Supporting Dynamic Reconfiguration in Distributed Data Stream Systems

Rafael Oliveira Vasconcelos[1,2], Igor Vasconcelos[1,2], Markus Endler[1], Sérgio Colcher[1]

[1]Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio) Rio de Janeiro, Brazil

[2]University Tiradentes (UNIT), Aracaju, Brazil

{rvasconcelos,ivasconcelos,endler,colcher}@inf.puc-rio.br

Abstract. *The design and development of adaptive systems brings new challenges since the dynamism of such systems is a multifaceted concern that ranges from mechanisms to enable the adaptation on the software level to the (self-) management of the entire system using adaptation policies or system administrators, for instance. Data stream processing is one example of system that requires dynamic reconfiguration. While dynamic reconfiguration is a desirable feature, stream systems may suffer with the disruption and overhead caused by the reconfiguration. In this paper, we propose and validate a non-disruptive reconfiguration approach for distributed data stream systems that support stateful components and intermittent connections. We present experimental evidence that our mechanism supports safe distributed reconfiguration and has negligible impact on availability and performance.*

1. Introduction

An adaptive system must be able to cope with changes in its execution environment and in the requirements that it must comply with [Ma et al. 2011]. The authors [Ma et al. 2011] further emphasize that changes are hard to predict at design time. On the other hand, many distributed systems have to provide services for 24x7, with no downtime allowed [Giuffrida et al. 2014]. This continuous 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 a new ones [Ertel and Felber 2014]. The authors in [Ertel and Felber 2014] 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. However, [Hayden et al. 2012] argue that the cost of redundant hardware, and the overhead to transfer and synchronize the system's state may be considerable high.

Despite extensive research in compositional dynamic software adaptation [Vasconcelos et al. 2014] [Kakousis et al. 2010] (i.e., the sort of reconfiguration addressed by our work), safe reconfiguration is still an open problem [Giuffrida et al. 2014]. A common approach is to put the component that has to be updated into a safe state, such as the quiescent state [Kramer and Magee 1990], before reconfiguring the system [Ghafari et al. 2012]. Thus, a safe reconfiguration must drive the system to a consistent state and preserve the correct completion of on-going activities [Ma et al.

2011]. At the same time, dynamic reconfiguration should also minimize the interruption of system's service (i.e. disruption) and the delay with which the system is updated (its timeliness). In [Ertel and Felber 2014], the authors also explain that orchestrating the restart of all the components is very challenging if the system's service must not be interrupted.

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 (and possibly mobile) sources, to obtain online notifications from complex queries over the steady flow of data items [Stonebraker et al. 2005] [Cugola and Margara 2012]. Thus, while dynamic reconfiguration is a desirable feature, stream systems may suffer with the disruption and overhead caused by the reconfiguration. Furthermore, at the same time, it is not feasible to block (or await a quiescent state) some of the involved components.

In order to address the aforementioned issues, we propose and validate a non-disruptive approach for dynamic reconfiguration that preserves global system consistency in distributed data stream systems. Such approach should (i) handle nodes that may appear and disappear at any time, (ii) ensure that all data items (of the data stream) are processed exactly once, and (iii) do not disrupt the system to perform a reconfiguration. More specifically, the main contributions of this work include a mechanism to enable safe reconfiguration of distributed data stream systems, a prototype middleware that implements the mechanism, and experiments that validate and demonstrate the safety of the proposed solution.

The remainder of the paper is organized as follows. Section 2 presents an overview of the key concepts and system model used throughout this work, as well as introduces a motivating scenario. Section 3 delves into details the proposed approach to dynamic reconfiguration in multiple distributed instances. Section 4 summarizes the main results of the assessment conducted. Finally, Section 5 reviews and discusses the central ideas presented in this paper, and proposes lines of future work on the subject.

2. Fundamentals

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

2.1. Related Work

A common problem in the field of software reconfiguration at runtime is the identification of states in which the system is stable and ready to evolve [Ma et al. 2011]. The authors [Ertel and Felber 2014] propose a framework for systems that are modeled using (data)flow-based programming (FBP) [Morrison 2010]. 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. When a component receives such message, it is substituted by the new version. Despite the advantages of such proposal, it neither handles nodes that may appear or disappear any time nor assumes that source components (i.e., components that

generate data) may have different versions. Therefore, the problem with the work [Ertel and Felber 2014] is that either all components will perform the reconfiguration or none of them can proceed with the reconfiguration, similar to a transaction. As demonstrated by [Nouali-Taboudjemat et al. 2010], such approach does not have a good convergence rate for systems that deal with mobile devices. Finally, while a component is updated to its new version, it is unable to process the data flow, thereby causing a system disruption (or, at least, a temporary contention).

