Towards a flexible data stream analytics platform based on the GCM autonomous software component technology

Françoise Baude, Léa El Beze, Miguel Oliva

Towards a flexible data stream analytics platform based on the GCM autonomous software component technology

Françoise Baude, Léa El Beze, Miguel Oliva
CNRS I3S UMR 7271, Université de Nice Sophia-Antipolis
2000 route des Lucioles, Bâtiment Euclide
F-06900 Sophia-Antipolis
Email: baude@unice.fr

Abstract—**Big data stream analytics platforms not only need to support performance-dictated elasticity benefiting for instance from Cloud environments. They should also support analytics that can evolve dynamically from the application viewpoint, given data nature can change so the necessary treatments on them. The benefit is that this can avoid to undeploy the current analytics, modify it off-line, redeploy the new version, and resume the analysis, missing data that arrived in the meantime. We also believe that such evolution should better be driven by autonomic behaviors whenever possible. We argue that a software component based technology, as the one we have developed so far, GCM/ProActive, can be a good fit to these needs. Using it, we present our solution, still under development, named *GCM-streaming*, which to our knowledge seems to be quite original.**

I. Introduction

Analytics of big data streams can be considered as one of the application domains of HPC. Indeed, this domain translates into in-memory and on-line (as opposed to batch, off-line) data intensive computing. Typical examples of data stream analytics consist in analyzing big data produced from social network activities, Internet of Things, etc. For decision makers, the benefit is to extract relevant knowledge, and detect trends, anomalies, or opportunities. Moreover, when a situation that requires a reaction is detected, the decision makers can react on the fly and provide alternative strategies that they foresee to be better fitted to the considered situation; once such an adaptation is performed, the data analytics can be resumed with the new scenario.

One of our long-term research goals is to provide a stream analytics middleware, capable to autonomously support such a reactive behaviour, that is: adapting the current analytics to an evolving situation, allowing it to always provide meaningful analysis. We argue in this paper that a fully flexible technology, encompassing both the programming and supporting runtime levels, is needed; flexible in terms of 1) the functional viewpoint, i.e. what the analytics aims to deliver as ”business” outcome, and 2) the non-functional (quality of service, like e.g. response time) viewpoint as a secondary and more classical objective.

In this paper we expose our arguments why an approach originating from software component based technologies can be a real asset towards this goal. Started years ago, in the context of HPC on cluster then on grid computing, now on cloud platforms, we developed a programming library named Grid Component Model (GCM for short) implemented by leveraging the ProActive parallel and distributed open source technology [1]. GCM permits to define software components, well-delimited hierarchical and autonomous software entities that feature parallelism, distribution, and also a clear support for autonomic computing thanks to a well-defined separation of the functional and the non-functional aspects a programmer has to handle.

In Section II we analyze the requirements towards this overall goal and summarize the corresponding state-of-the art solutions. Section III gives the necessary GCM background. Section IV gives a general view of our GCM-based proposition, named *GCM-streaming* for supporting a stream analytics. Section V focuses on the autonomic dynamic reconfiguration capabilities and sketches envisioned scenarios that should constitute relevant illustrations of our approach. We conclude in Section VI.

II. Requirements analysis and State of the art

Stream processing can rely on various approaches, but the one which seems the most appropriate is based upon Stream Processing Engines as opposed to database management systems and rule engines [2].

We do not want to rely on an approach that translates an abstract expression of the analytics into an internal representation that ends up being a very optimized data flow fine-grained parallel program (as in Stratosphere, Spark). Instead, we wish to allow the programmer to express the stream processing program accompanied with the adaption rules to permit this program to dynamically evolve. To ease that dual level of expressiveness, we believe it is preferable to adopt a runtime representation of the computation that is very close to the abstract view of the stream processing application. It does not prevent this abstract description to benefit from a relatively high-level language, like Spade/SPL [3] for example which then compiles into IBM's System S/InfoSphere Streams middleware format. The resulting transformation gives a dataflow graph (which could be cyclic), where each operator is a node whose respectively input/output links correspond to incoming/outgoing data streams from/to sources/sinks or from/to other operators that precedes/succeeds it in the graph. Even if being coarse-grained, it does not prevent to scale in and out each operator.

Indeed, a lot of platforms for big data stream analytic offer some flexibility and support – vertical or horizontal– elasticity in order to face the intrinsic unpredictable volume of incoming data from the input streams, but also the possible complexity of operators. The goal is to ensure a given performance level even if the volume of data to process evolves, while not overspending or under-spending resources. Given this last aspect, cloud computing is really a must [4] as resources can be acquired or released easily. However some anticipation is needed to acquire new nodes on the cloud and deploy all necessary software for them, before they are ready to be used by the analytics.

