig. 11 (a) when the accident happens at tick 500, the average mean speeds of cars moving on the 3 particular lanes that we examine, falls significantly. This means that congestion starts to build-up on these lanes. The loop detectors send their occupancy rates to aFlux every tick, and they are getting averaged by the moving average component one-by-one. Since we do not use any window to process the incoming data, each average occupancy value depends on all previous occupancy values, even on the low occupancy rates before the accident. As a result, the moving average value cannot reflect new environment changes fast enough and hence it reaches the threshold of 30 on the 3020^{th} tick for the first time to open the shoulder-lane. By observing the Fig. 11 (a) one can see an up-trend of the average speeds on the 4600^{th} tick, but the moving average value falls below 30 only on the 5680^{th} tick for the last time when the shoulder-lane gets closed as well, a fact that shows a slow reaction time.

Figure 12 (a) shows the average mean speeds of the lanes when using a content-based tumbling window of size 50 to process the occupancy rates of the loop detectors. Using a window of size 50 means that only the 50 latest occupancy values are aggregated and averaged and that the average does not keep the state of the previous values. We consider 50 to be a small window size, as it lasts for about 17 ticks. The difference in reaction time to non-window processing is significant since the system perceives much earlier

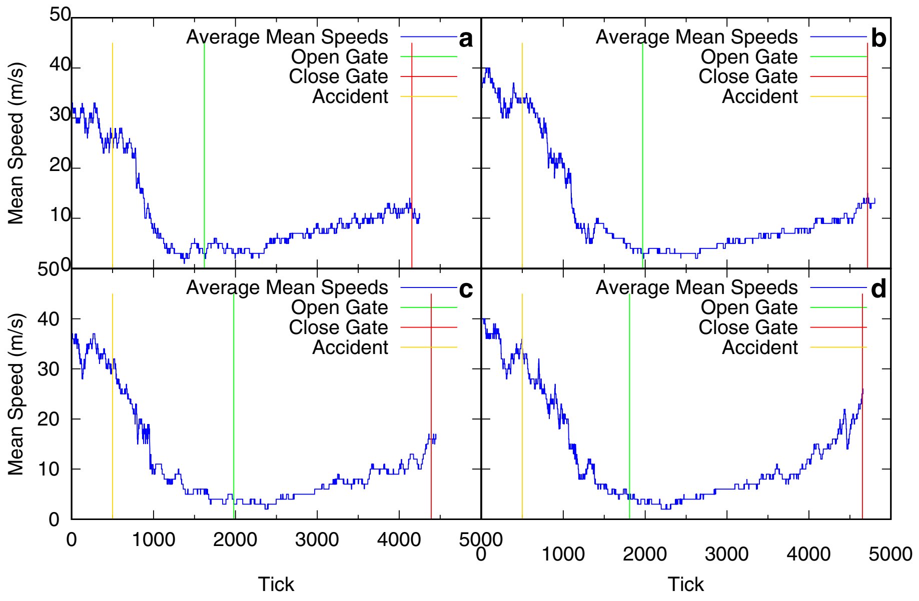


Fig. 12 Data analysis with content-based tumbling window of size **a** 50, **b** 300, **c** 500, and **d** content-based sliding window of size 500 and step 250

the traffic congestion and opens the shoulder-lane on tick 1620 for the first time. When the traffic congestion is alleviated, the system closes the shoulder-lane on the 4145^{th} tick which is also a much faster reaction in comparison to 5680 ticks that it took for the non-window processing.

Figure 12 (b) depicts the average mean speeds of cars when using a content-based tumbling window of size 300. Using this method, the system opens the shoulder-lane on tick 1970 and closes the shoulder-lane on the 4715^{th} tick for the last time. This processing method responds to changes faster than the no-window processing but a bit slower than the tumbling window of size 50. This performance is expected since a larger window size takes longer to aggregate more values (100 ticks) and hence adapts slower to changes in comparison to smaller window sizes. In Fig. 12 (c), we present the results of the tumbling window with size 500 (167 ticks). In comparison to window size 300, this processing method is slightly slower (shoulder lane opens at tick 1979 and closes at 4391). The distribution of the mean speeds is quite similar to window size 300 though which suggests that the system shows similar behaviour in both cases. Figure 12 (d) shows a sliding window of size 500 with a sliding step of 250. The difference of the previous window processing is that the sliding window takes into consideration the previous state as well by overlapping on previous values. In our case, the sliding window overlaps the 250 latest elements of the previous window. In general, a sliding window gives smoother and in some cases faster results, since it is moving faster (emits more values than a tumbling window). By comparing the figures of the sliding and tumbling window of size 500, one can observe that the distribution of the average of the mean speeds is quite similar in both graphs. The sliding window seems to be faster in opening the shoulder lane for the first time (tick 1810), but it is a bit slower in closing it (tick 4652). We consider this differentiation to be dependent on the variation of data in each experiment.