In the subject of dynamically reconfigurable stream processing systems, the work [Schweppe et al. 2010] proposes a method for flexible on-board processing of sensor data of vehicles that is modeled as a data stream processing system. In regards to dynamic reconfiguration issues, this approach is able to change component's parameters, system's topology (i.e., the graph structure), or how the component stores its data. However, the work does not deal with consistency for topology changes. Furthermore, the proposal does not allow the modification of a component by its new version (i.e., it does not permit compositional adaptation). Finally, the system targeted by the authors is deployed on a single node (i.e., a car), conversely from ours that is distributed and has multiple instances of a component type.

While some works [Vandewoude et al. 2007] block the system to enable its evolution, others [Ma et al. 2011] [Ghafari et al. 2012] [Chen et al. 2011] are capable of executing the old and new versions concurrently. The proposals in [Ma et al. 2011] [Ghafari et al. 2012] ensure that, while a reconfiguration is performed, any extant transaction with all its sub-transactions is entirely executed in the old or in the new system's configuration (i.e. old or new versions) [Ma et al. 2011]. The major drawback is the overhead required to maintain the graph representing the system's configuration [Ghafari et al. 2012]. With the aim of solving the latter drawback, the authors in [Ghafari et al. 2012] chose to use *evolution time* (i.e., the timestamp in which a reconfiguration is performed) as a mechanism to decide if a transaction should be served by the old or new version. Thus, a transaction initiated before the evolution time is served by the old version, otherwise it is served by the new version. Although the evolution time causes minor impact in the system's performance, time synchronization in distributed systems is a well-known problem for such systems.

To the best of our knowledge, no other research work copes with the problem of adapting at run-time a component type that has multiple distributed instances spread over the distributed data stream processing system while guaranteeing the system consistency. That is, each component type may be deployed over many distributed (and possibly mobile) nodes rather than on a single node. Thus, the modification to a new version of one component type requires the coordinated modification of many component instances deployed over numerous (mobile) nodes.

2.2. Motivating Scenario

Consider a data streaming application that collects some sensor data from a smartphone, for instance, and sends it to a distributed node (e.g., cloud) in order to process the data (Figure 1). In typical stream applications in the field of mobile social applications [Ganti et al. 2011], people share sensor information about their daily physical exercise among friends and acquaintances. As an example, BikeNet [Eisenman et al. 2007] probes

location and bike route quality data, such as CO_2 level and bumpiness of the road during the ride, in order to share information about the most suitable routes [Ganti et al. 2011]. We may decompose such applications using the components shown in Figure 1.

As the system may have an arbitrary number of mobile nodes and processing hosts or servers (in the cloud), each component in Figure 1 will typically have many instances that are deployed at different nodes. As illustrated in Figure 1, the components *Data Gathering* and *Pre-Processing* run on the mobile nodes (e.g., smartphones), the *Processing* and *Post-Processing* components run in the servers of a cloud, while the components *Sender* and *Receiver* run on both mobile nodes and servers. Therefore, in order to replace a component, the reconfiguration platform has to guarantee distributed consistency. Figure 1 illustrates the component types deployed on each node and that each component type has several distributed instances spread in the distributed system. We consider that each step in the processing flow has an arbitrary number of servers that share their workload and that data sent by a client can be forwarded to any server at step A, for load balancing purposes.