To be elastic, besides traditional lowest level resource management (including failure management), and mapping and scheduling of tasks on allocated resources, some more stream-related problems [2] have to be addressed: tolerate or be resilient against data miss or original data ordering loss induced by the introduction/removal/migration/restart of operator replica. One relevant aspect for our work pertains to how the data gets effectively routed to the right operator replica after a reconfiguration of elastic operators. This requires the platform to support channel connections reconfiguration through for instance dynamic routing tables [5], or through the reconfiguration of collective communications as multicast/unicast after a re-mapping of the static set of parallel operators slices on the available hosts [6].

Sometimes, horizontal and vertical elasticity is handled through an autonomic control approach (i.e. more or less explicitly following the typical Monitor-Analysis-Plan-Execute methodology). For instance, [7], [8] propose very well designed algorithms for auto-parallelization of operators within a parallel or even distributed setting comprised of hosts/cores. [6] relies on an elasticity enforcer and associated elastic policies. [9] handles the autonomic allocation of tasks to workers running on the targeted middleware; these tasks resulting from an original stream computing model consisting of replicated data-stream oriented workflows composed of Petri net like tasks. One original proposition is to not only sense system level performance parameters, like CPU or memory consumption, but to also install application triggers [10]. Indeed, "a signal inside the data generated by an application can serve as an earlier indicator that there will be a load change in the near future" [10]. In [10], tweets per minute, classified after the sentiment analysis operator treatment as expressing positive, neutral or negative sentiment are used. However [10] only demonstrates better resource usage than when relying on CPU usage sensors, using a simulator and not a real system. How the programmer can easily identify which are the relevant application-level produced data that should be used by the autonomic control part of the supporting platform is not addressed. Our approach will include a solution to that software engineering level question: it will clearly promote a clean separation of the core stream processing at functional level and the elements that are required for programming the meta-level responsible of the autonomic adaptation. We believe such strong and strict separation of concerns is important to ease operator programming, and operator reuse in various settings.

Towards this software engineering oriented goal and in particular concerning dynamic configuration capabilities, we are only aware of few solutions. One of them is Floe [11] where either a single operator of the analytics graph can be updated at runtime under some restrictions for easing state operator transfer, or even a sub-graph can be replaced by a new one but exhibiting a higher reconfiguration delay. The motivation is the same as ours: support always-on data flow analytics that by definition may need to change the applications logic in response to external operational needs. However, Floe does not try to offer a meta-level support for the programmer to express the conditions under which the dynamic modification of the data flow graph should occur. Consequently, their aim is not to feature a self-adaptable approach like the one suggested here; as a minimum, Floe supports some self-adaptation strategies to resource usage (as noticed above in works like [8]) only. Floe, like our solution based on GCM, adopts a component model of application execution so that a single centralized dataflow orchestrator is not a bottleneck. [12] based upon the System S IBM's stream analytics platform features the same goals as us but offers a different, not integrated, approach from what we intend to do: developers program in advance (in ORCA) an external (and centralized) orchestrator that corresponds to their applications management policy. They express which events (pertaining to either functional or non-functional aspects) are of interest and how the application should adapt upon the occurrence of these events. Application adaptation consists in dynamically and automatically connecting it with other applications from/to it features dependencies that thus may be started or canceled on-demand, which enforces application reuse. So, the adaptation logic is limited to the control of application submission and cancellation, and not to the full-fledged adaptation of a data stream graph proposed in our approach.

Overall, we aim at contributing to the improvement of industry-level software tools dedicated to stream analytics, for instance by enabling easy customization of the data-stream analytics and also capability of dynamic evolution, as summarized by Table I. None of the solutions we quoted above fulfills all these requirements as shown on Table II.

III. Background on GCM

A. Functional layer

GCM (Grid Component Model) [1] is a hierarchical component-oriented model for building parallel and distributed software. It is a framework that uses autonomous components

TABLE I
OUR REQUIREMENTS FOR A DATA STREAM ANALYTICS PLATFORM

R1	Coarse grained dataflow graph, close to the runtime view
R2	Support elasticy of dataflow graph operators
R3	Support on-the-fly replacement of used graph operators
R4	Strategies for managing data streams tuples during reconfiguration
R5	Autonomic control of runtime modification of the graph
R6	Fully distributed control of the adaptation
R7	High expressiveness for graph definition & adaptation rules

