D-Joseph: An Efficient Approach for Dynamic Software Reconfiguration in Data Stream Processing Systems

Rafael Oliveira Vasconcelos[1,2], Igor Vasconcelos[1,2], Markus Endler[1]
[1]*Department of Informatics Pontifical Catholic University of Rio de Janeiro (PUC-Rio)*
Rio de Janeiro, Brazil
email: {*rvasconcelos, ivasconcelos, endler*}*@inf.puc-rio.br*
[2]*Department of Informatics*
University Tiradentes (UNIT)
Aracaju, Brazil

Abstract— **While many data stream systems have to provide continuous (24x7) services with no acceptable downtime, they also have to cope with changes in their execution environments and in the requirements that they must comply (e.g., moving from on-premises architecture to a cloud system, changing the network technology, adding new functionality or modifying existing parts). On one hand, dynamic software reconfiguration (i.e., the capability of evolving on the fly) is a desirable feature. On the other hand, stream systems may suffer from the disruption and overhead caused by the reconfiguration. Due to the necessity of reconfiguring (i.e., evolving) the system whilst the system must not be disrupted (i.e., blocked), consistent and non-disruptive reconfiguration is still considered an open problem. This paper presents and validates D-Joseph, a non-quiescent approach for dynamic software reconfiguration that preserves the consistency of distributed data stream processing systems. Unlike many works that require the system to reach a safe state (e.g., quiescence) before performing a reconfiguration, the proposed approach enables the system to smoothly evolve (i.e., be reconfigured) in a non-disruptive way without reaching quiescence. The evaluation indicates that the proposed approach supports consistent distributed reconfiguration and has negligible impact on availability and performance. Furthermore, the implementation of the proposed approach showed better performance results in all experiments than the quiescent approach and Upstart.**

Keywords-Online Dynamic reconfiguration; Adaptability; Software adaptation; Data Stream Processing.

I. INTRODUCTION

Many stream processing systems have to provide services for 24x7, with no acceptable downtime [1]. However, they commonly have to cope with changes in their execution environment (e.g., moving from on-premises architecture to cloud architecture or changing the network technology) and in the requirements that they must comply with [2] (e.g., adding new functionality or modifying existing parts). The authors [2] further emphasize that changes are hard to predict at design time. The continuous service execution makes it difficult to fix bugs and add new required functionality on-the-fly as this requires non-disruptive replacement of parts of a software version by new ones [3]. Ertel and Felber [3] further explain that prior approaches to dynamic reconfiguration (a.k.a. dynamic adaptation, live update or dynamic evolution) require the starting of a new process and the transfer of states between the components being swapped [4]. However, some authors argue that the cost of redundant hardware may be considerable high [5].

Despite extensive research in dynamic software reconfiguration, safe reconfiguration is still an open problem [1]. A common approach is to put the component that has to be updated into a safe state, such as the quiescent state [6], before reconfiguring the system [7]. Thus, a safe reconfiguration must drive the system to a consistent state and preserve the correct completion of on-going activities [2]. At the same time, dynamic reconfiguration should also minimize the interruption of the system's service (i.e., disruption) and the delay with which the system is updated (i.e., its timeliness) [6]. Furthermore, coordinating (i.e., orchestrating) the restart of all the exchanged or added components is very challenging if the system's service must not be interrupted [3].

Aligned with the aforementioned requirements, applications in the field of data stream processing require continuous and timely processing of high-volume of data, originated from a myriad of distributed sources, to obtain online notifications from complex queries over the steady flow of data items [8]. Intelligent Transportation Systems, Network Monitoring, Stock Exchange, Smart Cities, Smart Energy management and logistics are some examples of application areas that require processing data streams. Thus, while dynamic reconfiguration is a desirable feature, such systems shall not suffer performance degradation due to the potential disruptions and overhead caused by the reconfiguration.

In order to enable dynamic software reconfiguration for stream based systems, our work allows the concurrent execution of multiple versions of a software component. Concisely, the proposed approach is based on the idea that a tuple (a.k.a. message) has to be entirely processed by a specific version of each component. However, there is no problem in updating a component *C* while a tuple *T* traverses the system as long as the system keeps the previous and the new versions of *C* (and of its dependent components) until all previous' version tuples are flushed (i.e., draining the tuples between the source and sink nodes).