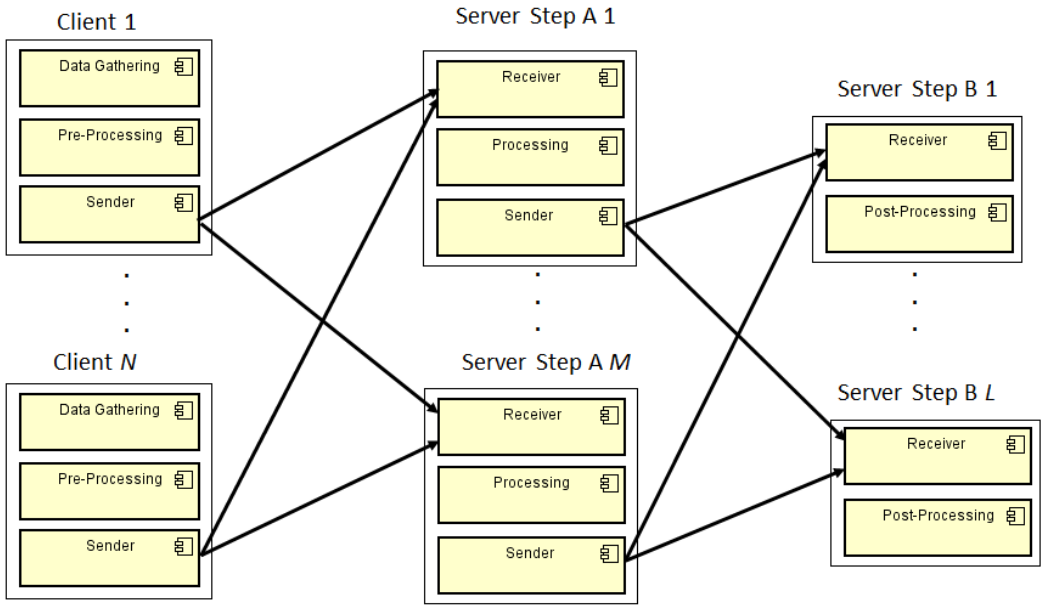

Figure 1. Deployed components and the system data flow

2.3. Data Stream Processing

Data stream processing is a computational paradigm [Schweppe et al. 2010] that is focused at sustained and timely analysis, aggregation and transformation of large volumes of data streams that are continuously updated [Cugola and Margara 2012]. A 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 [Babcock et al. 2002]. Thus, the data is processed on the fly as it travels from its source node downstream to the consumer nodes, passing through several distributed processing nodes [Cherniack et al. 2003], 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 the data paths among the nodes (i.e., stream channels).

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 [Cherniack et al. 2003], where each processing node executes one or several operators. Furthermore, such systems have an *execution dependency* among component types and each component type may have multiple instances. The execution dependency differs from deployment dependency in that it considers the runtime data and control flows, instead of the static dependencies [Ma et al. 2011] among software modules.

The work [Stonebraker et al. 2005] proposes eight general rules that hold for data stream processing; but the most important rules related to our work are: "keeping the data moving", "generating predictable outcomes", "guarantying data safety and availability", and "processing and respond instantaneously". In order to satisfy these rules, a system reconfiguration must be reliable (i.e., it does not produce erroneous outcomes and affect the system's availability), and not disrupt or block the stream. In spite of the recognized importance of dynamic reconfiguration for such systems [Babcock et al. 2002], researchers [Schweppe et al. 2010] confirm that most data stream processing middleware have no compositional adaptation mechanisms.

2.4. System Model

Our notion of a stream processing system, inspired by [Schweppe et al. 2010], 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 vertices. Correspondingly, vertices without output ports are called sink vertices. Finally, vertices with both input and output ports are called inner vertices. 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}. 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) 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 [Vasconcelos et al. 2013], we consider that some processing nodes (PNs) share their workload, as shown in Figure 1 where data originated by a client may be delivered to any server at step A.

As the main assumptions about the system and network, we assume that any node is able to enter or leave the system anytime, messages are reliably delivered in

FIFO (First In, First Out) order between two nodes. We also assume that nodes are able of correctly executing their components, and that components *per se* do not introduce any flaws that may lead the system to an inconsistent state. We consider that there are no Byzantine failures and if any node fails (fail-stop), the system is able to timely detect the node's failure.

3. Distributed Dynamic Reconfiguration

This section presents our approach to enable dynamic reconfiguration in distributed stream processing systems. The proposed approach is based on the idea that a data produced by a Client Node (CN) 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 C is multi versioned (i.e., C has a version C1 that represents the old version and C2 that represents the new version) and T is exclusively processed by C1 or C2. Differently from other works, our proposal does not need to wait for the system to reach a quiescent state (or safe state) to reconfigure a f^{update} function. In the scenario shown in Figure 1, there is a dependency between the Sender and Receiver components since they have to use a compatible algorithm in order to exchange messages (a.k.a. tuples) through the network. In this way, if the reconfiguration of the Receiver happens before any client sends a message, all messages are processed by the new version of the Receiver component since the clients use the new version of the Sender component. However, if the reconfiguration happens while the clients send messages (i.e., the data stream has a continuous flow), some messages must be processed by the new version (if and only if – iff – the message was sent by the new version of the Sender) whereas others have to be processed by the old version. At this time in which there are some clients with the old version and others with the new one, the servers must have deployed both versions of the component to be able to receive correctly the messages from any client. Therefore, both versions of the Receiver component must coexist at the servers while the system is being reconfigured.

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 correctly f^{update} to process T. To do so, the f^{select} function verifies the *path* of T. 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.

In many systems, such as data stream processing ones, the components have indirect dependencies. An indirect dependency is a mutual dependency [Ertel and Felber 2014] between components *X* and *Z* in which they are not directed interconnected to each other (i.e., there is no edge interconnecting them). In other words, they are not directly interconnected. Thus, there is at least one inner component *Y* to enable a tuple from X arrives at *Z*. In our example (Figure 1 and Figure 2), the Processing component depends on Pre-Processing component; however, there are two components between them. Due

to the possibility of multi versioned components and indirect dependencies, each tuple holds the execution path to enable our proposal to maintain the system consistency. Yet, Receiver depends on Sender and Post-Processing depends on Processing.

