Vasconcelos et al. Journal of Internet Services and Applications (2016) 7:8
DOI 10.1186/s13174-016-0050-z

Journal of Internet Services and Applications

RESEARCH Open Access

Dynamic and coordinated software reconfiguration in distributed data stream systems

Rafael Oliveira Vasconcelos[1,2*], Igor Vasconcelos[1,2] and Markus Endler[1]

Abstract

While many systems have to provide 24 × 7 services with no acceptable downtime, they have to be able to cope with changes in their execution environment and in the requirements that they must comply, in which data stream processing is one example of system that has to evolve during its execution. On one hand, dynamic reconfiguration (i.e., the capability of evolving on-the-fly) is a desirable feature. On the other hand, stream systems may suffer with the disruption and overhead caused by the reconfiguration. Due to these conflicting requirements, safe and non-disruptive reconfiguration is still an open problem. 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.

Keywords: Dynamic reconfiguration, Adaptability, Software adaptation, Mobile communication, Reflective middleware, Data Stream Processing

1 Introduction

Many stream processing systems have to provide services for 24x7, with no acceptable downtime [1, 2]. 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 [3] (e.g., adding new functionality or modifying existing parts). The authors [3] 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 [4, 5]. Ertel and Felber [4] 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 [6]. However, the authors in [7] argue that the cost of redundant hardware may be considerable high.

Despite extensive research in dynamic software reconfiguration [8–11], safe reconfiguration is still an open problem [2, 12–14]. A common approach is to put the component that has to be updated into a safe state, such as the quiescent state [15], before reconfiguring the system [16]. Thus, a safe reconfiguration must drive the system to a consistent state and preserve the correct completion of on-going activities [3]. 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) [15, 17]. In [4] the authors also explain that 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.

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 [18–20].

* Correspondence: rvasconcelos@inf.puc-rio.br
[1]Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rua Marques de São Vicente 225, Gávea, Office 503, Rio de Janeiro-RJ, Brazil
[2]University Tiradentes (UNIT), Av. Murilo Dantas, 300, Farolândia, Bloco A, Aracaju-SE, Brazil

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).

1.1 Problem statement

Stream processing systems are composed by distributed components, in which each component process a portion of the data stream. 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 [3] among software modules. Stream processing systems need to achieve a safe state and to maintain the consistency among the states of the components during and after the dynamic reconfiguration. At the same time, due to the continuous nature of the stream it is not feasible to block (or await a quiescent state of) some of the involved components. Another issue is that this sort of system has to handle coordinated adaptations of several – possibly distributed – instances of a same component type. The example by [16] in illustrates a simple scenario, such as a chat application, in which senders interact with the receivers through the execution of a message payload compression and de-compression algorithm, respectively. In a typical distributed system, both sender and receiver nodes have their own local instances of the (de)compression component type. Thus, in a system that has N users (or nodes), each component type has N instances (i.e., *node i* has one instance of each of the following component types: Sender, Receiver, Compression and Decompression).

Whenever one swaps the Compression and Decompression component types for new versions that are incompatible with the previous one, all distributed instances of the (De)Compression component types should be updated to the new versions in a coordinated manner to guarantee global system's consistency (i.e., "…a state in which the system can continue processing normally rather than progressing towards an error state…" [15]). In other words, a middleware has to ensure that none of the messages compressed by the old component will be decompressed by the new version (or vice-versa), in order to avoid putting the system into an inconsistent state, as discussed in [16]. Thus, the main challenge here is that there are N instances of both Compression and Decompression component types deployed in a distributed manner and the middleware has to manage the dynamic dependencies [3] between all the component instances, guarantying the local/global consistency while updating the components. Local consistency considers only the consistency of a single node, whereas global consistency comprises the local states of all distributed components and all their messages in transit [3]. If it is not possible to block all the N Senders and the middleware does not handle all messages in transit sent by the old version before changing all the N (De)Compression instances, some nodes might use the new Decompression component to decompress a message compressed using the previous Compression version (or vice-versa). Thus, the system has to coordinate the replacement of the (De)Compression instances in all the N nodes. Furthermore, a node that gets temporally disconnected/unavailable during the system's adaptation process may delay the evolution of the rest of the system since the middleware cannot conclude the software evolution.