The remainder of the paper is organized as follows. Section II presents an overview of the key concepts and system model used throughout this work. Section III delves into details D-Joseph. Section IV summarizes the main results of the assessment conducted to evaluate the proposal. Finally, Section V reviews and discusses the central ideas presented in this paper.

II. FUNDAMENTALS

This section presents the main concepts about data stream processing, as well as our system model and related works.

A. *Data Stream Processing*

Data stream processing is a computational paradigm [9] that is focused at sustained and timely analysis, aggregation and transformation of large volumes of data streams that are continuously updated [8]. Data stream is a continuous and online sequence of unbounded items where it is not possible to control the order of the data produced and processed [10][11]. Thus, the data is processed on-the-fly as it travels from its source nodes downstream to the consumer nodes, passing through several distributed processing nodes [12], that select, classify or manipulate the data. This model is typically represented by a graph where vertices are source nodes that produce data, operators that implement algorithms for data stream analysis, or sink nodes that consume the processed data stream, and where edges define possible data paths among the nodes (i.e., stream channels).

In order to cope with the high processing demand, stream processing systems typically employ Single Instruction, Multiple Data (SIMD) parallelism and use multiple instances of an operator (i.e., processing units), where each operator instance is responsible for processing a subset of the data stream independently of the remaining data stream, and hence without need to manage communication or synchronization among those operators [13]. Therefore, many stream processing systems are inherently distributed and may consist of dozens to hundreds of operators distributed over a large number of processing nodes [12], where each processing node executes one or several operators.

B. *System Model*

Our notion of a stream processing system is a directed acyclic graph that consists of multiple operators (i.e., components) deployed at distributed device nodes. More formally, the graph G = (V, E) consists of vertices and edges. A vertex represents an operator and an edge represents a stream channel. An edge *e* = (*v1*, *v2*) interconnects the output of vertex *v1* with the input of vertex *v2*. Vertices without input ports (i.e., without incoming edges) are referred as source vertex. Correspondingly, vertices without output ports are called sink vertices. Finally, vertices with both input and output ports are called inner vertex. A tuple *t* = (*val*, *path**) consists of a value (*val)* and an execution path (*path*)* that holds the operators, and their versions, that a tuple *t* traveled through G. For instance, a tuple *t* that traveled from source vertex SO1 to sink vertex SI1 via operators O1 and O2 holds *path* = {SO1, O1, O2}. The tuple's *val* field is transformed (i.e., processed) along the graph. A stream s = (t*) between *v1* and *v2* consists of an ordered sequence of tuples t* where t1 < t2 represents that *t1* was sent before *t2* by a node *n1*. A vertex is composed of f^{select}, f^{output} and f^{update} functions. When a vertex *v1* generates a tuple (i.e., sends it via the output port), its succeeding vertices (i.e., the vertex that receive the stream from *v1*) receives such tuple via the function f^{select}, which is in charge to select, or not, this tuple to be processed by the function f^{update}.

In order to standardize the terms and notations used throughout this work, an operator (a.k.a. graph vertex) [14] will be generically referred to as a component. A node is any physical device node (e.g., desktop and smartphone) that executes a component. A Processing Node (PN), in turn, is a node that holds at least one inner operator (i.e., an operator with input and output ports). Furthermore, as data stream systems must be elastic to adapt to variations in the volume of the data streams [15], we consider that some PNs share their workload [16].

Taking into account that many current distributed systems follow the mobile-cloud architectural paradigm [17][18], our model is composed of Client Nodes (CNs), which may be mobile or stationary nodes, and PNs deployed in the cloud. The CNs are interconnected to the cloud through a Gateway (GW), which in turns forwards the stream to the PNs. Considering that we model our system as distributed data stream system, some software components are concerned with communication issues, while other are concerned with processing issues (i.e., the analysis, aggregation and transformation the data stream). The GW, for instance, is a node in charge of forwarding the data stream from/to the CNs to/from the PNs and interconnecting the CNs to the Reconfiguration Manager (RM). Conversely, a CN has some communication component for enabling the interaction with the GW while CN may also have a processing component that performs some pre-processing on the produced data before sending the stream to the cloud.