3.1. Dependency Management

In our example of the data stream system, the Processing's f^{select} function has to know the version of the Pre-Processing's f^{update} applied to pre-process a tuple T to avoid inconsistency. Figure 2 shows the partial data flow of a tuple T when the system has the f^{update} functions A1, D1 and E1 of Pre-Processing, Processing and Post-Processing components, respectively. Figure 3 shows that the versions A2, D2 and E2 were added to the system and that Processing D1 (i.e., the Processing's f^{update} function D1) and Post-Processing E1 processed the tuple T in order to maintain the system consistency. Thus, when T arrives at the Processing's f^{select} function (Figure 3), it verifies that T comes from Pre-Processing A1 and then uses the Processing D1 to process T. The same happens at Post-Processing. Thus, every component has to be aware of its dependency to be able to choose the right f^{update} function.

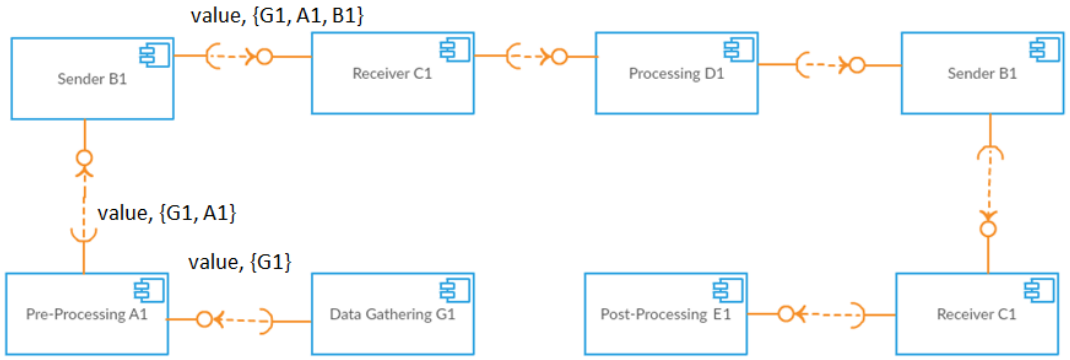

Figure 2. Partial data flow of the motivating scenario where data is to be received by Receiver C1

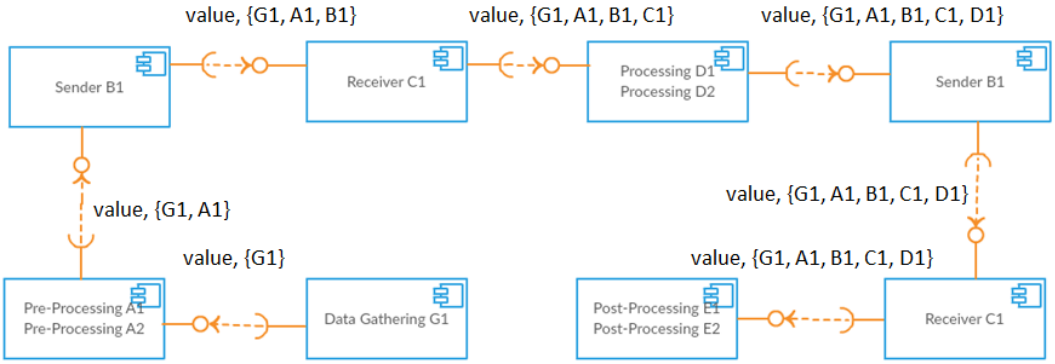

Figure 3. Tags of the data in a partial reconfigured system

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, the f^{update} function's version of such component is added into the tuple's execution path, as illustrated by Figure 2 and Figure 3. 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.

3.2. Distributed Multi Instance Reconfiguration

So far, we have not discussed how to adapt distributed multi instances of component types. As shown in Figure 1, each component type may have many instances deployed over the distributed nodes, such as the Processing component type that is deployed on all servers at step A. If one needs to change the Pre-Processing and Processing component types, the old/new version of the Processing instance can only process data originated from the old/new version of the Pre-Processing instance, as mentioned before. However, considering that the servers share the workload and our reconfiguration proposal cannot forward the data stream to a specific server, the system has to keep the old Processing instances while there is some old Pre-Processing instance to ensure that all data stream originated from the old Pre-Processing instances are processed by the old Processing instances. Furthermore, the new Processing instances must be deployed before the new Pre-Processing instances. Thus, we have to coordinate the reconfiguration execution at all nodes.