While the concept of *adaptation transaction* [15, 21] is enough and suitable to ensure the consistency when considering a single component instance, it may not be sufficient for multiple and distributed instances of component types. For example, in the case of Fig. 1, a transaction should start at T0 and finish only at T4. Although the system is safe to be updated before time T1, if at T1 the Sink Node receives a reconfiguration message to replace the (De)Compression components but the same message is delivered to the Source Node after T1, the reconfiguration will drive the system to an inconsistent sate since source and sink nodes will interact using different (De)Compression algorithms. Analyzing only the Sink Node, its local Decompression instance is in safe state until T2, time in which the Decompression is called. One can apply the transaction concept to handle multiple instances of a component type; but a transaction imposes a considerable overhead because a sender would have to start a transaction with each other receiver node, for every message transmission. This entails two problems: (i) for each message exchange, one has to initiate a transaction that includes message exchanges with all the receivers, but this becomes unfeasible in a scenario with many (e.g., thousands) of receivers, and (ii) the middleware (reconfiguration platform) should be aware of the receivers of each message and initiate a transaction with every receiver regardless of the fact that a reconfiguration be required at this time.

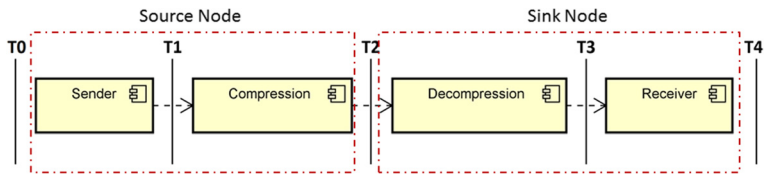

Fig. 1 Message delivery system scenario

1.2 Motivating scenario

As a motivating scenario, consider a data streaming application that collects some sensor data from smartphones, for instance, and sends them to the cloud in order to process the data. Typical stream analysis occurs in back-office servers of mobile social applications [22] where people share sensor information about their daily physical exercise among friends and acquaintances. As an example, BikeNet [23] 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 [22]. Likewise, Google has been granted a patent to detect uneven road surfaces to plot the smoothest routes [24]. We may decompose such streaming applications using the components shown in Fig. 2.

As the system may have an arbitrary number of mobile nodes and processing hosts or servers (typically deployed in the cloud), each component in Fig. 2 will typically have many instances that are deployed at different nodes. As illustrated in Fig. 3, the components *Data Gathering* and *Pre-Processor* run on the mobile nodes (e.g., smartphones), the *Processor* and *Post-Processor* 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 the system consistency. Fig. 3 illustrates the component types deployed on each node and that each component type has several distributed instances spread over 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 such as in [18].

Whenever one changes the sender's compression algorithm to a new version, the receiver's decompression should also be replaced to a version that is compatible with the new compression algorithm. This modification has to be made in such a way that none of the messages will be compressed and decompressed using different algorithms in order to guarantee the system consistency.

1.3 Objective and contributions

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

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. 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 to evaluate the proposal. 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 the main concepts about data stream processing, as well as our system model and related works.