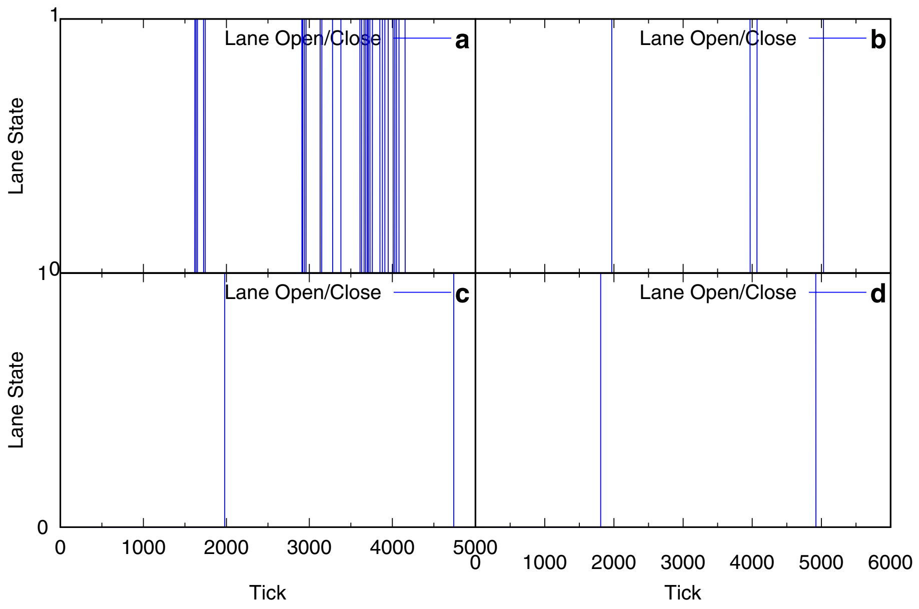


Fig. 13 Shoulder-lane state. 0 means lane closed, 1 means lane open. Data analysis with content-based tumbling window of size **a** 50, **b** 300, **c** 500, and **d** content-based sliding window of size 500 and step 250

Analysis of Shoulder-lane state: The shoulder-lane state depends entirely on the average of the occupancy rates of the lanes. If the average occupancy rate is above 30 the shoulder-lane state turns to 1 (lane opens) otherwise it is 0. The analysis of the state of the shoulder lane shows the variability of each method. When the occupancy rate reaches 30, it may climb above 30 (overshoot). Then it may fall below 30 (undershoot) again on the next tick. In control theory, overshoot refers to an output that exceeds its target value. In contrast, the phenomenon where the output is lower than the target value is called undershoot. In our case, it is reasonable to have an overshoot as we expect the occupancy rates to rise above 30. Still, here we want to examine the overshoot followed by an undershoot ratio which leads to an unstable state where the shoulder-lane opens and gets closed on successive ticks. We also focus on the settling times of each method. A stable system must have short settling times [62], i.e. converge quickly to its steady value, and must not overshoot.

Figure 11 (b) shows the variation of the state of the shoulder lane when occupancy rates are processed one-by-one. On tick 3020 the lane opens for the first time, and we observe an overshoot-undershoot case which lasts for 3 ticks before the lane state value settles on 1. Thus, the settling time when the shoulder-lane opens for the first time is 3 ticks. When the traffic is about to be alleviated, and just before the shoulder-lane closes for the last time on tick 5680, we see another overshoot-undershoot incident with a longer settling time. Concerning the variation of the shoulder-lane state, Fig. 13 (a) shows that this particular method has much overshoot and undershoot incidents causing the shoulder-lane to open and close many times successively. This fact implies an unbalanced system with long settling times. We can attribute this lack of stability to the small window size, which is sensitive to the behaviour of a small sample of data. Figure 13 (b) depicts the shoulder-lane states during the experiment. In this method, there

Table 2 Stream analytics method characteristics

Method	Responsiveness	Settling time	Stability
No window	Very slow	Long	Low
Tumbling window 50	Very fast	Very long	Very low
Tumbling window 300	Slow	Very short	High
Tumbling window 500	Slow	None	Very high
Sliding window 500	Slow	None	Very high

is no big variation between the states and almost no overshoot-undershoot incidents. The settling time is short, and the system seems to be balanced. The big window size allows the system to make a decision, based on a bigger sample of data and hence it is more stable than the two previously mentioned methods.