In addition to these nodes, the RM manages software component deployments, and coordinates the execution of the reconfiguration by the nodes. The RM is responsible for coordinating (i.e., initiate and orchestrates the execution of all the operations that encompass a distributed reconfiguration) the system-wide reconfiguration process (e.g., deployment of new software components) on many CNs. For example, if the reconfiguration is the deployment of a new component version, the RM sends the code that implements the new component to the nodes and then verify whether all of them successfully deployed it. The red dashed lines represent the reconfiguration control channel between the RM and the other nodes, while the black lines represent the system data flow. Thus, all reconfigurations performed at the nodes are driven and orchestrated from the RM.

C. *Related Work*

Software reconfiguration at runtime is a research topic that combines issues and approaches from areas, such as software engineering, programming languages and operating systems. However, a common problem is the identification of states in which the system is stable and ready to evolve [2]. The authors Ertel and Felber [3] propose a framework for systems that are modeled using (data)flow-based programming (FBP) [19]. The idea behind FBP is to model the system as a directed acyclic dataflow graph where the operators (vertices) are functions that process the data flow and the edges define de input and output ports of each operator. Since the messages are delivered in order, this proposal forwards special messages informing when a component (a.k.a. operator) is safe to be reconfigured [3]. Despite the advantages, the problem with the work is that either all components will perform the reconfiguration or none of them can proceed with the reconfiguration, similar to a transaction.

The seminal work by Kramer and Magee [6] proposed and proved that the *quiescence* criterion guarantees the system consistency over the update process. Their model represents the distributed system as a directed graph whose nodes interact by means of transactions (i.e., a sequence of messages that should be atomically executed). The weakness of their work is that it causes a high disruption since it blocks all potentially dependent computation during system evolution.

III. D-JOSEPH

This section presents D-Joseph, our approach to enable dynamic reconfiguration in distributed stream processing systems. Differently from other works, D-Joseph does not need to wait for the system to reach a quiescent state (or safe state) to reconfigure a f^{update} function.

Each component has one or more f^{select}, f^{update} and f^{output} functions and components have interdependencies. The advantage of enabling a component to have more than one f^{update} function executing concurrently is that, in face of a reconfiguration, the new function is able to process part of the data stream while the old one is still in use and thus cannot be deactivated. Accordingly, when a tuple T is received by an f^{select} function, it has to choose the right f^{update} to process T. To do so, the f^{select} function verifies the *path* of T when there is more than one f^{update}, otherwise there is no need to verify the path since there is only one f^{update}. The f^{select} and f^{output} represent the input and output ports, respectively, of a component, whereas the f^{update} is the algorithm in charge of processing the transformation on the incoming data stream. Thus, we are able to reconfigure the algorithms that process the data streams (i.e., f^{update} functions) and the system's topology by means of reconfiguring the f^{select} and f^{output} functions.

A. *Management of Multiple Versions*

In the example of Figure 1, the f^{select} function of the *Processor* component has to know the version of the f^{update} applied at the *Pre-Processor* component in order to avoid inconsistency. Figure 1 shows the partial data flow of a tuple *T* when the system has the f^{update} functions A1, D1 and E1 of *Pre-Processor*, *Processor* and *Post-Processor* components, respectively. Figure 2 shows that the versions A2, D2 and E2 were added to the system and that *Processor D1* (i.e., the f^{update} function of *Processor D1*) and *Post-Processor E1* transformed the tuple *T* in order to maintain the system consistency. Thus, when *T* arrives at the f^{select} function of the *Processor* component, the f^{select} function verifies that *T* comes from *Pre-Processor A1* and then uses the *Processor D1* to transform *T*. The same happens at *Post-Processor* component. Thus, every component has to be aware of its dependency to be able to choose the right f^{update} function.

