Stream Analytics in IoT Mashup Tools

Tanmaya Mahapatra*, Christian Prehofer, Ilias Gerostathopoulos and Ioannis Varsamidakis
Lehrstuhl für Software und Systems Engineering, Fakultät für Informatik
Technische Universität München
Email: *mahapatr@in.tum.de, prehofer@in.tum.de, gerostat@in.tum.de, ioannis.varsamidakis@tum.de

***Abstract*—Consumption of data streams generated from IoT devices during IoT application development is gaining prominence as the data insights are paramount for building high-impact applications. IoT mashup tools, i.e. tools that aim to reduce the development effort in the context of IoT via graphical flow-based programming, suffer from various architectural limitations which prevent the usage of data analytics as part of the application logic. Moreover, the approach of flow-based programming is not conducive for stream processing. We introduce our new mashup tool aFlux based on actor system with concurrent and asynchronous execution semantics to overcome the prevalent architectural limitations and support in-built user-configurable stream processing capabilities. Furthermore, 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. We validate the effectiveness of parametrization in a real-time traffic use case.**

***Index Terms*—Internet of Things, IoT mashup tools, graphical flows, end-users, stream analytics**

I. Introduction

With the proliferation of ubiquitous connected physical objects commonly known as the Internet of Things (IoT) there has been a steady increase in the amount of data generated. Data analysis can help us e.g. understand the mobility pattern of users in a city or monitor the city continuously for potential traffic congestion. Despite this great potential, deriving insights from data has been typically a separate process of using Big Data analytics tools while the application development is concerned mainly with the creation of user application for relevant business use cases [1].

For IoT applications, mashups have been proposed as a way to simplify the application development. Mashup tools [2], [3] are graphical tools designed for quick software development. They typically offer graphical interfaces for specifying the data flow between sensors, actuators, and services. They offer a data flow-based programming paradigm where programs form a directed graph with "black-box" nodes which exchange data along connected arcs.

The overall problem we address here is the lack of integrated tools for both IoT development and stream analytics [1], [4]. For instance, Node-RED [5], [6] is a visual programming environment developed by IBM which supports the creation of mashups. It is however not designed for developing stream analytics applications, and, for instance, the IBM cloud solution (https://www.ibm.com/cloud/) features separate graphical tools for modelling data processing and analysis.

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 important 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 evaluate our proposal, we have implemented a new Java-based tool called *aFlux*. aFlux supports concurrent and asynchronous execution of components in mashup flows, overcoming many limitations of current mashup tools such as Node-RED. It also has built-in support for stream analytics and allows users to tune the important parameters of stream processing in the tool front-end. This simplifies the task of devising and comparing different configurations of stream processing for the application at hand. We have evaluated the practicality of the built-in parametric stream processing in aFlux via realistic use cases in real-time traffic control of highways.

II. aFlux Concepts and Design

In this section, we present the main concepts of our new tool approach. Despite promised benefits in having data analytics in graphical mashup tool, there are several limitations of current approaches [1], [4]. First, existing tools allow users to design data flows which have synchronous execution semantics. This can be a major obstacle since a data analytics job defined within a mashup flow may consume great amount of time causing other components to starve or get executed after a long waiting time. Hence, asynchronous execution patterns are important in order 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. Second, mashup tools restrict users in creating single-threaded applications which are generally not sufficient to model complex repetitive jobs. Third, mashup

978-1-5386-4235-1/18/$31.00

tools use visual notations for the program logic which is not very expressive to model logic of complex analytics jobs.

To summarize, we designed our mashup tool, called aFlux, to support the following requirements:

1) asynchronous execution of components in flows;
2) concurrent, multi-threaded execution of components in flows;
3) support for modelling complex flows via flow hierarchies (sub-flows).