TABLE II
MAJOR FOCUS OF STUDIED SOLUTIONS W.R.T. OUR REQUIREMENTS

R1	R2	R3	R4	R5	R6	R7
[3]	[5],[6]	[11],[5]		[6],[7],[8],[9]		[3],[12],[10]

as the building block for building applications, each of which communicates and sends messages (in the form of remote method or service calls) to each other to accomplish the execution of the defined tasks. By *autonomous*, we mean that the GCM programming model enforces to define loosely coupled components that do not share memory nor control, and thus can be considered as independent entities. The core of a component is made of a ProActive multi-active object. Each call may produce a reply, in this case the reply is transparently awaited and handled by the caller through a mechanism known as a future. Pending calls to be served by a component are held in a queue managed in FIFO order by default; and can be served one by one or by multiple threads hosted by the component following the multi-active object programming paradigm: some annotations of the method signature indicate which of the methods can be safely executed concurrently. So far, the dynamic management of the number of available threads for serving method calls is not totally open to the programmer; it is a future work to allow that management at runtime and in an autonomic way (as done e.g. in [7]).

Each component works as an independent and autonomous piece of software that exposes one or many interfaces that can be used by the others, and at the same time requiring services from others. Despite this set of requirements, the components have a life-cycle of their own and can be managed to be rewired during the execution of an application which will be one of the key points we leverage in this work. Incoming method calls triggered by the components that are using the component being reconfigured are automatically made pending, i.e they are kept in the queue of the component under reconfiguration, they are not served until it is restarted.

Components can be assembled within composite components, thus forming a hierarchically organized structure. A composite can be deployed on several nodes of a distributed platform. The implementation of GCM on top of the ProActive technology made it possible to easily express in a high-level way how nodes map to physical or virtual hosts (through the concept of virtual ProActive nodes). As a meta scheduler, ProActive then is in charge to get access to such hosts, from any possibly heterogeneous combination of computing platforms be they servers, cluster or grid nodes, or VMs in private or public clouds. Composite composition can help for managing elasticity. For instance, a composite component can duplicate one of its inner component, deploy it on a new node, and add a connection towards this new component in its exposed client interface. For this the *multicast* kind of interfaces, defined in the GCM model, is particularly useful; it represents the one-to-many interfaces. As the behaviour of multicast interface can also be customized, including at runtime, it becomes possible to express the needed policy of sharing incoming service calls between the set of bound slaves inside the composite. Symmetrically the *gathercast* collective interface allows many-to-one connections; it makes it possible for a component to gather in a configurable (and a more or less strong synchronized way) the outgoing method calls triggered by the components that are bound to it.

While composite components allow the programmer to compose complex applications, primitive components are the place where the programmer expresses the functional logic. Each method of exposed interfaces is implemented by default in Java even if some native code wrapping is also possible. On the contrary, a composite component does not feature a customizable functional logic, and each of its functional methods consists in routing the incoming method call to the bound inner component interface, going through the *membrane* as can be seen in Figure 1.

B. Non-functional layer

Each component is equipped with what is named a *membrane*, which contains all the controllers that govern the non-functional behaviour of the component, thus named the *host component*.

By-default controllers permit to manage the lifecycle and the host component architecture that is its bindings, and its content in the case of a composite. Attribute controllers can be used to get and set some properties of the business code in a practical way, this provides a limited form of dynamic adaptation of the component behaviour. Besides, some of the controllers can themselves be GCM components bound together, forming an embedded GCM architecture part of the membrane. One such predefined architecture consists of a MAPE control loop, where each element of the MAPE loop translates into a component bound to the necessary others. In Figure 1, component C could be replaced by 4 bound components acting respectively as Monitor, Analysis, Planning, Execution forming a MAPE framework exposing adequate APIs (refer to [1] for more details).

Considering monitoring of components, sensing information can be obtained and relayed to the monitor component through JMX events. Predefined JMX events exist for monitoring classical resource usage, information on the host component, request service performance, etc. for instance one classical monitoring functionality is to compute the mean delay for the component to serve a request. If needed, JMX events can also be generated by the programmer application code, and the MAPE monitor component can subscribe to them. Rules to support analysis and planning of decisions can be expressed and also modified at runtime through a specific API defined for the MAPE GCM architecture. Actions handled within the execution component to reconfigure the architecture of the GCM application are either expressed by using the GCM programming API (for creating a new component, establish some bindings, etc), or by using a high-level scripting language named GCM-script.

A host component exposes non-functional client or server interfaces, by promoting the functional client or server interfaces of its controllers. These non-functional interfaces can either be internal or external: components within the membrane of the host component can be bound through non-functional interfaces to the membrane (binding to non-functional interfaces) of inner components, or to the membrane of components at the same hierarchical level as the host component (e.g. components that are connected with the host component). This is illustrated in Figure 2. All this machinery allows the programming of distributed and hierarchical autonomic strategies, as they encompass more than one single component.