In Fig. 13 (c), the results of the tumbling window with size 500 can be seen. This window size is considered very big, and it is used as an extreme case here. As expected, there are no overshoot-undershoot incidents, and the system seems to be very balanced. The shoulder-lane opens and closes only once when needed, and there is no settling time. This is the best window size compared to the previous one. In the following section, we will examine the same window size for the sliding version of the tumbling window. Figure 13 (d) shows the results of the sliding window with size 500 and sliding step 250. As expected, there are no overshoot-undershoot incidents here as well, and the system is balanced. The shoulder-lane opens and closes only once when needed, and there is no settling time. This figure is quite similar to the corresponding one of the tumbling window, which implies that there no big difference between a tumbling and a sliding window in this case concerning the stability of the system.

Discussion of the experimental results

The results of various stream analytics methods based on their performance in solving a traffic control problem in real-time are summarized in Table 2. By observing the table, two points become evident: (i) stream processing can be done in various ways, (ii) these methods perform differently thereby affecting the final outcome and performance of the application. For instance, in the above traffic use-case a small window processing method, like the tumbling window of size 50, showed very good responsiveness as it was the fastest method to open and close the shoulder-lane when traffic congestion occurred. Still, it showed poor stability since its settling times were the longest of all five methods. On the other hand, the no-window processing method is the slowest method to perceive and respond to a change in the environment (e.g. traffic congestion). This method also has low stability since it is prone to long settling times and overshoot-undershoot incidents. From Table 2, it can be stated that the most efficient method to control traffic in our scenario is a tumbling window of normal size (not too big or small). Still, it will take further iterations to define the ideal window size.

What has been evaluated? First, When data needs to be processed in real-time, and the result of such analysis impact the final outcome, i.e. performance of the application, there is no easy way to know the right stream processing method with the correct parameters. Hence, it becomes very tedious to manually write the relevant code and

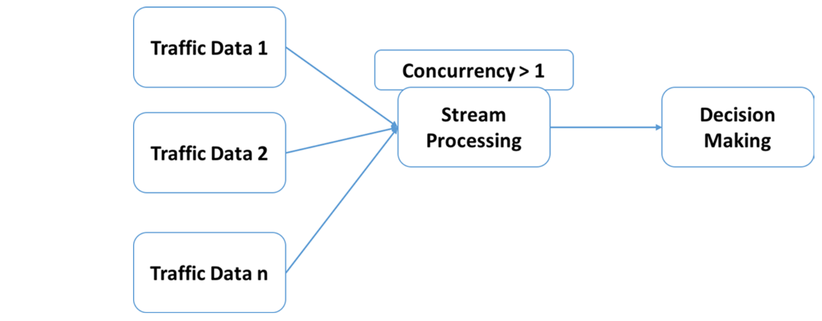


Fig. 14 Concurrent processing of streaming data

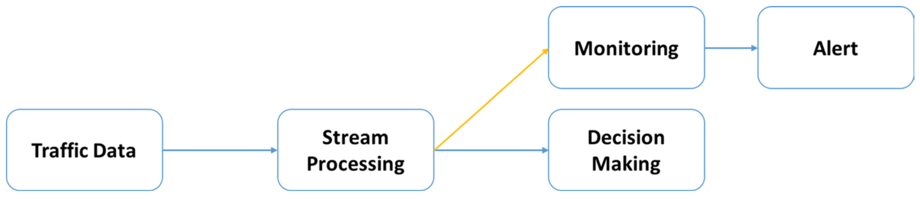


Fig. 15 Asynchronous processing of streaming data

re-compile every time a user wants to try something new. By parametrizing the controlling aspects of stream processing, it becomes easy for non-experts to test various stream processing methods to suit their application needs.