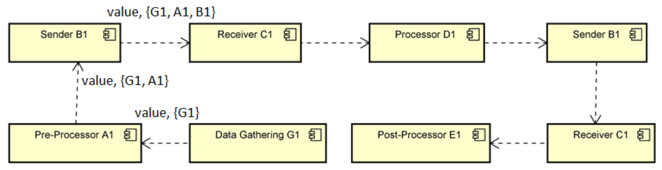

Figure 1. Teste Partial data flow of the motivating scenario where data is to be received by Receiver C1

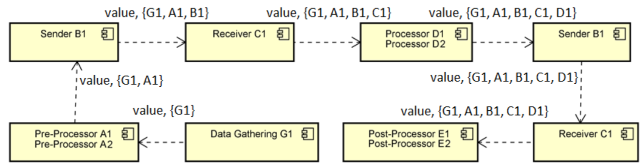

Figure 2. Execution path of the data in a partially reconfigured system

The dependencies can be managed using two approaches, static or dynamic dependency management. The former, which is the simplest one, does not take into account the "downstream" dependent components to generate the execution path of a tuple. Thus, whenever a component processes a tuple *T*, the f^{update} function's version of such component is added into the tuple's execution path, as illustrated by Figure 1 and Figure 2. Finally, when *T* arrives to a downstream component, such as the Processor component, its f^{select} function verifies the execution path of *T* to decide which is the correct f^{update} function to process *T*. To do so, each component has a list of all its upstream dependent components. Conversely, the latter approach verifies if there is any dependent component before adding the version of the f^{update} function into the execution path. If there is no dependent component, the version is not added into the execution path. Furthermore, at each component, the execution path is evaluated to check and discard the versions that have no more dependent components. In Figure 3, for instance, G1 is removed from the execution path at the Pre-Processor component since there is no dependent component of Data Gathering after Pre-Processor.

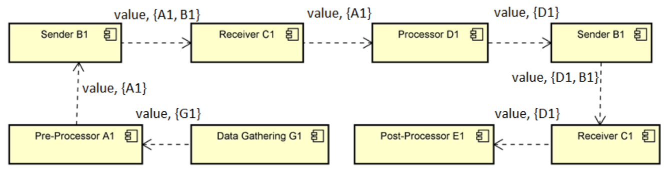

Figure 3. Execution path using the dynamic management

The advantage of applying the static dependency management is that it is simple, has a low execution cost and the dependency changing (e.g., insertion or removal of components) does not affect the system since the execution path field holds all components that a tuple traversed. Thus, a reconfiguration is performed in a simpler and faster way. However, if the execution path grows in size (i.e., there are numerous inner components between the source and the sink nodes), it may degrade the system's performance due to the network and memory costs. On the other hand, the dynamic dependency management has the advantage that does not waste network and memory since the execution path field holds only useful information, which is an advantage for huge paths. The weakness is the complexity introduced to keep the execution path field as short as possible. At each component, all downstream dependency has to be evaluated to remove the unnecessary information in the execution path field.

B. *Distributed Reconfiguration*

If one (e.g., the system administrator) needs to change the Pre-Processor and Processor component types for some reason the new Processor instances must be deployed before the new Pre-Processor instances. Thus, the reconfiguration execution of all instances has to be coordinated by the RM. Whenever the system administrator needs to replace some components, the administrator uses the RM to start the dynamic software reconfiguration. To replace the Pre-Processor and Processor component types, the RM first deploys the new version of such component on the affect nodes and then activates the instances. After that, it deactivates and removes the previous instances.