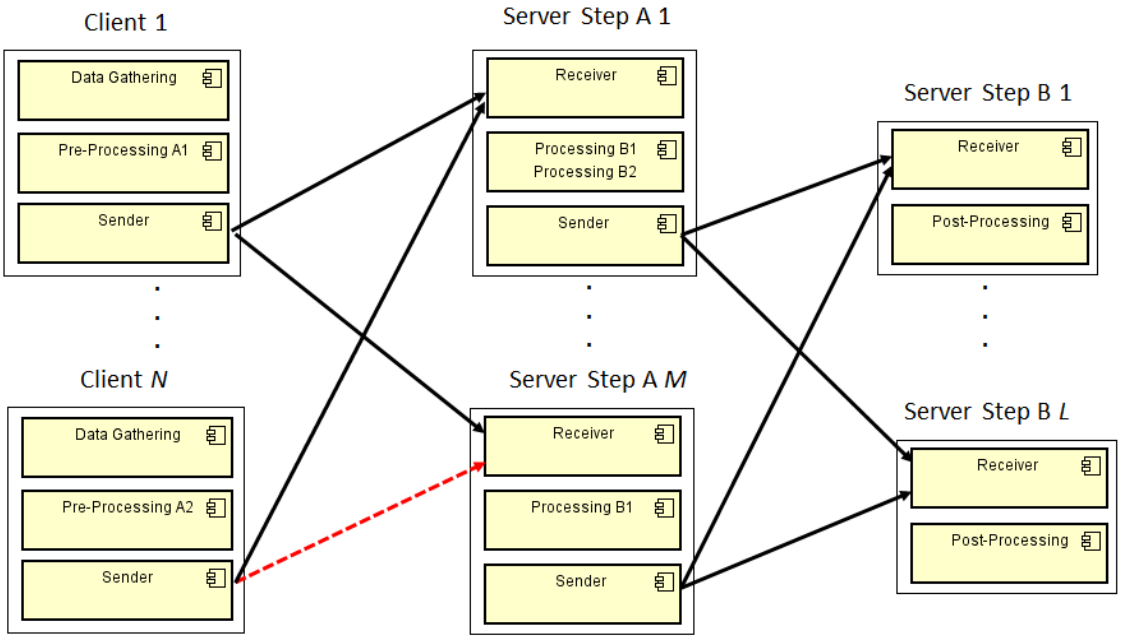

Figure 4. Partial inconsistent reconfiguration

In Figure 4, if part of the data stream from Client N goes to Server Step A M, the system achieves an inconsistent state since the server is unable to properly process the data stream. Thus, the servers must have both versions (i.e., Processing 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-Processing A1, the Processing B1 instances are removed from the servers at step A and the reconfiguration terminates, as shown in Figure 5. Therefore, our approach guarantees that the servers are able to handle data stream from any client, reconfigured or not.

In order to safely remove a component R and to guarantee that there are no tuples in transit towards R, we have borrowed ideas from the seminal papers [Lamport 1978] [Chandy and Lamport 1985] to have the notion of computation time in a decentralized manner. The idea is to add a special message (called *marker*) into the data stream to mark a specific time T in which a component is safe to be removed. Similarly to [Ertel and Felber 2014], as soon as R receives markers from all its dependent component instances, R is removed from the system. Thus, in order to remove Processing B1, whenever a client removes Pre-Processing A1, it adds a marker into its *M* stream channels to servers at step A, and whenever *N* markers (i.e., one marker from each client) arrives at a server at step A, Processing B1 is removed from such server. Then, Processing B1 is safely and gradually removed from the entire system.