Overall, the following aspects of aFlux have been captured via the example — the integrated stream processing capabilities in a flow, parametrization of the buffer capacity and overflow strategies and modelling of different kinds of window methods to process data, i.e. tumbling and sliding windows. *Second*, having stream processing components within aFlux allows users to quickly prototype their stream processing applications without relying on external stream processing suites. It becomes easier to prototype streaming applications, test them and finally port them to stream analytics suites like Flink. *Third*, the concurrent execution semantics of aFlux fits really well to real-world scenarios as shown in Fig. 14. If we have traffic data coming from all over the city, the streaming processing component can be made to execute concurrently to handle those transactions very quickly and efficiently. *Last*, the user can also monitor the impact of the streaming pipeline on the application by running a monitoring process asynchronously, as shown in Fig.15.

Discussion

This section compares our concepts (as in aFlux) with exiting solutions supporting flow-based stream analytics. In Sect. Related work, we have discussed relevant existing solutions which deal with supporting flow-based data analytics, mainly stream processing. The relevant and comparable tools include StreamIt, IBM Streams Processing Language or the IBM Infosphere Streams, QM-IConf and IBM SPSS modeller. We also consider Node-RED for comparison as it is one of the prominent platforms used for flow-based programming in the context of IoT and is widely supported by IBM.

Table 3 Comparison of aFlux with existing solutions, as in [52]

Tools	Concurrent execution	Data analytics app. development	Flows with built-in stream processing	Execution of each component in separate threads	Scaling up of individual components	Parametrization of component buffer	Streaming paradigm
aFlux	Yes	Batch jobs. Streaming is a special case	Yes	Yes	Yes	Yes	Actor model with support for CEP and relational streaming paradigm
StreamIt	Yes	Streaming only. Batch is a special case	No	Unknown	No	No	Synchronous dataflow paradigm
IBM Infosphere Streams	Yes	Streaming only. Batch is a special case	No	Yes	Unknown	No	Relational streaming paradigm with support for CEP
IBM SPSS	Yes	Streaming only. Batch is a special case	No	Unknown	No	No	Appears to follow relational streaming paradigm
QM-IConf	No	Streaming only. Batch is a special case	No	No	No	No	Relational streaming paradigm
Node-RED	No	No support for streaming	No	No	No	No	Not Applicable

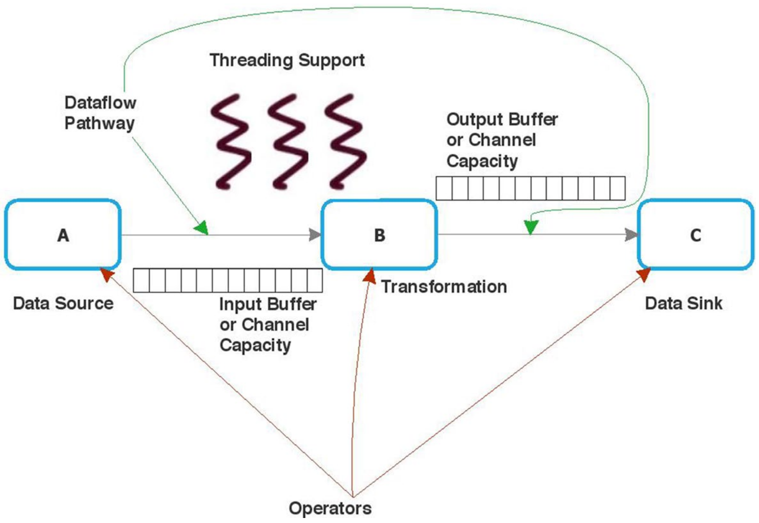


Fig. 16 Streaming application modelled as a dataflow graph with vertices representing operators and edges representing dataflow pathways, as in [52]

First, we begin by defining the scope and the parameters to compare. Figure 16 shows the model of a streaming application. This model is also known as dataflow model. Every stream application in high-level programming tools is modelled as a dataflow graph where vertices represent operators and edges represent valid dataflow pathways.

