Distributed Reactive Stream Processing

Shweta Khare, Aniruddha Gokhale
Vanderbilt University, EECS
Nashville, TN 37212, USA
shweta.p.khare@vanderbilt.edu, a.gokhale@vanderbilt.edu

***Abstract*—**
Reactive programming paradigm successfully overcomes the limitations of observer pattern which has traditionally been used for developing event-driven distributed systems. Due to its declarative style, compositionality and automatic management of dependencies, reactive programming offers a promising new way for building complex distributed data-flow systems. This article outlines some open challenges in extending the reactive programming paradigm for distributed stream processing.

***Keywords*-Distributed Stream Processing, IoT, Reactive Programming**

I. Introduction

The *"internet of things"* (IoT) paradigm is driven by the expansion of internet to include physical objects; thereby bridging the divide between the physical world and cyberspace. Large number of devices are increasingly getting connected to the internet. As per Gartner, approximately 26 billion devices will be a part of IoT by 2020 [1]. These devices or "things" are uniquely identifiable, fitted with sensors and actuators, which enables them to gather information about their environment and respond intelligently [2]. IoT paradigm has helped realize critical infrastructures like smart-grids, intelligent transportation systems, advanced manufacturing, health-care tele-monitoring, etc. Such systems are also called Industrial IoT (IIoT) or Cyber-Physical Systems (CPSs).

Data pushed by sensors in a CPS can be visualized as a stream of tuples, which needs to be processed in a distributed and parallel manner for timely processing. This requirement has given rise to Stream Processing Engines (SPEs); a new class of applications, specifically tailored for high-volume stream data analysis. In SPEs, data is continuously processed by user-defined queries that "sit" on top of streaming data and produce results each time query predicate is satisfied [3]. A query is represented as a Directed Acyclic Graph (DAG), where vertices define operators and edges define the flow of data between these operators [3].From centralized SPEs like Aurora [4], the state of the art has advanced to Distributed Stream Processing Systems (DSPSs) like Storm [5], S4 [6], etc. to facilitate large-scale, real-time complex data analytics.

Resilience (*i.e.* tolerating faults), *responsiveness* (*i.e.* timely and predictable processing) and *elasticity* (*i.e.* accommodating growing/shrinking load patterns) - which are the desired characteristics of a dependable DSPS, require asynchrony and loose-coupling between system components [7]. Data-centric publish-subscribe technologies like Object Management Group (OMG)'s Data Distribution Service (DDS) [8], offer asynchrony and loose-coupling between publishers (data generators) and subscribers (data receivers). This makes DDS particularly well suited for distributed stream processing, where *subscription* to a publisher can be viewed as having a *continuous query* registered over streams [9].

We have developed a DSPS which integrates DDS with a reactive programming library [10], for scalable and high-performance stream processing [7]. This article describes some benefits that reactive programming offers for stream processing and discusses future directions of research.

II. Reactive Programming

Reactive languages [11] provide a dedicated abstraction for time-changing values called signals or behaviors. Hence, a reactive value (singal or behavior) can be used to represent an external/incoming source of data stream over which computations can be composed using inbuilt functional operators to produce an output reactive value (data-stream). The language runtime tracks changes to the values of signals/behaviors and propagates the change through the application by re-evaluating dependent variables automatically. Hence, the application can be visualized as a data-flow, wherein data and respectively changes thereof implicitly flow through the application [12].

Taditionally,event-driven systems have been programmed using the observer pattern [13] wherein reaction to events occurs via call-backs and inversion of control [14]. However, the observer pattern has many well-documented drawbacks [15], [16], which are addressed by reactive programming languges which offer the following benefits [12]: 1) *Declarative Style*- functional dependencies between values/streams are expressed directly thereby enhancing the readability of the code unlike the observer pattern which inverts the control flow. 2) *Composition*- reactive values can be composed to capture complex dependencies unlike callbacks which aren't composable. 3) *Automation*- language runtime automatically tracks and safely propagates changes through the application to ensure consistent re-computation [17] of dependencies. Hence, the programmer is relieved from the responsiblity of co-ordinating callbacks and ensuring

consistent updates. 4) Interoperability- Different reactive abstractions can inter-operate with each other, i.e., signals can be converted into events and back.

III. Research Roadmap

In addition to consistent propagation of changes through the distributed data-flow network, the reactive runtime should be able to monitor and auto-tune its performance in order to sustain desired throughput and latency of stream processing. We intend to systematically assess the performance of our distributed reactive data-flow system [7] with existing stream data processing systems like Samza [18], Storm [5], Spark [19], etc. and gain insights into potential areas of improvement.