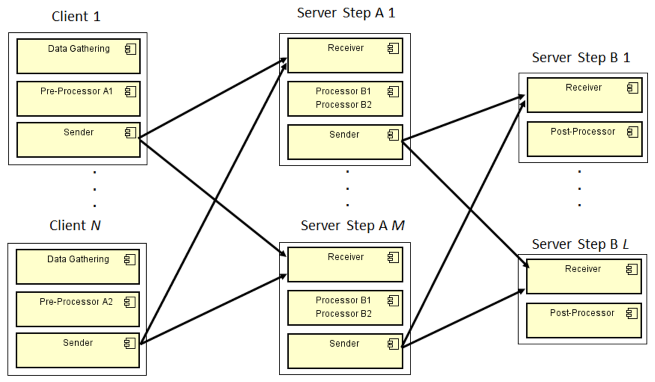

Figure 4. Partial consistent reconfiguration

Figure 4 shows that the servers must have both versions (i.e., Processor B1 and B2) while the system is partially reconfigured because some clients are not yet reconfigured. As soon as the clients are reconfigured, and there are no tuples in transit from Pre-Processor A1, the Processor B1 instances are removed from the servers at step A and the reconfiguration terminates. Therefore, our approach guarantees that the servers are able to handle data stream from any client, reconfigured or not.

IV. PERFORMANCE EVALUATION

In this section, we present the evaluation of D-Joseph. We also have measured the update time and the disruption caused by our reconfiguration approach varying the number of CNs and rate (i.e., frequency) of tuple production, as well as the overhead in terms of throughput imposed by our approach.

Our hardware test was composed of six Desktops Intel i5, 4GB DDR3 and gigabit Ethernet running Windows 7 64 bit, and a gigabit switch. We used three computers to emulate the CNs, and the other three computers to run the PNs and the RM. Our prototype application used for evaluation has been implemented using the Java programing language and Scalable Data Distribution Layer (SDDL), a middleware for scalable real-time communication [20].

Our evaluation scenario consists of a hospital that monitors patients. Each patient has a mobile equipment, composed of some sensors, that continuously monitor each patient vital signs (e.g., temperature, blood pressure, respiratory rate and systolic blood pressure). The mobile equipment sends the patient's vitals (i.e., tuple) to the hospital servers every second where the tuples must be processed as seamless data flow [21][22] in order to generate timely alerts to the medical staff. The success of such application depends on the continuous and timely monitoring of the patients [23].

In order to measure the update time and the service disruption, we varied the number of CNs from three to 300 and the system's tuple production rate from 150 tuples/s (tuples per second) to 15,000 tuples/s, using static and dynamic dependency management. The JAR file that encapsulates each deployed component has nearly 4 kilobytes (KB). The first reconfiguration performed is optimizing the system to discard the tuples that do not meet a criterion (i.e., if the patient vitals do not meet the SIRS criteria, they also will not meet the other criteria) and the second one is changing the temperature unit from Fahrenheit to Celsius.

Regarding consistency of the reconfiguration approach, all reconfigurations were performed consistently. This means that all tuples were properly *processed exactly once* by the right f^{update}. Thus, we were able to achieve global system consistency while the system is being reconfigured.

A. *Update Time*

The update time experiment measured the Round-trip Delay (RTD), which encompasses the time interval from the instant of time the RM sends the reconfiguration to the nodes until it receives an acknowledgment informing that all nodes completed the execution of the reconfiguration. In other words, it is the time from the first message sent by the RM until all components are reconfigured correctly (i.e., the system has gone from a version v1 to v2). The tuple production rate informs the production rate of the entire system, and not for each CN (i.e., the system has the same production rate in the first three scenarios of Table 1). In the case of 30 CNs and 150 tuples/s, for instance, each CN produces five tuples each second (i.e., the tuple production rate of each CN is 5 tuples/s).

As expected since our approach does not need to wait for a safe state to proceed the reconfiguration, the update time is considerably stable. It ranges from 24.07ms in the scenario with three CNs, production rate of 150 tuples/s to 26.69ms in the scenario with 300 CNs and 15,000 tuples/s, both using the static dependency management. On the other hand, with the dynamic dependency management, the update time ranges from 24.07ms to 26.48ms in the same scenarios.

TABLE 1. UPDATE TIME FOR EACH SCENARIO

# CNs	Tuple Production Rate (tuples/s)	Static Dependency Management	Dynamic Dependency Management
		Update Time (ms)	Update Time (ms)
3	150	24.29	24.07
30	150	24.18	25.20
300	150	24.88	24.75
3	1,500	25.18	25.2
30	1,500	24.60	21.36
300	1,500	25.62	23.62
3	15,000	25.05	25.63
30	15,000	26.87	26.27
300	15,000	26.69	26.48

B. *Service Disruption*

In the service disruption experiment, we measured the impact that a reconfiguration causes on the system's throughput and latency, i.e., the time interval between the tuple being sent by the source node until it is received by the sink node. In order to measure the service disruption, we assess the throughput and the latency with 300 CNs and a tuple production rate of 15,000 tuples/s. We performed two reconfigurations, at moments T1 and T2, and at each of them, we compared the throughput of the system with the throughput a second before these reconfigurations took place.