Operators can either be data sources, data sinks or data transformers. The edge between two operators has a channel capacity Accordingly, we compare the following key parameters between the solutions (Table 3 summarises the compared parameters):

Concurent execution of components: aFlux is based on actor-model hence it supports concurrent execution semantics, i.e. one operator in the dataflow graph can start its execution in parallel before the finish of its predecessor. All tools with exception of Node-RED and QM-IConf support asynchronous execution of data operators in the dataflow graph. *The second column in Table 3 lists this criterion.*

Data analytics application development: A second criterion for comparison is the kind of data analytics application that can be developed using these tools. *The third and fourth column in Table 3 list this criterion.* aFlux relies on asynchronous message passing techniques to model the dataflow graph and extends the actor model to support continuous streaming of datasets. This allows a user to model batch analytics application, a completely streaming application or a batch application with some components doing stream analytics. Streaming is considered as a special case in aFlux while other platforms treat modelling of batch jobs as a special case. Other solutions are stream only platforms and are not designed to specify batch analytic jobs though they can be modelled as both stream and batch analytics rely on the dataflow paradigm [63].

Component execution and scaling: One of the important criterion for comparison is to see if the individual components used in an application flow are executed in separate threads and can scale. *The fifth and sixth column in Table 3 list this criterion.* Since operators, i.e. components in aFlux, are independent from other components it supports scaling up of instances of a specific component. Additionally, each actor instance of the same component is a different unit of computation hence boots performance. Other solutions do not support scaling up of instance of a component if the data load increases. Nevertheless, IBM Infosphere Streams specifies separate thread of execution for each data operators in a dataflow graph.

Parametrisation of Buffers and overflow strategy: The streams of data travelling from one operator to another are stored in a buffer queue before being processed. Customisation of this buffer is an important parameter for comparison. *The seventh column in Table 3 lists this criterion.* aFlux supports parametrisation of this buffer and specify over-flow strategies. This concept is non-existent in all existing solution and from the experimental in Sect. Experimental it is clear that optimisation of this aspect affects the performance of the streaming application.

Stream processing paradigm: *The eighth column in Table 3 lists the criterion of streaming paradigm the tool is built upon.* There are three major stream processing paradigms: (i) synchronous dataflow where the data ingestion and data production rates are predefined which makes it unrealistic in real-world use cases, (ii) relational paradigm which relies on the concept of relational databases to process streams which simplifies stream processing and (iii) complex event processing paradigm in which operators detect patterns in input streams and infer outputs. Platforms supporting relational are easier to model stream applications, scale-well while the complex event processing paradigm permits to model complex streaming applications.

aFlux supports the relational and the complex event processing paradigm while StreamIt supports only synchronous dataflow paradigm.

Conclusion

In this paper, we described the requirements, main concepts, and architectural decision behind aFlux, a new mashup tool for graphical composition of Data Analytics Applications (DAAs). aFlux is inspired by IoT mashup tools such as NodeRED and tries to provide a common abstraction layer for the development of DAAs that combines flow-based programming with Big Data analytics. Since it can be used for specifying parallel and asynchronous computations and it provides several built-in stream processing capabilities, aFlux can be used for developing complex, real-life DAAs. We have demonstrated its usage and provided initial evidence of its effectiveness in composing stream processing pipelines from an end user perspective in a traffic use case. In the future, we would like to evaluate its usability in a user study.

Abbreviations

API: Application Program Interface; CEP: Complex Event Processing; CQL: Continuous Query Language; DAaas: Data Analytics as a Service; DAG: Directed Acyclic Graph; IBM: The International Business Machines Corporation; IDAR: Identify, Down, Aid, and Role; IoT: Internet of Things; QM-IConf: The QualiMaster Infrastructure Configuration tool; REST: Representational state transfer; SASE: Stream-based and Shared Event processing; SDF: Synchronous data-flow; SPADE: Stream Processing Application Declarative Engine; SPLv: Streams Processing Language; SPSS: Statistical Package for the Social Sciences; UML: Unified Modelling Language.

Acknowledgements

The author is grateful to all his colleagues who actively provided insights, reviews and comments leading to the materialisation of this work.

This paper is an extension of the conference paper titled “Stream Analytics in IoT Mashup Tools”, published in the conference proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 2018 [1]

Authors’ contributions

TM did this work as part of his PhD dissertation. The author read and approved the final manuscript

Funding

Open Access funding enabled and organized by Projekt DEAL. This work was supported by the German Research Foundation (DFG) and the Technical University of Munich within the Open Access Publishing Funding Programme.

Availability of data and materials

The datasets generated and/or analysed during the current study are available in the aFlux repository,https://www.aflux.org/images/data/sumo-experiment-data.zip.