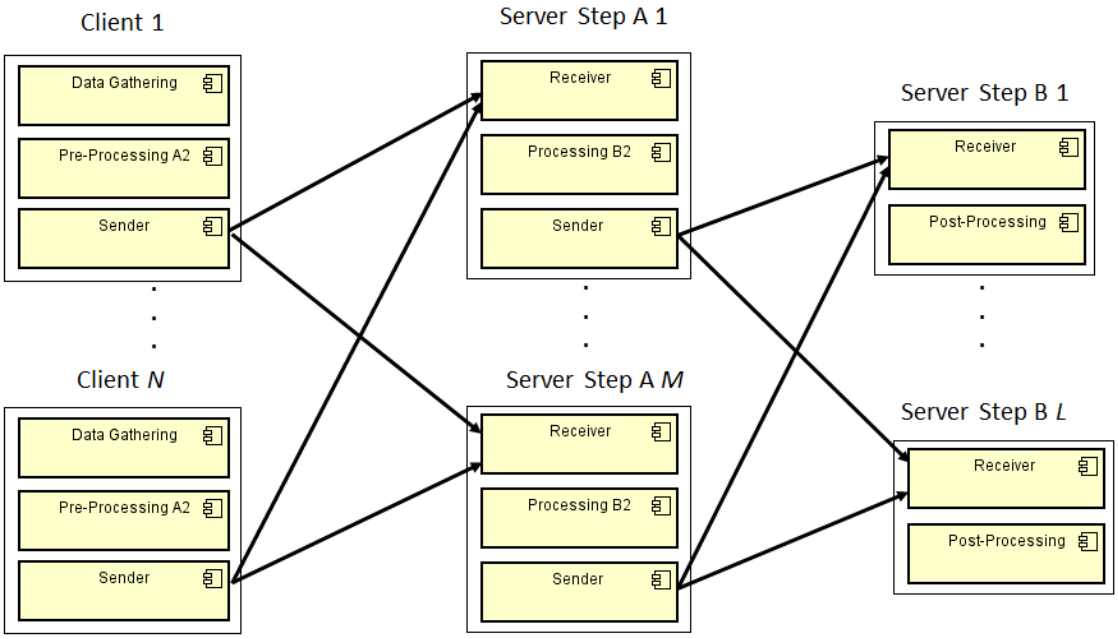

Figure 5. Reconfigured system

As in our example clients are expected to be mobile nodes using mobile networks (e.g., 3G and 4G), they may lose their connection any time. Our proposal handles nodes that may appear or disappear as follows. While the system is being reconfigured, if a client node X becomes unavailable before updating its Pre-Processing instance to version A2, the servers at step A should postpone the removal of Processing B1 until X becomes available and finishes its reconfiguration. We could adopt other strategies such as a timeout to consider that X leave permanently the system. However, the discussion about reconfiguration policies is not the focus of this work. Thus, we choose to adopt a pessimistic approach and guarantee that all tuples must be properly processed by the correct component version.

Although the system is partially reconfigured while client node X does not finish its reconfiguration, the system is able to evolve to Pre-Processing A3 and Processing B3. This is possible because each component is able to have many f^{select}, f^{update} and f^{output} functions. Thus, the Pre-Processing and Processing component types are able to have the f^{update} functions A1, A2 and A3, and B1, B2 and B3, respectively. After the second

reconfiguration from A2 and B2 to A3 and B3, the versions A2 and B2 are removed, considering that this second reconfiguration have finished.

4. Evaluation

In this section, we present the evaluation results regarding our approach. Using our prototype implementation, we have measured the time required to reconfigure the system (i.e. update time) and the disruption caused by the reconfiguration varying the number of CNs and the rate (i.e. frequency) of tuple production, as well as the overhead imposed by our approach. Finally, concerning mobile CN disconnections, we have emulated CN disconnections to verify the amount of time required to complete a reconfiguration after an MN becomes available again.

Our hardware test was composed of a Dell Laptop Intel i5-3210M 2.5GHz, 8GB DDR3 1333MHz and Wi-Fi (802.11n) interface running Windows 8.1 64 bit, a Dell Laptop Intel i5 M 480 2.66GHz, 8GB DDR3 1333MHz and Wi-Fi (802.11n) interface running Ubuntu 12.04 LTS 64 bit, and a Fast Ethernet router with 802.11n wireless connection. We used one laptop to emulate the CNs and the other one to run the PNs. Our prototype application used for evaluation has been implemented using the Java programing language and SDDL (Scalable Data Distribution Layer), a middleware for scalable real-time communication [Silva et al. 2012].

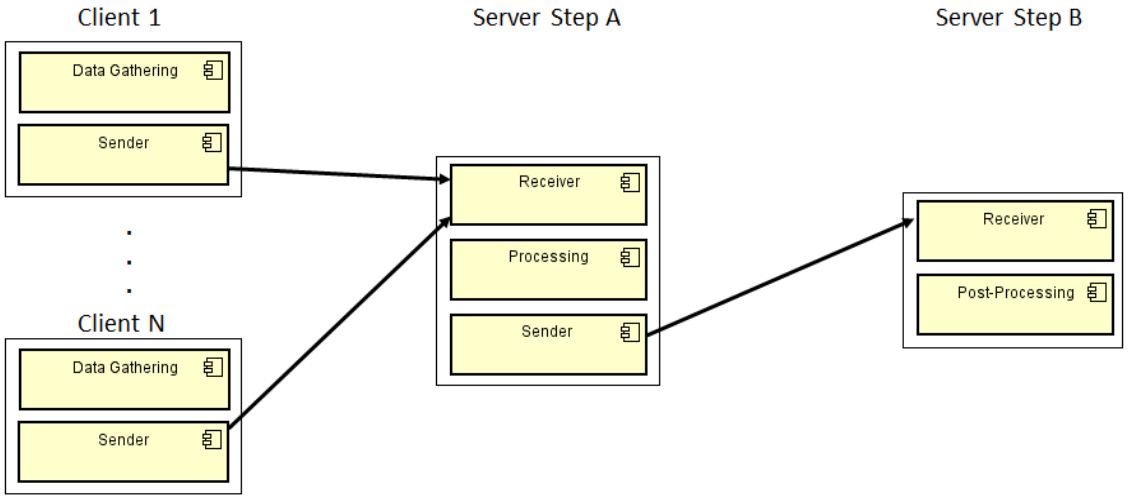

Figure 6. Deployed components and validation application data flow

The evaluation application, shown in Figure 6, is a simplification of the one presented in the motivating scenario in which there is no Pre-Processing component at the CN. The Data Gathering component generates mock data that is encoded and sent by the Sender, while the Receiver decodes and forwards the data to the Processing component. Data Gathering generates pre-defined strings that are later concatenated with other pre-defined strings at the Processing component. The Post-Processing then checks if the received string is valid or not. As we reconfigure the f^{update} functions of the Sender and Receiver to use a different encoding/decoding method, the tuple's value would be inconsistent at Post-Processing if our prototype did not properly handle such reconfiguration. Figure 6 illustrates how the components were deployed at each node

where client is a CN and server is a PN. The reconfiguration performed is changing the encoding/decoding method from ISO-8859-1 to UTF-16.