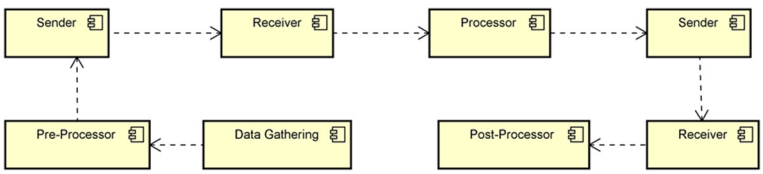

Fig. 2 Illustrating scenario in which data is gathered and pre-processed before being sent to a distributed node, which is in charge of processing the data

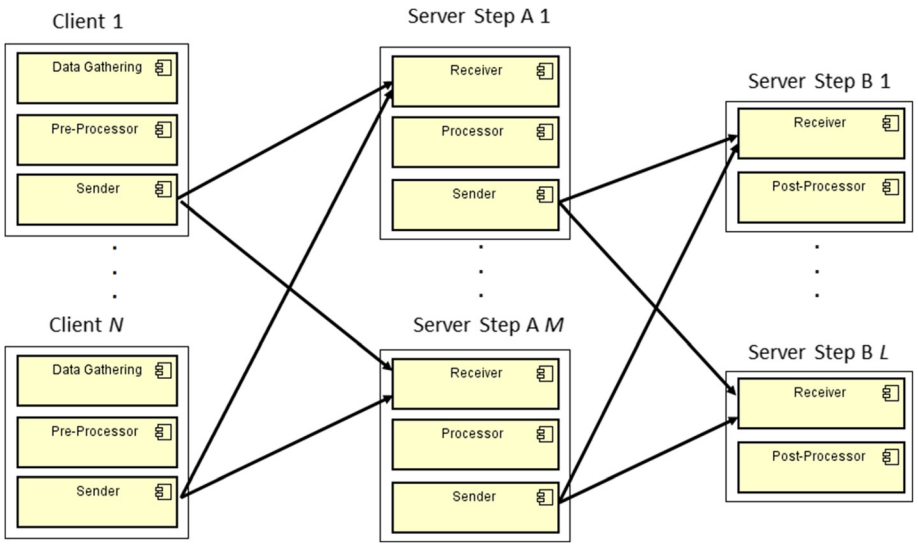

Fig. 3 Deployed components and the system data flow

2.1 Data stream processing

Data stream processing is a computational paradigm [25] that is focused at sustained and timely analysis, aggregation and transformation of large volumes of data streams that are continuously updated [20]. 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 [26, 27]. 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 [28], 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 SIMD (Single Instruction, Multiple Data) 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 [29]. 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 [28, 30], where each processing node executes one or several operators.

The work [19] 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., avoid erroneous outcomes and impacting on the system's availability), and not disrupt or block the stream. In spite of the recognized importance of dynamic reconfiguration for such systems [27], researchers [25] confirm that most data stream processing middleware have no dynamic software reconfiguration mechanisms. The works [4, 31] are examples of stream processing middlewares that support dynamic reconfiguration.

2.2 System model

Our notion of a stream processing system, inspired by [25], 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, and an internal state in case of a stateful vertex. 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) [32] 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 [33, 34], we consider that some processing nodes (PNs) share their workload [35, 36], as shown in Fig. 3 where data originated by a client may be delivered to any server at step A.

Taking into account that many current distributed systems follow the mobile-cloud architectural paradigm [37, 38], our model (Fig. 4) is composed of client nodes (CNs), which may be mobile or stationary nodes, and processing nodes (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. Currently, many of the CNs are expected to be mobile devices, which introduces new problems such as the high rate of connections/disconnection (e.g., a mobile node – MN – may become (un)available at any time). 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 Reconfiguration Manager manages software component deployments, and coordinates the execution of the reconfiguration by the nodes. The Reconfiguration Manager 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 Reconfiguration Manager 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 Reconfiguration Manager 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 Reconfiguration Manager.

As the main assumptions about the system and network, we consider that a client node is able to enter or leave the system any time, messages are reliably delivered in FIFO (First In, First Out) order between two nodes, nodes execute their components correctly, and that all components *per se* do not introduce any flaws that may lead the system to an inconsistent state. We also assume that there are no Byzantine failures and if any node fails (fail-stop), the system is able to timely detect the node's failure [39]. Although a node may leave the system due to a fail-stop or disconnection, the system itself remains operating 24x7.