Competing interests

The author declares that he has no conflict of interest.

Author details

[1] Technical University of Munich, Software and Systems Engineering Research Group, Boltzmannstraße 03, Garching 85748, Germany. [2] Technical University of Munich, School of Medicine, Institute of Medical Informatics, Statistics and Epidemiology (IMedIS), Ismaninger Straße 22, Munich 81675, Germany.

Received: 5 June 2020 Accepted: 3 September 2020
Published online: 29 September 2020

References

1. Mahapatra T, Prehofer C, Gerostathopoulos I, Varsamidakis I. Stream Analytics in IoT Mashup Tools. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 2018; 227–231. Available from: https://doi.org/10.1109/VLHCC.2018.8506548.
2. Health N. com T, editor. How IBM’s Node-RED is hacking together the internet of things; 2014. [Online; posted 13-March-2014]. http://www.techrepublic.com/article/node-red/.
3. IBM. Node-RED, Flow-based programming for the Internet of Things;. [Online; accessed 10-May-2016]. http://nodered.org/.

4. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster Computing with Working Sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10. USA: USENIX Association. 2010;10.
5. Zecevic P, Bonaci M. Spark in Action. 1st ed. Greenwich, CT, USA: Manning Publications Co.; 2016.
6. Katsifodimos A, Schelter S. Apache Flink: Stream Analytics at Scale. In: 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW). 2016; 193–193.
7. Friedman E, Tzoumas K. Introduction to Apache Flink. : O'Reilly; 2016.
8. Apache. Flink Programming Concepts;. [Online; accessed 09-May-2019]. https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/programming-model.html.
9. Apache Kafka. A Distributed Streaming Platform; 2018. [Online; accessed 24-April-2019]. https://kafka.apache.org.
10. Santos Wd, Avelar GP, Ribeiro MH, Guedes D, Meira W Jr. Scalable and Efficient Data Analytics and Mining with Lemonade. Proc VLDB Endow. 2018; 11(12):2070–2073. https://doi.org/10.14778/3229863.3236262.
11. Eichelberger H, Qin C, Schmid K. Experiences with the Model-based Generation of Big Data Pipelines. In: Mitschang B, Nicklas D, Leymann F, Schöning H, Herschel M, Teubner J, et al., editors. Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband. Bonn: Gesellschaft für Informatik e.V; 2017. p. 49–56.
12. StreamSets. DataOps for Modern Data Integration; 2018. [Online; accessed 18-May-2020]. https://streamsets.com.
13. Touk. Nussknacker. Streaming Processes Diagrams;. [Online; accessed 27-May-2019]. https://touk.github.io/nussknacker/.
14. Stratio. Sparta 2.0: The definitive visual build tool for Apache Spark; 2018. [Online; accessed 18-May-2020]. https://www.stratio.com/blog/apache-spark-visual-tool-sparta/.
15. Mahapatra T, Gerostathopoulos I, Prehofer C. Towards Integration of Big Data Analytics in Internet of Things Mashup Tools. In: Proceedings of the Seventh International Workshop on the Web of Things. WoT '16. New York, NY, USA: ACM. 2016; p. 11–16. https://doi.org/10.1145/3017995.3017998.
16. Mahapatra T, Prehofer C. Service Mashups and Developer Support. Digital Mobility Platforms and Ecosystems. 2016; 48 – 65. https://doi.org/10.14459/2016md1324021
17. Mahapatra T, Gerostathopoulos I, Fernández FA, Prehofer C. Designing Flink Pipelines in IoT Mashup Tools. 2018;2316(03):41–53 http://ceur-ws.org/Vol-2316/paper3.pdf.
18. Mahapatra T, Gerostathopoulos I, Prehofer C, Gore SG. Graphical Spark Programming in IoT Mashup Tools. In: 2018 Fifth International Conference on Internet of Things: Systems, Management and Security; 2018. p. 163–170.https://doi.org/10.1109/IoTSMS.2018.8554665.