4.1. Experiments

In order to measure the update time and the service disruption, we varied the number of CNs from one to 1,000 and the system's tuple production rate from 10 tuples/s (tuples per second) to 10,000 tuples/s. We emulated each scenario 10 times, totalizing 160 emulation runs and the confidence level for all results is 95%. The JAR file that encapsulates each deployed component has 3KB (kilobytes). We also evaluated the state management overhead and the overhead that our prototype imposes while no reconfigurations are performed. All experiments were done using the static approach of the dependency management.

Regarding to reliability of the reconfiguration approach, all reconfigurations were performed consistently. This means that all tuples were *processed exactly once* and were handled by the right f^{update} function at the Receiver. Thus, we were able to achieve the global system consistency.

4.1.1 Update Time

The update time experiment measured the Round-trip Delay (RTD), which encompasses the time interval from the instant of time the reconfiguration starts until all CNs complete the execution of the reconfiguration. The tuple production rate informs the production rate of the entire system, not for each CN.

As expected since our approach does not need to await for a safe state to proceed the reconfiguration, the update time is considerably stable. It ranges from 10.05ms in the scenario with 1 CN and production rate of 10 tuples/s to 11.88ms in the scenario with 1,000 CNs and 1,000 tuples/s. The update time starts increasing with a tuple production rate of 10,000 tuples/s; however, this time increase could be affected by the high CPU (central processing unit) usage. With such production rate, both laptops experienced CPU usages of 100%. This scenario has to be revaluated to show if the update time increase is due to the production rate or due to the laptops' overload.

4.1.2 Service Disruption

In the service disruption experiment, we show the impact that a reconfiguration causes on the system's throughput (of tuples). The goal was to identify if the throughput at the end of the stream processing was affected by the reconfiguration performed. Thus, in order to measure the service disruption, we assess the system's throughput at the Post-Processing component while the tuple production rate was of 10,000 tuples/s.

According to our experiment results, the service disruption was negligible. The throughput in in the scenarios with 1 and 100 CNs had a minor increase in the reconfiguration time T when compared with $T - 1$, from 9,996 tuples/s to 10,001 tuples/s and from 10,000 tuples/s to 10,013 tuples/s, respectively. In both scenarios, the throughput was not affect by the reconfiguration. In the scenarios with 10 and 1,000 CNs, on the other hand, the reconfiguration had an insignificant influence when we compare the throughput at time T and $T - 1$. With 10 CNs, the throughput varied from 10,010 tuples/s at $T - 1$ to 10,008 tuples/s at T, which represents only 0.02% of service

disruption. In the same way, the throughput varied from 9,999 tuples/s to 9,992 tuples/s, which represents a decrease of 0.07%, in the scenario with 1,000 CNs. The experiments demonstrates that our approach causes, in some cases, a marginal decrease (lower than 0.1%) in the system's throughput.

4.1.3 Overhead

We measured the overhead that our mechanism imposes on the application prototype when no reconfiguration is performed. To do this experiment, we calculated the time required by the application to generate and process 100,000 tuples with and without the reconfiguration prototype. The overhead was 2.32%, which seems to be reasonable for the most of the applications considering the functionality provided by our proposal.

4.1.4 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.

5. Conclusion and Future Work

This paper proposes and validates a non-quiescent approach for dynamic reconfiguration that preserves system consistency in distributed data stream systems. Unlike many works that require blocking the affected parts of the system to be able the proceed a reconfiguration, our proposal enables the system to evolve in a non-disruptive way. Apart from the consistency, our proposal handles nodes that may disconnect and reconnect at any time. Hence, the main contributions of this paper are (i) a mechanism to enable safe reconfiguration of distributed data stream systems, (ii) a prototype middleware that implements the mechanism, and (iii) experiments that validate and demonstrate the safety of our proposal.

Problems such as parametric variability and reconfiguration reasoner, 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 an optimal state are not covered by our research.

We are aware that much work and research is still needed; however, considering the encouraging preliminary performance evaluation, we are confident that our approach will facilitate the development of reconfigurable stream systems. In a scenario with 1,000 CNs and 1,000 tuples/s, our prototype was able to reconfigure the entire system in 16.65ms, while imposing an overhead of 2.32%. Regarding to the system's throughput, the service disruption was lower than 0.1% when a reconfiguration was performed. Another important result is the capability of finishing the reconfiguration when a CN reconnects. In this way, our proof of concept implementation took 31.50ms to complete the reconfiguration after the CN's regress. We expect to advance our work along the

following lines: (i) implementing and evaluating the dynamic dependency management, (ii) advancing in the field of reconfiguration policies and state reconfiguration, (iii) using a more realistic and complex evaluation application to better evaluate the proof of concept prototype, and (v) supporting legacy nodes that cannot perform a reconfiguration due to some limitation.