2.3 Related work

Software reconfiguration at runtime is a research topic that combines issues and approaches from areas such as software engineering [13, 40], programming languages [41, 42] and operating systems [1, 43]. However, a

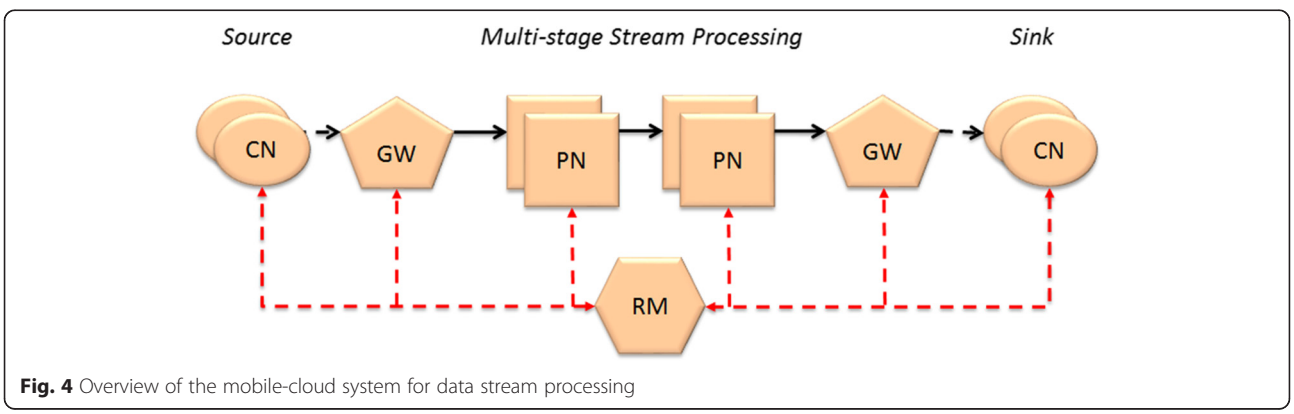

Fig. 4 Overview of the mobile-cloud system for data stream processing

common problem is the identification of states in which the system is stable and ready to evolve [3]. The authors [4] propose a framework for systems that are modeled using (data)flow-based programming (FBP) [44]. 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 [4] 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 in [4] 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 [45], such approach does not have a good convergence rate (i.e., the reconfiguration fails in most cases) in scenarios that have a considerable amount of failures or disconnections, such as 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 [25] proposes a method for flexible vehicle on-board processing of sensor data that is modeled as a data stream processing system. In regards to dynamic reconfiguration issues, this approach is able to change the component's parameters, system's topology (i.e., the graph structure), or the way components store their data. Whenever a component's parameter is updated, their work uses a state transfer mechanism to perform an update from the old component state to the new one. However, the proposal does not allow the modification of a component by its new version (i.e., it does not permit compositional adaptation [10, 46]). Finally, the systems considered by the authors are always deployed on a single node (i.e., a car), unlike ours where the system is distributed and has multiple instances of a component type.

The seminal work by Kramer and Magee [15] 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. *Tranquillity* [21] is a weaker alternative to the quiescence criterion. The idea behind *tranquility* is that the reconfiguration may proceed even if there is an ongoing transaction as long as the component to be reconfigured is not involved in such transaction. The authors in [3] argue that tranquility would permit unsafe updates if a sub-transaction was initiated by another sub-transaction that is not directed connected (i.e., has no direct dependency) to the component that started the root transaction. Another drawback of both quiescence and tranquillity is that they block at least part of the system in order to replace the target components to their new versions, which in turn may cause a significant system disruption [3, 16].