Finally, we provide an interceptor mechanism to intercept each incoming or outgoing method call on a functional interface. Contrary to an application-level JMX event, the interception mechanism works only for method calls, and as in Aspect Oriented Programming (AOP) can act before or after the method call is received or sent by the component. An interceptor takes the form of a component plugged in the membrane of the host component. Such components can also be composed in an ordered chain of interceptors, like aspects that can be woven in the context of AOP. Such interceptor components can also be bound to the other component controllers present in the membrane, through their respective functional interfaces. This allows the injection of some functional level information to the membrane, that can then be handled by the adaptation layer implemented within the membrane; this can allow the membrane to steer the dynamic adaptation of the component architecture depending on information on the functional requests targeting the component. Moreover, interceptors can also be added/removed dynamically to fit the needs of a dynamic adaptation of the control logic itself.

Figure 1 summarizes these elements by showing all parts of the host composite component. A and B are inner components, C a membrane component; different types of binding involving functional and non-functional interfaces are shown.

GCM is a rich technology ready to be used to build a complex application, featuring dynamic and even autonomic reconfiguration capabilities. Such an application can itself act as a middleware, this last point of view can be compared to solutions like [13]. In the sequel, we describe such a middleware dedicated to support end-user data stream analytics applications. By relying upon the dynamic adaptation capabilities of this middleware, our goal is to deliver self-adaptable analytics.

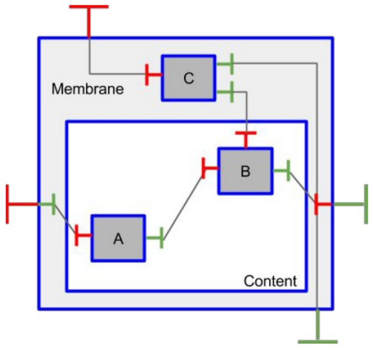

Fig. 1. Typical composite component architecture. Green color for client interfaces, red for server interfaces

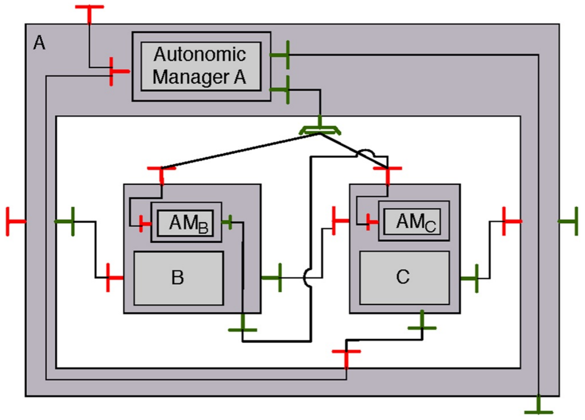

Fig. 2. Example of a more elaborated composite component architecture. Each involved component, including composite A, is managed by an autonomic manager component. Autonomic Manager of A component can act upon an external component or upon each inner component respective autonomic manager (through the broadcast interface). Manager of C can interact if needed with it, whereas manager of B can interact with manager of C

IV. *GCM-streaming* HIGH-LEVEL DESCRIPTION

GCM-streaming stands for our support to provide component-oriented autonomic stream analytics. It is to be considered as a GCM application, made of components that are either taken off-the-shelves, or can be programmed from scratch. Those components are composed and bound together to form the data analytics graph. Because it can compose existing analyses but also new components, this GCM application can be viewed as a framework. So, for easing this programming, we provide a library[1] of predefined components corresponding to typical operators, as can be found in SPADE/SPL, Storm, Floe [11] or other similar systems, that the user has simply to customize by providing the specific functional logic.

A. *Operator customization*

Type compatibility for binding interfaces of the various operator components instances requires that exposed service interfaces are of a similar Java interface type than client interfaces. The definition of this Java type is as follows: `public interface InStream { void receive(List<Tuple> tuples);}`. The re-

[1]https://github.com/moliva/gcm-streaming

ceive method of each operator type implementing `InStream` is implemented in an abstract class. Besides, depending on the operator type, or role, one abstract method is prepared. For instance, for the operator named **Operator** (see below in IV-B), one abstract method named `processTuples` is predefined. To customize the selected operator instance to his needs, the programmer simply extends this abstract class. We provide[1] some examples of such concrete extension classes (e.g., counting words by key, analyze/classify text, etc). Below is the code of the abstract class serving as a basis for the **Operator** component.

```
public abstract class BaseOperator extends
   MulticastInStreamBindingController
   implements InStream {
    protected abstract List<? extends Tuple>
        processTuples(List<Tuple> tuplesToProcess);
    public void receive(final int inputSource,
        final List<Tuple> newTuples) {
          if (inputSource > 0) {
            throw new RoutingException("this
            operator doesn’t allow an input
            source greater than 0, invalid
            input source "+ inputSource);
          }
          send(processTuples(newTuples));
          //inherited send method transmits operator
          //result (as list) to the bound components
    }
```