In designing aFlux, we decided to go with the actor model [7], [8], a paradigm well suited for building massively parallel [9], [10], distributed and concurrent systems [11], [12]. In the actor model, an actor is an agent, analogous to a process or thread, which does the actual work. Actors respond to messages, which is the only way of interaction between actors. 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 [13]. 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*.

aFlux is a web-based tool. Its front-end, implemented using the React JavaScript framework, allows users to create mashup flows via a graphical editor. The main intuition is that when a user designs a flow, we model this flow in the back-end in terms of actors making an actor the basic execution unit of our mashup tool. In the implementation of aFlux back-end we have used Akka [14], a popular library for building actor systems in Java and Scala.

A mashup flow in aFlux is called *flux*. Every time a flux is saved in the front-end, its specification is sent to the back-end where it is parsed in order to create a corresponding graph model—the *Flux Execution Model*. The parser scans for special *start nodes* in the specification of a flux. Start nodes correspond to specialized actors which can be triggered without receiving any message. All other nodes correspond to normal actors which react to messages. On detection of a start node, the graph model is built by simply traversing the connection links between the nodes as designed by the user on the front-end. On deployment of a flux, a runner fetches the flux execution model and proceeds to:

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

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 next actors and so on.

A. 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, blocking and non-blocking (Figure 1). 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 it (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 asynchronously execute time-consuming parts of the mashup flow while ensuring other components do not get starved from execution for a longer time period.

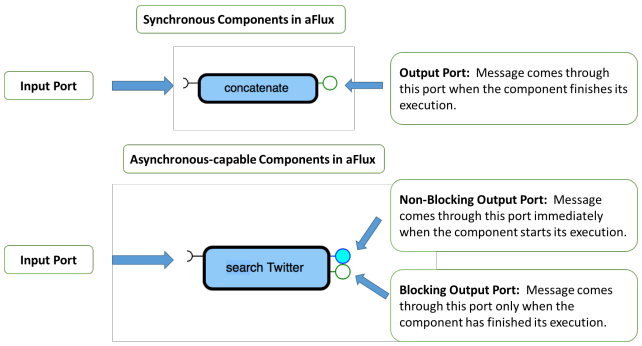

Fig. 1. Executable Components in aFlux

B. Concurrent Execution of Components

Every component in aFlux has a special configurable concurrency parameter. If a component has concurrency level of n, the actor system can spawn up to n instances of that component to process the messages concurrently. Beyond that, messages are queued as usual and processed whenever any instance finishes its current execution.

C. Sub-flows in aFlux

To encapsulate independent and reusable logic within an application flow, aFlux supports logical structuring units called *sub-flows*. A sub-flow encompasses a complete business logic and is independent from 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 are modelled as asynchronous-capable components, i.e. they have input ports and two sets of output ports (i.e. blocking and non-blocking).

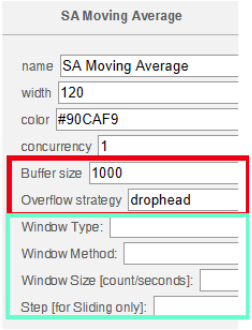

Fig. 2. Specification of buffer size, overflow strategy & window parameters.

III. Stream Processing in aFlux

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, we have introduced an abstraction called *streaming component* to model components which need to process streaming data. The implementation of streaming components relies on the Akka Streams library.

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. Streaming components are categorized into *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 sub-streams. 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.

Every stream analytics component offers a number of configurable attributes (Figure 2). 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 receives 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. It can be configured as: (i) *drop buffer:* drops all buffered elements to make space for the new element, (ii) *drop head*: drops the oldest element from the buffer, (iii) *drop tail*: drops the newest element from the buffer, (iv) *drop new*: drops the new incoming element.

TABLE I
Stream analytics method characteristics

Method	Responsiveness	Settling Time	Stability
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

The user can also specify different windowing properties. Our implementation currently 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*) (this attribute only applies to sliding windows).

IV. Evaluation

In order to evaluate the built-in stream processing capabilities of aFlux, we implemented an aFlux flow which involves stream processing to derive actionable insights. In our tested, the parameters of stream processing influence the end result. By selecting different values for such parameters and running different micro-benchmarks, we showcase the ease with which stream processing can be customized in aFlux and compare the different versions of the application.