While the works [15] and [21] block the system to enable its evolution, others [3, 16, 40] are capable of executing the old and new versions concurrently. The proposals in [3, 16] ensure that, while a reconfiguration is performed, any extant transaction with all its subtransactions is entirely executed in the old or in the new system's configuration (i.e., old or new versions) [3]. The work [3] manages the dependencies between the components by means of a directed graph where vertices represent versioned components and edges represent the dependencies. The major drawbacks are the overhead required to maintain the graph representing the system's configuration [16, 47]. With the aim of solve the former drawback, the authors in [16] 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. The authors [39] explain that clocks can easily drift and accumulate significant errors, and that physical clock poses serious problems to applications that depend on a synchronized notion of time.

The work by [48] proposes a *cooperative update* in which the component to be replaced is notified about the reconfiguration and cooperates with the reconfiguration platform in order to proceed to a consistent state before the reconfiguration takes place. None of the works [40] and [48] discuss the problems related to reconfiguration of distributed nodes since in dynamic patching the problem is of how to generate a patch at runtime and how to procced some local update of the software.

To the best of our knowledge, no other research work copes with the problem of adapting at run-time, in a non-disruptive way, a component type that has a dynamic set of distributed (and possibly mobile) instances spread over the distributed data stream processing system while guaranteeing the global system consistency.

That is, each component type may be deployed over many distributed (and possibly mobile) nodes rather than on a single node, and the adaption process does not disrupt the system. Thus, the modification to a new version of one component type requires the coordinated modification of many component instances deployed over numerous (mobile) nodes that may get temporally disconnected/unavailable.

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 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 (a.k.a. message) 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 the same version (i.e., the old or the new version). Differently from other works, such as [43], 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 Figs. 2 and 3, 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 (i.e., source node) 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 components to be able to receive correctly the messages from any client. Therefore, both versions of the Receiver component 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 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. As our proposal deal with stateful and stateless components, a component is also able to hold its internal state.

3.1 Direct and indirect dependencies

In many systems, such as data stream processing ones, the components have indirect dependencies. An indirect dependency is a mutual dependency [4] between components X and Z in which they are not directed interconnected to each other (i.e., there is no edge of the graph interconnecting them). Thus, there is at least one inner component Y to enable a tuple from X to arrive at Z. In Fig. 2, the Processor component depends on the Pre-Processor 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 the f^{select} function to choose the correct component version, thus maintaining the system consistency. In Fig. 2, for instance, Receiver depends on Sender, and Post-Processor depends on Processor, which in turn depends on Pre-Processor, and Pre-Processor depends on Data Gathering.

In their work, the researchers [4] show an example (Fig. 5) in which the load and reply components have an indirect dependency. The authors explain that to change the load component from a blocking I/O to a non-blocking I/O (NIO) version, it is required to change also the reply component to a NIO version due to the indirect dependency between the reply and load component types. As tuples might be in transit in such path, the system has to ensure that all tuples are processed by the proper version.

3.2 Dependency management

In our example of the data stream system, 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 6 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 7 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

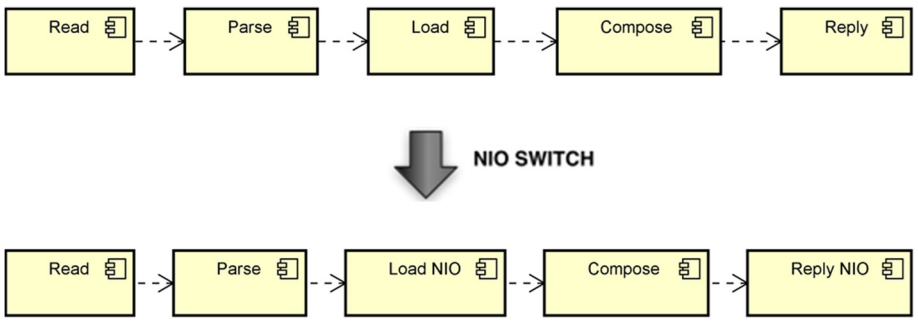

Fig. 5 Example of an indirect dependency

T in order to maintain the system consistency. Thus, when T arrives at the f^{select} function of the *Processor* component (Fig. 7), 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.

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 Figs. 6 and 7. 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 Fig. 8, 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.