B. Data stream operator types

The predefined components to be customized, subsequently instantiated and bound at runtime, are as follows:

- **InTap** to read data from an external source transforming it into tuples that can be pushed to the next component through its client interface. Examples of this kind of operation could be a file or console reader, database queries, Tweet fetcher or streamers for other APIs.
- **Operator** to represent an operation. It acts as an intermediate process in a data flow graph, taking tuples from its server interface processing tuples and transforming them and forwarding the result to the next component. Typical processing steps include text normalizing, filtering (whether the tuple holds a certain property or not), reducing (as a group or sort operation), buffering, under some typical window behavior (tumbling, sliding windows).
- **OutTap** to write tuples in an external system or file, like a file writer, writing to console or a database, exposing only a server interface from tuples are received.
- Operator(s) with a **map-reduce** flavor. This is a composite component that contains by default a master and one slave. The master receives the tuples, distribute them in a balanced way to workers that apply the map function. In the first flavor the reduce operation simply aggregates map results, in this case, map workers are directly bound to an internal gathercast interface acting as server so to push results to the next operator forward. In the second (refer to Figure 3), map results are sent back as (future) return values to the master that applies the reduction to obtain new tuples pushed forward in the graph. A third flavor could embed a binary tree of reducers within the composite to implement a parallel reduction before pushing the results forward.

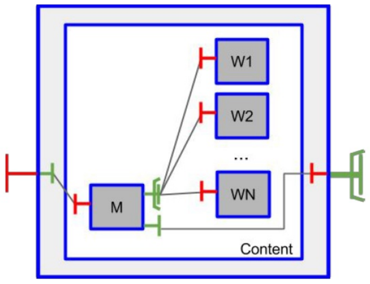

Fig. 3. A possible flavor of a map-reduce (master worker) operator

All of these operator definitions work by default with multicast client interface (see green client interface of type multicast at the very right hand side in Figures 3 or 4 for instance). This enables the application developer to build graphs of operations where the result from one of them can then be used by more than one operation in the forward direction (e.g. like from the *Text Normalizer* component in Figure 5, which duplicates output tuples to two operators). These multicast interfaces work by default in a broadcast fashion, sending the same list of tuples to all the different clients attached to it. This also enables the developer to plug in a new branch of operations during runtime if these results could be used for a new process in the business.

An operator (except InTap) exposes one single unicast server interface. It is possible that many client interfaces are bound to such a server interface. Meaning that the lists of tuples received through it are stored in the queue of the component according to their arbitrary arrival order, without being able to distinguish from which previous component they come from.

So, a last useful operator is **Aggregator**. It will receive already generated tuples from a previous operation and will forward new tuples after a certain process. The main difference with a simple operator is that in this case, the Aggregator needs to combine tuples that come from different input sources, **identifying the source origin** to come up with new results. To allow for this, the Aggregator must expose not just one but a variable number of different server interfaces from potentially different semantics for the list of tuples. In order to provide this functionality and at the same time maintain our principle of working with a single kind of interface, shared by all the operators in the system, we introduced this new component definition (refer to Figure 4): it is in fact a primitive GCM component (not a composite one) exposing as many server interfaces of the same type (`InStream`) as needed, with a different name. But the primitive component has just one such interface to implement. We put in the membrane an interceptor as a non functional Router component: it intercepts the received messages (i.e. calls of the `receive` method) before they are served by the primitive component (by the content) as they were all received through a unique server interface. To this aim, we redirect all of the server interfaces to this unique internal interface of type `InStream`. The interceptor adds an information of the source origin of each message (like source id), so that messages be then easily classified by the functional implementation of the `receive` method (e.g. aggregate one received tuples list received from source 1 with one received from source 2, to form a single aggregated new one, and again).

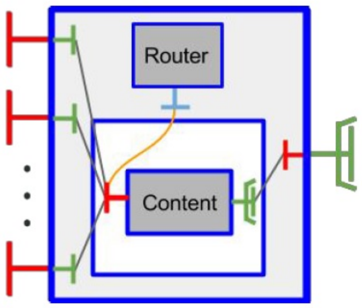

Fig. 4. A generic (w.r.t. number of sources) aggregator operator

A stream application (as sketched by figure 5), as any other GCM application, can be designed in a declarative way thanks to the use of the specific XML-based GCM Architecture Description Language. Triggering the deployment of the GCM ADL file will automatically first make sure the necessary hosts to map GCM virtual nodes used within the ADL file are ready to run GCM components; second instantiate the GCM components on the nodes, bind them, and start their respective lifecycle.