References

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. (2002). 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*. . ACM Press. http://portal.acm.org/citation.cfm?doid=543613.543615, [accessed on Feb 11].

Chandy, K. M. and Lamport, L. (1985). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, v. 3, n. 1, p. 63–75.

Chen, H., Yu, J., Hang, C., Zang, B. and Yew, P.-C. (sep 2011). Dynamic Software Updating Using a Relaxed Consistency Model. *IEEE Transactions on Software Engineering*, v. 37, n. 5, p. 679–694.

Cherniack, M., Balakrishnan, H., Balazinska, M., et al. (2003). Scalable distributed stream processing. In *In CIDR*.

Cugola, G. and Margara, A. (1 jun 2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, v. 44, n. 3, p. 1–62.

Eisenman, S. B., Miluzzo, E., Lane, N. D., et al. (2007). The BikeNet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th international conference on Embedded networked sensor systems - SenSys '07*. . ACM Press. http://portal.acm.org/citation.cfm?doid=1322263.1322273.

Ertel, S. and Felber, P. (2014). A framework for the dynamic evolution of highly-available dataflow programs. In *Proceedings of the 15th International Middleware Conference on - Middleware '14*. . ACM Press. http://dl.acm.org/citation.cfm?doid=2663165.2663320.

Ganti, R., Ye, F. and Lei, H. (nov 2011). Mobile crowdsensing: current state and future challenges. *IEEE Communications Magazine*, v. 49, n. 11, p. 32–39.

Ghafari, M., Jamshidi, P., Shahbazi, S. and Haghighi, H. (2012). 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*. . ACM Press. http://dl.acm.org/citation.cfm?doid=2304736.2304765.

Giuffrida, C., Iorgulescu, C. and Tanenbaum, A. S. (2014). Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs. In *Proceedings of the 15th International Middleware Conference*. , Middleware '14. ACM. http://doi.acm.org/10.1145/2663165.2663328.

Hayden, C. M., Smith, E. K., Denchev, M., Hicks, M. and Foster, J. S. (2012). Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the*

ACM International Conference on Object Oriented Programming Systems Languages and Applications. , OOPSLA '12. ACM. http://doi.acm.org/10.1145/2384616.2384635.

Kakousis, K., Paspallis, N. and Papadopoulos, G. A. (nov 2010). A survey of software adaptation in mobile and ubiquitous computing. *Enterprise Information Systems*, v. 4, n. 4, p. 355–389.

Kramer, J. and Magee, J. (1990). The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, v. 16, n. 11, p. 1293–1306.

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, v. 21, n. 7, p. 558–565.

Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V. and Lu, J. (2011). 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*. . ACM Press. http://dl.acm.org/citation.cfm?doid=2025113.2025148.

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

Nouali-Taboudjemat, N., Chehbour, F. and Drias, H. (2010). On Performance Evaluation and Design of Atomic Commit Protocols for Mobile Transactions. *Distrib. Parallel Databases*, v. 27, n. 1, p. 53–94.

Schweppe, H., Zimmermann, A. and Grill, D. (feb 2010). Flexible On-Board Stream Processing for Automotive Sensor Data. *IEEE Transactions on Industrial Informatics*, v. 6, n. 1, p. 81–92.

Silva, L. D., Vasconcelos, R., Lucas Alves, R. A., et al. (2012). A Large-scale Communication Middleware for Fleet Tracking and Management. In *Salão de Ferramentas, Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2012)*. . http://sbrc2012.dcc.ufmg.br/app/pdfs/p-06/SF-3-3.pdf.

Stonebraker, M., Çetintemel, U. and Zdonik, S. (1 dec 2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, v. 34, n. 4, p. 42–47.

Vandewoude, Y., Ebraert, P., Berbers, Y. and D'Hondt, T. (dec 2007). Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, v. 33, n. 12, p. 856–868.

Vasconcelos, R. O., Endler, M., Gomes, B. and Silva, F. (2013). Autonomous Load Balancing of Data Stream Processing and Mobile Communications in Scalable Data Distribution Systems. *International Journal On Advances in Intelligent Systems (IARIA)*, v. 6, n. 3&4, p. 300–317.

Vasconcelos, R. O., Vasconcelos, I. and Endler, M. (2014). A Middleware for Managing Dynamic Software Adaptation. In *13th International Workshop on Adaptive and Reflective Middleware (ARM 2014), In conjunction with ACM/IFIP/USENIX ACM International Middleware Conference 2014*. . http://dl.acm.org/citation.cfm?id=2677022.