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 and the system consistency whenever the dependency changes. At each component, all downstream dependency has to be evaluated to remove the unnecessary information in the execution path field. Furthermore, if a reconfiguration inserts a new dependency, many upstream components must be notified and the in transit tuples have to be handled by the system. In Fig. 8, if the system administrator adds a Processor D2 component that depends on Data Gathering, the downstream dependencies at Receive, Sender and Pre-Processor have to be updated, and the in transit tuples, which do not have the information about G1 in their execution path field, have to be processed by the old version of Processor (i.e., Processor D1).

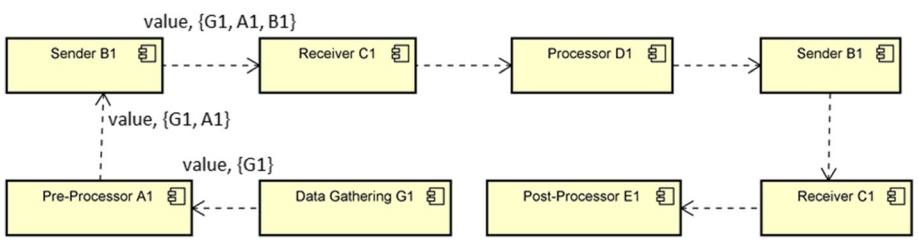

Fig. 6 Partial data flow of the motivating scenario where data is to be received by Receiver C1

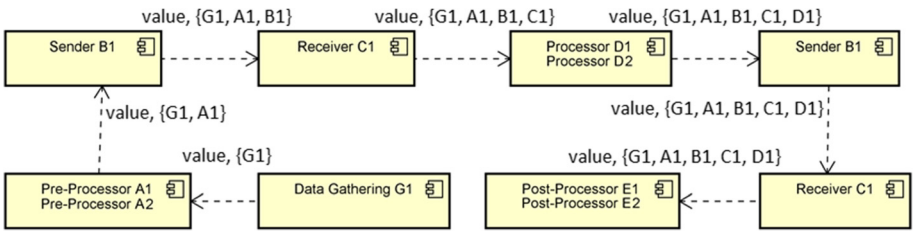

Fig. 7 Execution path of the data in a partially reconfigured system

3.3 State management

Whenever a f^{update} function C is reconfigured (i.e., updated), the system has to update the instances C_N to C_{N+1} (i.e., the old and new versions, respectively). For stateful f^{update} functions, the system has to transfer the state from C_N to C_{N+1}. However, the coexistence of C_N and C_{N+1} requires a synchronization mechanism to ensure that the concurrent execution of C_N and C_{N+1} does not drive the system to an inconsistent state (or the system may choose does not permit concurrent execution of a partially reconfigured function). In order to avoid duplicated states between C_N and C_{N+1} (i.e., each version holding its own state), and simplify the synchronization mechanism, the old and new versions share the same state (see Fig. 9) during the system reconfiguration. The advantage of sharing a global state for C_N and C_{N+1} is that there is no need to apply state transfer and that synchronization mechanisms between C_N and C_{N+1} are simplified due to no state duplication. Hence, we are able to update from C_N to C_{N+1} in a non-disruptive way (i.e., the data stream keeps flowing without any contention).

Although such characteristic, a state reconfiguration requires transferring the state from the old version to the new one in a synchronized way. Hereupon, we choose to update the state not allowing the coexistence of both states since we do not need to create specialized synchronization functions to map from/to the old state to/from the new state, such as in [40]. If a state S_N has the interface I_M, a state S_{N+1} that replaces S_N must have I_M to maintain the compatibility with the already deployed components. However, S_{N+1} may also have an interface I_{M+1} that is expected to serve new components. Similarly to Rubah [49], the state reconfiguration is performed in parallel to reduce the system's disruption. However, we do not support lazy state reconfiguration for two reason: (i) we consider that the state is not significantly large since it is kept in memory, and (ii) as a continuous stream processing system, we consider that there is not enough time between two successive calls of a component to justify the use of a lazy reconfiguration approach.