Our test-bed is a traffic simulation of a highway[1] implemented in Python on top of traCI, a Python interface for SUMO microscopic traffic simulator [15]. In the scenario, a number of cars run on the highway which consists of three lanes. The cars pass over loop detectors placed next to each other at a particular mile of the highway. A loop detector measures the occupancy rate of the lane in the range of 0 to 100. A high occupancy rate signals a more busy lane and therefore the possibility of a traffic congestion. The highway operators have implemented a simple logic for reacting to traffic congestions in our simulation: if the average of the occupancy rates of the three lanes exceeds an empirical threshold of 30, a fourth lane (shoulder-lane) opens to reduce congestion. On the contrary, when the average of the occupancy rates falls below 30, the shoulder-lane closes again. We have implemented the above logic in aFlux (Figure 3), using Kafka to get the loop detector data from the simulation and communicate back the action of opening/closing the shoulder lane.

We have run different micro-benchmarks to compare the average speed of cars when changing the processing parameters of the stream of loop detector data. In particular, we have used the four methods depicted in Table I. In all micro-benchmarks, we artificially induce traffic congestion by an "accident" happening on the 500^{th} tick of the simulation which closes one of the three normal lanes for the rest of the experiment (each experiment took 5000 ticks).

[1] https://github.com/iliasger/Traffic-Simulation-A9

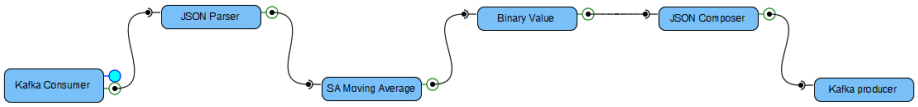

Fig. 3. aFlux flow used in the experiment - Subscribes to a Kafka topic that publishes the occupancy rates of loop detectors and calculates their moving average in real-time.

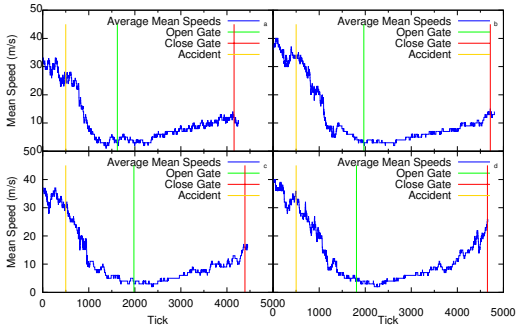

Fig. 4. Data analysis with content-based tumbling window of size (a) 50, (b) 300, (c) 500, and (d) content-based sliding window of size 500 and step 250.

The results are plotted in Figure 4 and summarized in Table I. We can observe that different methods change the time at which the extra lane is opened (*responsiveness* of the system), but they also have implications on the time when fluctuations in the shoulder-lane state end, after a change is initiated (*settling time*) and the number of fluctuations in the shoulder-lane state (*stability*). We omit the data that show the settling time and stability measurements for length constraints.

Discussion. Overall, we can make the following observations. Firstly, when data needs to be processed in real-time and the result of such analysis impacts 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 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. Secondly, 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 platforms.

V. Related Work

We have discussed some of the most popular mashup tools in Section I. Although these tools are good for modelling control-flow, nevertheless their in-flow data analytics capabilities are very limited [16], as discussed above. Additionally, the architecture of flow-based programming languages does not accommodate the requirements of stream processing as discussed earlier in Section II. IBM Watson Studio does not offer an integrated solution to develop IoT applications containing in-flow data analytics [17]. One of the closest solution is Apache NiFi which is an easy to use, powerful and a reliable system to process and distribute data. It offers a highly intuitive web-based graphical user interface which allows the user to design data flows and transform data [18]. However, it does the processing via other stream processing engines (via connectors) and does not provide options to experiment with different kinds of stream processing. Kafka Streams [19], Apache Spark [20] and Apache Flink [21] are geared for developing stream processing applications however the user has to write the code using their built-in APIs and they do not have any simplified graphical user-interface for users. In addition to this, setting up and deploying clusters to try out basic stream processing increases the learning curve substantially for non-experts.