19. Mahapatra T, Prehofer C. aFlux: Flow-based programming for Big Data; 2019. [Online; accessed 20-June-2019]. https://aflux.org.
20. Mahapatra T, Prehofer C. aFlux: Graphical flow-based data analytics. Software Impacts. 2019;2:100007. https://doi.org/10.1016/j.simpa.2019.100007.
21. Mahapatra T, Prehofer C. Graphical Flow-based Spark Programming. Journal of Big Data. 2020;7(1):4. https://doi.org/10.1186/s40537-019-0273-5
22. Prehofer C, Chiarabini L. From Internet of Things Mashups to Model-Based Development. In: COMPSAC, 2015 IEEE 39th Annual. IEEE; 2015. p. 499 – 504.
23. Blackstock M, Lea R. IoT mashups with the WoTKit. In: Internet of Things (IOT), 2012 3rd International Conference on the; 2012. p. 159–166.
24. Kleinfeld R, Steglich S, Radziwonowicz L, Doukas C. glue.things: A Mashup Platform for wiring the Internet of Things with the Internet of Services. In: Proceedings of the 5th International Workshop on Web of Things. WoT '14. ACM; 2014. p. 16–21. Available from: https://doi.org/10.1145/2684432.2684436.
25. Akpinar K, Hua KA, Li K. ThingStore: A Platform for Internet-of-things Application Development and Deployment. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. DEBS '15. ACM; 2015. p. 162–173. https://doi.org/10.1145/2675743.2771833.
26. Kim J, Lee JW. OpenIoT: An open service framework for the Internet of Things. In: Internet of Things (WF-IoT); 2014. p. 89–93.
27. Derhamy H, Eliasson J, Delsing J, Priller P. A survey of commercial frameworks for the Internet of Things. In: ETFA; 2015. p. 1–8.
28. Hirzel M, Andrade H, Gedik B, Jacques-Silva G, Khandekar R, Kumar V, et al. IBM Streams Processing Language: Analyzing Big Data in motion. IBM Journal of Research and Development. 2013 May;57(3/4):7:1–7:11.
29. Seyfer N, Tibbetts R, Mishkin N. Capture Fields: Modularity in a Stream-relational Event Processing Langauge. In: Proceedings of the 5th ACM International Conference on Distributed Event-based System. DEBS '11. New York, NY, USA: ACM; 2011. p. 15–22.https://doi.org/10.1145/2002259.2002263.
30. Thies W, Karczmarek M, Amarasinghe S. StreamIt: A Language for Streaming Applications. In: Horspool RN, editor. Compiler Construction. Springer, Berlin Heidelberg: Berlin, Heidelberg; 2002. p. 179–96.
31. Berry G, Gonthier G. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Sci Comput Program. 1992 Nov;19(2):87–152. https://doi.org/10.1016/0167-6423(92)90005-V.
32. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, et al. TelegraphCQ: Continuous Dataflow Processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. SIGMOD '03. New York, NY, USA: ACM; 2003. p. 668–668. https://doi.org/10.1145/872757.872857.
33. Arasu A, Babu S, Widom J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. The VLDB Journal. 2006 Jun;15(2):121–142. Available from: https://doi.org/10.1007/s00778-004-0147-z.
34. Abadi DJ, Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, et al. Aurora: A New Model and Architecture for Data Stream Management. The VLDB Journal. 2003 Aug;12(2):120–139. https://doi.org/10.1007/s00778-003-0095-z.
35. Çetintemel U, Abadi D, Ahmad Y, Balakrishnan H, Balazinska M, Cherniack M, et al. In: Garofalakis M, Gehrke J, Rastogi R, editors. The Aurora and Borealis Stream Processing Engines. Berlin, Heidelberg: Springer Berlin Heidelberg; 2016. p. 337–359. https://doi.org/10.1007/978-3-540-28608-0_17.