V. Towards Autonomic reconfiguration mechanisms of *GCM-streaming*

Figure 5 is a typical example of what kind of analytics graph our research targets. Notice here that the graph is flat, and each used component is a primitive component. It is however totally feasible to follow another design where some of the component operators are grouped within a composite operator. Anyway, by default, all components such as those depicted are part of a single (but distributed) composite component, even if not shown on the figure.

A. Application dynamism

The main goal is to add dynamic adaptation capabilities of such graphs, as suggested also in Floe [11]. So far, any GCM application can be dynamically controlled through the invocation of methods offered by the exposed non-functional interfaces of its components. Whatever dynamic modification is needed, the GCM API or the GCM-script language based on this API allow one to have an external application (like a console for an end-user to interact with the graph) that can safely sequence these control operations. But, this can also be alternatively embedded as an execution plan triggered by the autonomic control loop given some programmed analysis rules that monitor adequate indicators. To this aim, the MAPE API of GCM allows the dynamic addition or modification of analysis/planning rules of any MAPE equipped component. Besides, the membrane of the global composite component can be equipped with a MAPE loop in order to express a global autonomic strategy for the whole analytics if needed. Moreover, this high-level MAPE loop can interact with each MAPE loop of inner components, supporting multi-level autonomic strategies.

We can for instance change some attributes of components (through the non-functional interfaces corresponding to attribute controllers which are the only ones depicted in blue on the figure). In this sentiment analysis application, this allows us for example to dynamically switch from an English to a Spanish dictionary when analyzing the received tweets without un-deploying and redeploying the graph (see Figure 6). In this simple case, we even do not need to stop the target component *Sentiment Classifier*, to modify the attribute that indicates the selected language. Similar attribute-level modifications to window operator behaviour (through pre-defined attributes for changing e.g. the sliding or tumbling window strategies regarding the tuples that go through the window buffering operator) is also possible.

Any component can be rebound to another one. To do this, the component whose client interface needs to be connected to another server interface simply has to be stopped, rebound, and then started again. While being rebound, the incoming messages (here, incoming list of tuples) on its client interfaces are stored but not treated by the component, and will be served again when the component is started. Any new component could be added (or removed) within the composite (the composite must be temporarily stopped during the addition). Such capability is key as it allows updating any existing operator, and replacing the component by a better version. For instance, a more optimized *Text Normalizer* component could replace the current one.

All these sorts of reconfiguration actions are expressed using the GCM API, or GCM-script higher level operations, and orchestrated from either the outside or the inside (autonomic part) of components.

More drastic modification of the analytics, i.e. of its architecture itself, can be achieved as easily. It is just a matter of adding the necessary new components, and bind them accordingly within the composite graph. The necessary components can be mapped to run on existing GCM/ProActive nodes (JVMs), or can also be started on newly acquired hosts of the computing platform. Acquisition of additional hosts can be anticipated, and is delegated to the ProActive underlying middleware which is in charge of launching ProActive nodes using for instance the ProActive multi-IaaS deployment technology. Resource management for supporting GCM components is handled by an API, meaning a specific resource acquisition & sharing manager for the analytics graph can be programmed as e.g. a non-functional component plugged within the membrane of the composite component. Such a manager could act like as YARN supporting the analytics graph.

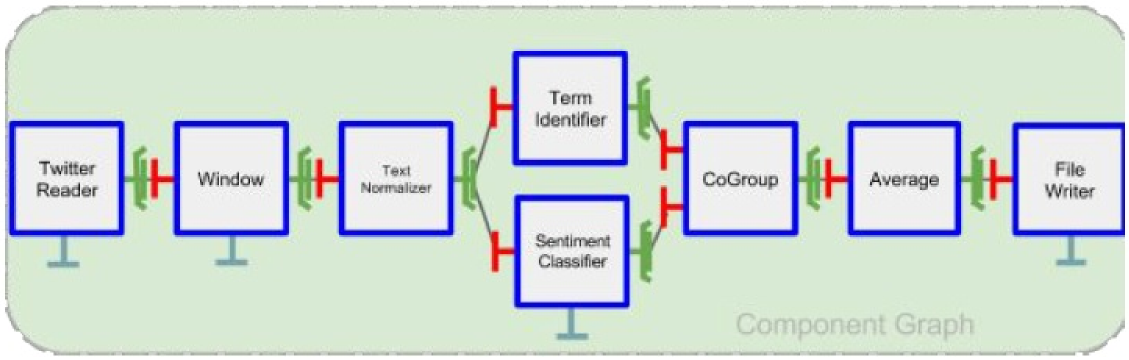

Fig. 5. A sentiment analysis data stream GCM application. Reconfiguration is limited to the usage of the exposed attribute controllers (blue interfaces)