VI. Conclusion

In this paper, we have argued the needs for stream processing in mashup tools for IoT application development. We have demonstrated how aFlux enables rapid development of applications with stream processing integrated in the application logic which makes it unique among all the current available solutions. In this direction, the goal of the paper was to (i) integrate stream processing capabilities within aFlux (ii) parametrize the controlling factors of stream processing to the tool front-end so that it becomes easy for non-experts to try out various methods of stream processing, find the impact, tweak and re-tweak to easily arrive at the optimal configuration options for their scenario. Every stream processing component in aFlux has its own adjustable settings. This parametrization based approach makes it easy for non-experts to run adjustable stream analytics jobs. Additionally, the concurrent execution and asynchronous execution semantics of the tool facilitates non-experts to develop complex real-world applications by easy abstraction. Currently, we are working towards mapping the stream processing semantics of aFlux to popular streaming frameworks like Apache Spark and Apache Flink. This would enable non-experts to prototype streaming application using the built-in stream processing of aFlux and finally deploy the flux as a full-scale Spark or Flink application.

Acknowledgement

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

References

[1] T. Mahapatra, I. Gerostathopoulos, and C. Prehofer, "Towards integration of big data analytics in internet of things mashup tools," in *Proceedings of the Seventh International Workshop on the Web of Things*, ser. WoT '16. New York, NY, USA: ACM, 2016, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/3017995.3017998

[2] F. Daniel and M. Matera, *Mashups: Concepts, Models and Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, dOI: 10.1007/978-3-642-55049-2. [Online]. Available: http://link.springer.com/10.1007/978-3-642-55049-2

[3] M. Ogrinz, *Mashup patterns: designs and examples for the modern enterprise*. Addison-Wesley, 2009, oCLC: ocn262433525.

[4] "Project consortium tum living lab connected mobility: Digital mobility platforms and ecosystems," Software Engineering for Business Information Systems (sebis), München, Tech. Rep., Jul 2016. [Online]. Available: https://mediatum.ub.tum.de/node?id=1324021;

[5] N. Health, "How ibm's node-red is hacking together the internet of things," March 2014, http://www.techrepublic.com/article/node-red/TechRepublic.com [Online; posted 13-March-2014].

[6] "IBM Node-RED, A visual tool for wiring the Internet of things." [Online]. Available: http://nodered.org/

[7] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[8] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, Jun. 1977. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(77)90033-9

[9] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1017/S095679689700261X

[10] C. L. Talcott, "Composable semantic models for actor theories," *Higher-Order and Symbolic Computation*, vol. 11, no. 3, pp. 281–343, Sep 1998. [Online]. Available: https://doi.org/10.1023/A:1010042915896

[11] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[12] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *SIGPLAN Not.*, vol. 36, no. 12, pp. 20–34, Dec. 2001. [Online]. Available: http://doi.acm.org/10.1145/583960.583964

[13] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, Sep 2007. [Online]. Available: https://doi.org/10.1007/s10586-007-0032-9

[14] "Akka: Implementation of the actor model," https://akka.io/, accessed: 2017-12-25.

[15] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MObility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[16] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of internet-of-things platforms," *CoRR*, vol. abs/1502.01181, 2015. [Online]. Available: http://arxiv.org/abs/1502.01181

[17] "Put the power of AI and data to work for your business," https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=97014197USEN, accessed: 2018-04-20.

[18] "Apahce nifi," https://nifi.apache.org/, accessed: 2017-12-25.

[19] J. Kreps, "Introducing kafka streams: Stream processing made simple," *Confluent Blog, March*, 2016.

[20] P. Zecevic and M. Bonaci, *Spark in Action*. Manning Publications Co., 2016.

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.