According to our experimental results, the service disruption related to the throughput was negligible. The throughput for the static dependency management had a minor increase at the reconfiguration time *T* (i.e., the moment in which the reconfiguration was performed) when compared with *T – 1* (i.e., one second before the reconfiguration), from 14,795 tuples/s to 15,019 tuples/s at reconfiguration T1 and from 14,869 tuples/s to 14,924 tuples/s at reconfiguration T2. For the dynamic dependency management, the throughput varied from 15,060 tuples/s to 15,030 tuples/s at reconfiguration T1 and from 15,073 tuples/s to 15,043 tuples/s at reconfiguration T2. In both dependency management, the throughput was not significantly affected by the reconfiguration, i.e., the experiments demonstrate that our approach causes just a marginal decrease (lower than 0.2%) in the system's throughput.

The reconfiguration may affect the latency when the system has a considerable high workload (e.g., high CPU – Central Processing Unit – usage). In both static and dynamic dependency managements, the reconfiguration T1 from *v1* to *v2*, which reduces the system's workload by discarding the tuples that do not meet some criteria, interfered the tuples' latency for a short period. However, after optimizing the system and thus reducing its workload, the reconfiguration T2 had minor impact on latency (≈ 2ms) in both cases.

C. *Overhead*

We also measured the overhead that D-Joseph imposes on the prototype application when no reconfiguration is performed. To do this experiment, we assessed the time required by the application to generate and process 100,000 tuples, as well as the throughput and latency, with and without the reconfiguration mechanism. Concerning the required time to complete the computation of all tuples, the static dependency management imposed 3.83% of overhead while the dynamic one imposed 8.98%. The throughput was reduced by 2.38% and 2.84% using the static and dynamic dependency management approaches, respectively. Finally, the latency was impacted by 6.57% and 12.50% % using the static and dynamic dependency management approaches, respectively. Thus, for such prototype application, the better choice is the static dependency management.

D. *CN Disconnection*

Due to the possibility of disconnections of mobile CNs, we assessed the amount of time required to complete a reconfiguration after an MN becomes available again. To do so, we have forced a CN to disconnect before the reconfiguration and reconnect after the reconfiguration. The reconnection time encompasses the time interval from the instant of time the CN reconnects until it completes the execution of the reconfiguration. As the number of CNs and the tuple production rate has minor impact on the update time (see Section 4.1.1), we conducted this experiment with 1,000 CNs and 1,000 tuples/s. As soon as the CN reconnects, it took 31.50ms to complete the reconfiguration.

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed and validated D-Joseph, a non-quiescent approach for dynamic reconfiguration that preserves global system consistency in distributed data stream systems. Unlike many works that require blocking the affected parts of the system to be able to proceed a reconfiguration, our proposal enables the system to smoothly evolve in a non-disruptive way.

We are aware that more work and research is still needed. However, considering the encouraging preliminary performance evaluation, we are confident that our approach can be used for development of reconfigurable data stream processing systems. In a scenario with 300 CNs and 15,000 tuples/s, our reconfiguration prototype was able to reconfigure the entire system in 24.07ms, while the service disruption in terms of throughput was lower than 0.2% due to a reconfiguration. On the other hand, the tuples' latency may increase due to a reconfiguration. When comparing the reconfigurable with the non-reconfigurable version of the application prototypes, the reconfiguration capabilities imposed an overhead of only 3.83% and 8.98% on the latency using the static and dynamic dependency approaches, respectively. Our prototype middleware reduced at most 2.84% of the system's throughput and increased at most 12.50% the system's latency when compared to the corresponding system without reconfiguration support.