For example, reactive frameworks lack backpressure to throttle the source if a downstream operator can't keep up with incoming data-rate. When this happens, the operator keeps buffering incoming samples causing unbounded increase in queue lengths, arbitrarily large response times or out-of-memory exceptions. **Reactive-Streams** [20] project has suggested a *dynamic push-pull model* for implementing backpressure. Their model can shift dynamically from being push-based (when consumer can keep up with data rate) to a pull-based model if the consumer is getting overwhelmed. The consumer specifies its "demand" using the backpressure channel to throttle the source. The producer can also use the "demand" specifications of downstream operators to perform intelligent load-distribution. Instead of throttling the source which can decrease the overall throughput, the system should be able to provision more resources or distribute load to relieve the bottleneck dynamically similar to the *HashPartition* function in **Timestream** or dynamic load balancing in **Streamcloud**. We intend to develop the reactive runtime with implicit support for load monitoring and appropriate reaction to handing load by distribution over cores and distributed nodes. To this effect, the concept of auto-parallelization [21] can also be used to identify parallel regions within a distributed node's local data-flow graph so as to distribute the processing across available cores.

Introducing concurrency is profitable only when gain in performance is greater than its overhead because of context-switching, synchronization, etc. The system should be able to auto-tune its degree of concurrency dynamically to achieve higher performance e.g. like in **SEDA** [22]. A SEDA application is composed of a set of stages much like stream-operators, connected via explicit event queues. Every stage has an associated *controller* that monitors its event-queue depth and runtime characteristics to dynamically tune its operation. For example, a *threadpool controller* adaptively configures pool size to meet perceived concurrency demands and *batching controller* dynamically adjusts batch sizes to trade-off throughput and latency.

Language level reactive abstractions for various events specific to distributed systems like node failures, network partitioning, topology changes etc. can also be developed with clear and verifiable semantics. For example, ambient references [23] represent a collection of connected publishers. This collection is kept updated to reflect the currently available set of publishers and made available to the application as a reactive value. This will allow encoding the system in response to failures, topology changes in a declarative manner and help us reason about the distributed system more easily.

We intend to incorporate the outlined research ideas in our existing solution [7] that unifies local and distributed processing aspects in a common data-flow programming model.

References

[1] "Forecast: The Internet of Things, Worldwide, 2013," http://www.gartner.com/newsroom/id/2636073.

[2] L. Coetzee and J. Eksteen, "The internet of things - promise for the future? an introduction," in *IST-Africa Conference Proceedings, 2011*.

[3] V. Gulisano *et al.*, "Streamcloud: An elastic and scalable data streaming system," *Parallel and Distributed Systems, IEEE Transactions on*.

[4] D. J. Abadi *et al.*, "Aurora: a new model and architecture for data stream management," *The VLDB Journali£¡The International Journal on Very Large Data Bases*.

[5] A. Toshniwal *et al.*, "Storm@twitter."

[6] L. Neumeyer *et al.*, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, 2010.

[7] S. Khare, S. Tambe *et al.*, "Scalable reactive stream processing using dds and rx," *ISIS*, vol. 14, p. 103, 2014.

[8] OMG, "The Data Distribution Service specification, v1.2," http://www.omg.org/spec/DDS/1.2, 2007.

[9] A. Salehi, "Design and implementation of an efficient data stream processing system," Ph.D. dissertation, IC, Lausanne, 2010.

[10] "The Reactive Extensions (Rx)," http://msdn.microsoft.com/en-us/data/gg577609.aspx.

[11] E. Bainomugisha *et al.*, "A survey on reactive programming," *ACM Comput. Surv.*

[12] G. Salvaneschi, P. Eugster, and M. Mezini, "Programming with implicit flows," *Software, IEEE*, 2014.

[13] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[14] P. Haller and M. Odersky. Springer Berlin Heidelberg, 2006, ch. Event-Based Programming Without Inversion of Control.

[15] I. Maier, T. Rompf, and M. Odersky, Tech. Rep.

[16] L. A. Meyerovich *et al.*, "Flapjax: A programming language for ajax applications," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, 2009.

[17] G. H. Cooper and S. Krishnamurthi, "Embedding dynamic dataflow in a call-by-value language," ser. ESOP'06, 2006.

[18] "Apache Samza," http://samza.apache.org/.

[19] "Apache Spark," http://spark.apache.org/.

[20] "Reactive Streams," http://www.reactive-streams.org/.

[21] S. Schneider *et al.*, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

[22] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*.

[23] A. L. Carreton, S. Mostinckx, T. Van Cutsem, and W. De Meuter, "Loosely-coupled distributed reactive programming in mobile ad hoc networks," in *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 41–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894386.1894389