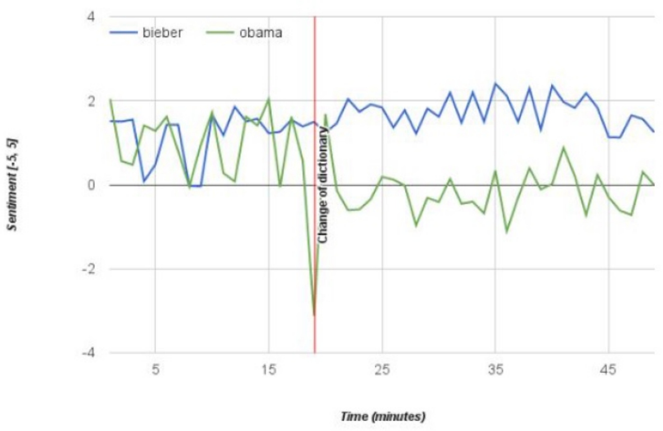

Fig. 6. Execution of the sentiment analysis application with dynamic adaptation of the dictionary used to rank the sentiment (negative -5 to positive +5) in tweets pertaining to popular terms (`Bieber`, `Obama`, a personality which seems less appreciated by Spanish speakers compared to English speakers).

Whatever modification is enacted to the graph, there is a clear need to ensure some correctness properties regarding for instance the awaiting-to-be-processed tuples [8]. Indeed, removing a component implies its awaiting tuples are stored in the stopped component. In the near future, we intend to devise adequate reconfiguration protocols in order to express what has to be done with such tuples. A reasonable strategy could be to extract all pending tuples from the queue of the component that is stopped to be removed, transfer them in front of the queue of the new one, before this new one gets bound, that is before it receives tuples from its predecessor(s). An alternative strategy could allow the two components to run in parallel until the old one becomes useless and can be garbage-collected. Likewise, tuples disordering should be addressed by adequate (sequencer or timestamp-based) protocols if necessary. This can happen for instance if several workers of a Map-Reduce operator are processing tuples in parallel in a non-synchronized fashion, but the reduction should apply on processed tuples conforming to their initial ordering. Disordering can also happen as tuples are consumed within an operator component using its native multi-thread service mechanism. Such protocols to reconfigure GCM applications can even be proved correct and sound using adequate formal approaches, like was done in the past [14].

B. Targeted scenarios requiring autonomic functional adaptation

The ”Internet of things” domain features typical situations that must analyze pervasive sensors and instruments data for extended periods. Analysis must be always on, if for instance they control a smart power grid system, a flood detection system, or any early warning system for natural disaster, or in situations like urgent computing or crisis management. Dynamism is needed as data nature, not only data volume, may evolve in time, requiring not only well-studied elasticity to meet deadlines but also application changes: add new sensed data filters or cleaners, modify a machine learning component used for predictive analytics, etc. We are in particular interested by detecting specific situations (a.k.a. situation aware computing through CARE –Classification, Assessment, Resolution, and Enactment– loops [15]) ; in particular detect *somehow ahead-of-time* what the new situation will be, and consequently what the analytics should become to fit to the new situation. Obviously, the new deployed analytics must also be equipped with situation detection autonomic capabilities, for it to further anticipate new situations in the future.

In practice, we aim at having Analysis GCM MAPE components fed with sensed information to detect which is now the situation among a predefined list of anticipated situations. Once the new situation gets identified, the framework will Plan and Execute the dynamic deployment and activation of a new sub-graph of the current analytics that will take over the obsolete sub-graph.

The richness of the GCM architecture description language (ADL) permits to define in a declarative way a possibly complex GCM architecture, with its functional and non-functional levels, whose deployment can be autonomously triggered. This ADL file can be considered as a deployment plan that must be enacted by an autonomic loop once new situation detected and decision adaptation taken. The way the end user will be able to define such adaptation plan without having any deep expertise in GCM and distributed systems is still to be defined. To realize this, we need: JMX events or functional interceptors to be automatically woven to the functional logic, Event-Condition-Action rules or machine learning algorithms as analysis/planning components of the loop in order to take reaction decision, or better to anticipatory act (like suggested by [16]). All would better be expressed by some specifically designed Domain Specific Languages we may need now to work on.