36. Cangialosi FJ, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang JH, et al. The Design of the Borealis Stream Processing Engine. In: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005). Asilomar, CA; 2005. .
37. Barga RS, Goldstein J, Ali MH, Hong M. Consistent Streaming Through Time: A Vision for Event Stream Processing. In: CIDR; 2007. p. 363–373.
38. Gedik B, Andrade H, Wu KL, 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. SIGMOD '08. New York, NY, USA: ACM; 2008. p. 1123–1134. https://doi.org/10.1145/1376616.1376729.
39. 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. SIGMOD '00. New York, NY, USA: ACM; 2000. p. 379–390. Available from: https://doi.org/10.1145/342009.335432.
40. Agrawal J, Diao Y, Gyllstrom D, Immerman N. Efficient Pattern Matching over Event Streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08. New York, NY, USA: ACM; 2008. p. 147–160. https://doi.org/10.1145/1376616.1376634.
41. Apache. Apache Storm;. [Online; accessed 27-May-2019]. https://storm.apache.org.
42. IBM. IBM SPSS Modeller;. [Online; accessed 22-June-2018]. Available from: https://www.ibm.com/products/spss-modeler.
43. Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press; 1986.
44. Hewitt C. Viewing Control Structures As Patterns of Passing Messages. Artif Intell. 1977 Jun;8(3):323–364. https://doi.org/10.1016/0004-3702(77)90033-9.
45. Agha GA, Mason IA, Smith SF, Talcott CL. A Foundation for Actor Computation. J Funct Program. 1997 Jan;7(1):1–72. https://doi.org/10.1017/S095679689700261X.
46. Talcott CL. Composable Semantic Models for Actor Theories. Higher-Order and Symbolic Computation. 1998 Sep;11(3):281–343. https://doi.org/10.1023/A:1010042915896.
47. Virding R, Wikström C, Williams M. Concurrent Programming in ERLANG (2Nd Ed.). Hertfordshire, UK: Prentice Hall International (UK) Ltd.; 1996.
48. Varela C, Agha G. Programming Dynamically Reconfigurable Open Systems with SALSA. SIGPLAN Not. 2001 Dec;36(12):20–34. https://doi.org/10.1145/583960.583964.
49. Musser DR, Varela CA. Structured Reasoning About Actor Systems. In: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control. AGERE! 2013. New York, NY, USA: ACM; 2013. p. 37–48. https://doi.org/10.1145/2541329.2541334.
50. Goodwin J. Learning Akka. Packt Publishing; 2015. https://www.packtpub.com/application-development/learning-akka.
51. Carkci M. Dataflow and Reactive Programming Systems: A Practical Guide. 1st ed. USA: CreateSpace Independent Publishing Platform; 2014.
52. Mahapatra T. High-level Graphical Programming for Big Data Applications [Dissertation]. Technische Universität München. München; 2019. http://mediatum.ub.tum.de/?id=1524977.
53. Desell T, Maghraoui KE, Varela CA. Malleable applications for scalable high performance computing. Cluster Computing. 2007 Sep;10(3):323–337. https://doi.org/10.1007/s10586-007-0032-9.
54. Akka. Implementation of the Actor Model. Build powerful reactive, concurrent, and distributed applications more easily;. [Online; accessed 25-December-2017]. https://akka.io/.
55. Overton MA. The IDAR Graph. Commun ACM. 2017 Jun;60(7):40–45. https://doi.org/10.1145/3079970.
56. Krajzewicz D, Erdmann J, Behrisch M, Bieker L. Recent Development and Applications of SUMO - Simulation of Urban MObility. Int J Adv Syst Measurements. 2012;5(3&4):128–38.
57. Wegener A, Piorkowski M, Raya M, Hellbrück H, Fischer S, Hubaux JP. TraCI: An Interface for Coupling Road Traffic and Network Simulators. 11th Communications and Networking Simulation Symposium (CNS). 2008;.
58. Minni S. Apache Kafka Cookbook. Packt Publishing Ltd; 2015.
59. Garg N. Apache Kafka. Packt Publishing Ltd; 2013.
60. Dunning T, Friedman E. Streaming architecture: new designs using Apache Kafka and MapR streams. "O'Reilly Media, Inc."; 2016.
61. Filieri A, Maggio M, Angelopoulos K, D'ippolito N, Gerostathopoulos I, Hempel AB, Control Strategies for Self-Adaptive Software Systems. ACM Trans Auton Adapt Syst. , et al. Feb; 11(4):24:1–24:31. Available from: 2017;. https://doi.org/10.1145/3024188.
62. Abdelzaher T, Diao Y, Hellerstein JL, Lu C, Zhu X. In: Liu Z, Xia CH, editors. Introduction to Control Theory And Its Application to Computing Systems. Boston, MA: Springer US; 2008. p. 185–215. https://doi.org/10.1007/978-0-387-79361-0_7.
63. Soulé R, Hirzel M, Grimm R, Gedik B, Andrade H, Kumar V, et al. A Universal Calculus for Stream Processing Languages. In: Gordon AD, editor. Programming Languages and Systems. Springer, Berlin Heidelberg: Berlin, Heidelberg; 2010. p. 507–28.

Publisher's Note