Problems such as parametric variability and reconfiguration making, which is responsible for deciding when an reconfiguration is required, which alternative best satisfies the overall system goal, and which reconfigurations are needed in order to drive the system to the next state (i.e., an optimal state or state with a new functionality), are not covered by our research. Security is also an important concern for many real systems, particularly for distributed systems since nodes are potentially exposed on the Internet. Therefore, authenticity, integrity and confidentiality emerge as key aspects. Thus, ensuring that only the system administrators, or the system itself, have the ability to drive a software reconfiguration will avoid unauthorized component deployments, such as viruses, on the nodes. However, security aspects are beyond the scope of our current work.

REFERENCES

[1] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, “Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs,” in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 133–144.

[2] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, “Version-consistent dynamic reconfiguration of component-based distributed systems,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - ESEC/FSE ’11*, 2011, p. 245.

[3] S. Ertel and P. Felber, “A framework for the dynamic evolution of highly-available dataflow programs,” in *Proceedings of the 15th International Middleware Conference on - Middleware ’14*, 2014, pp. 157–168.

[4] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, “State Transfer for Clear and Efficient Runtime Updates,” in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 179–184.

[5] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, “Kitsune: Efficient, General-purpose Dynamic Software Updating for C,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 249–264.

[6] J. Kramer and J. Magee, “The evolving philosophers problem: dynamic change management,” *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.

[7] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi, “An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems,” in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering - CBSE ’12*, 2012, p. 177.

[8] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, Jun. 2012.

[9] H. Schweppe, A. Zimmermann, and D. Grill, “Flexible On-Board Stream Processing for Automotive Sensor Data,” *IEEE Trans. Ind. Informatics*, vol. 6, no. 1, pp. 81–92, Feb. 2010.

[10] L. Golab and M. T. Özsu, “Issues in Data Stream Management,” *ACM SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS ’02*, 2002, p. 1.

[12] M. Cherniack *et al.*, “Scalable distributed stream processing,” in *2003 Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, 2003, p. 12.

[13] R. O. Vasconcelos and M. Endler, “A Dynamic Load Balancing Mechanism for Data Stream Processing on DDS Systems,” M.Sc Thesis, Departamento de Informática, PUC-Rio - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2013.

[14] B. Gedik and H. Andrade, “A Model-based Framework for Building Extensible, High Performance Stream Processing Middleware and Programming Language for IBM InfoSphere Streams,” *Softw. Pr. Exper.*, vol. 42, no. 11, pp. 1363–1391, 2012.

[15] R. O. Vasconcelos, M. Endler, B. Gomes, and F. Silva, “Autonomous Load Balancing of Data Stream Processing and Mobile Communications in Scalable Data Distribution Systems,” *Int. J. Adv. Intell. Syst.*, vol. 6, no. 3&4, pp. 300–317, 2013.

[16] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, “Balancing load in stream processing with the cloud,” in *2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 16–21.

[17] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, Apr. 2010.

[18] D. J. Cook and S. K. Das, “Pervasive computing at scale: Transforming the state of the art,” *Pervasive Mob. Comput.*, vol. 8, no. 1, pp. 22–35, Feb. 2012.

[19] J. P. Morrison, *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Paramount, CA: CreateSpace, 2010.

[20] L. David, R. Vasconcelos, L. Alves, R. André, and M. Endler, “A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes,” *J. Internet Serv. Appl.*, vol. 4, no. 1, p. 16, 2013.

[21] S. I. Lee *et al.*, “Remote patient monitoring: what impact can data analytics have on cost?,” in *Proceedings of the 4th Conference on Wireless Health - WH ’13*, 2013, pp. 1–8.

[22] Forbes, “4 Interesting Tech Trends In Patient Monitoring,” 2014. [Online]. Available: http://www.forbes.com/sites/robertszczerba/2014/12/10/4-interesting-tech-trends-in-patient-monitoring. [Accessed: 14-Jul-2016].

[23] H. Catalyst, “The Year of Healthcare Data Analytics,” 2014. [Online]. Available: https://www.healthcatalyst.com/2014-Year-Healthcare-Data-Analytics. [Accessed: 14-Jul-2016].