Another functional adaptation need pertains to "what-if" hypothesis testing. Dynamically, a data scientist may want to manually or automatically add to the current running analysis graph some new branches in order to explore alternate hypothesis; without needing to undeploy then redeploy the running core analysis that must be always running.

VI. Conclusion

In this paper, we have explained why we believe the GCM technology can be a perfect fit to support data stream autonomous analytics.

Indeed, GCM features (1) native dynamic reconfiguration capabilities and hierarchical definition, that can be handled by the runtime GCM API and a declarative architecture description language, (2) clear still inter-operable functional and non-functional levels that can host predefined or user-designed control loop components. Consequently, this greatly improves the expressiveness compared to the existing approaches.

From the performance viewpoint, particularly important for handling big data, it incorporates both distributed and multi-core native support as it relies upon the multi-active parallel and distributed active object technology, and the ProActive platform for dynamic host acquisition. Thus, operators of the analytics can run efficiently. However, we lack yet some resource management autonomic adaptation strategies to be able to adapt the number of threads at multi-active object scheduling level, or to adapt the number of hosts allocated to the analytics graph given some constraints for optimizing the cost of hired cloud resources, or to share available resources the best among the GCM components. Reconfiguration of the GCM-streaming analytics needs to be performant, however, it will always be less time consuming to apply dynamic reconfiguration of the already deployed analytics, compared to having to destroy it, and redeploy a new one from scratch.

In the near future, our priority is to work at the adaptation logic expressiveness and to better gear it towards the end-users. In particular, this adaptation logic has to be translated into the corresponding GCM non-functional aspects. Then we plan to illustrate our approach in well chosen scenarios like the ones sketched above that require online application dynamism, both in the composition and in performance.

Acknowledgment

The authors would like to thank the Labex *UCN@Sophia* ucnlab.eu for financial support of this work.

References

[1] F. Baude, L. Henrio, and C. Ruz, "Programming distributed and adaptable autonomous components - the GCM/ProActive framework," *Softw., Pract. Exper.*, vol. 45, no. 9, pp. 1189–1227, 2015.

[2] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[3] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "Ibm streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[4] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 5, pp. 333–345, 2013.

[5] T. Chen, Z. Man, H. Li, X. Sun, R. K. Wong, and Z. Yu, "Building a massive stream computing platform for flexible applications," in *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 2014, pp. 414–421.

[6] R. Barazzutti, T. Heinze, A. Martin, E. Onica, P. Felber, C. Fetzer, Z. Jerzak, M. Pasin, and E. Riviere, "Elastic scaling of a high-throughput content-based publish/subscribe engine," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 567–576.

[7] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[8] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1447–1463, 2014.

[9] R. Tolosana-Calasanz, J. Diaz-Montes, O. Rana, and M. Parashar, "Extending cometcloud to process dynamic data streams on heterogeneous infrastructures," in *Cloud and Autonomic Computing (ICCAC), 2014 International Conference on*. IEEE, 2014, pp. 196–205.

[10] A. A. D. Souza and M. A. Netto, "Using application data for sla-aware auto-scaling in cloud environments," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*. IEEE, 2015, pp. 252–255.

[11] Y. Simmhan and A. Kumbhare, "Floe: A continuous dataflow framework for dynamic cloud applications," *arXiv preprint arXiv:1406.5977*, 2014.

[12] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar, "Building user-defined runtime adaptation routines for stream processing applications," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1826–1837, 2012.

[13] B. Surajbali, P. Grace, and G. Coulson, "Ao-opencom: An ao-middleware architecture supporting flexible dynamic reconfiguration," in *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*. ACM, 2014, pp. 75–84.

[14] L. Henrio and M. Rivera, "Stopping safely hierarchical distributed components: application to gcm," in *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*. ACM, 2008, p. 8.

[15] E. S. Chan, D. Gawlick, A. Ghoneimy, and Z. H. Liu, "Situation aware computing for big data," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1–6.

[16] I. D. P. Anaya, V. Simko, J. Bourcier, N. Plouzeau, and J.-M. Jézéquel, "A prediction-driven adaptation approach for self-adaptive sensor networks," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2014, pp. 145–154.

[17] N. Mohamed and J. Al-Jaroodi, "Real-time big data analytics: Applications and challenges." in *HPCS*, 2014, pp. 305–310.

[18] A. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 57.

[19] M. Le and M. Li, "Towards true elasticity of spark," 2014. [Online]. Available: http://www.slideshare.net/SparkSummit/08-le-li