3.4 Distributed multi instance reconfiguration

So far, we have not discussed about how to adapt distributed multiple instances of component types. As shown in Fig. 3, each component type may have many instances deployed over the distributed nodes, such as the Processor component type that is deployed on all servers at step A. If one (e.g., the system administrator) needs to change the Pre-Processor and Processor component types for some reason, the old/new version of the Processor instance can only process data originated from the old/new version of the Pre-Processor instance, as mentioned before. However, considering that the data stream can be forwarded to any server due to the load balancing, the system has to keep the old Processor instances up and running while there is some old Pre-Processor instance, so as to ensure that all data stream originated from the old Pre-Processor instances is still processed by the old Processor instances. Furthermore, 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 Reconfiguration Manager. Whenever the system administrator needs to replace some components, the

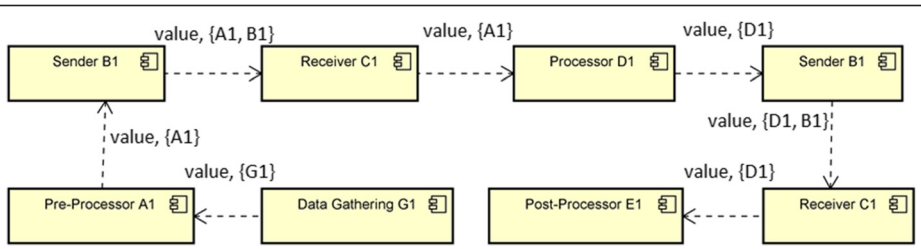

Fig. 8 Execution path using the dynamic management

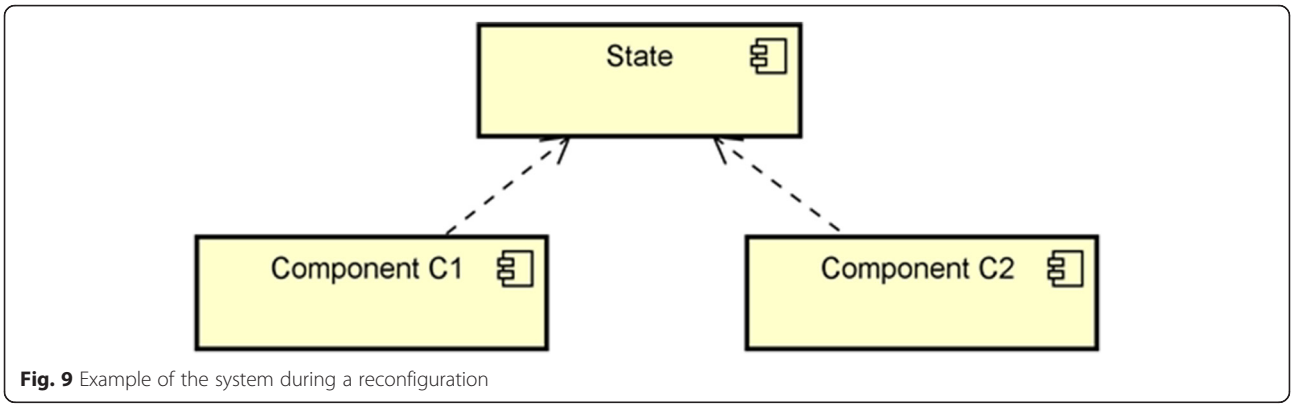

Fig. 9 Example of the system during a reconfiguration

administrator uses the Reconfiguration Manager to start the dynamic software reconfiguration. To replace the Pre-Processor and Processor component types, the Reconfiguration Manager 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.

In Fig. 10, 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, Fig. 11 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, as shown in Fig. 12. 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 [50, 51] in order to know when a component R is safe to be removed. 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 [4], as soon as R receives markers from all its

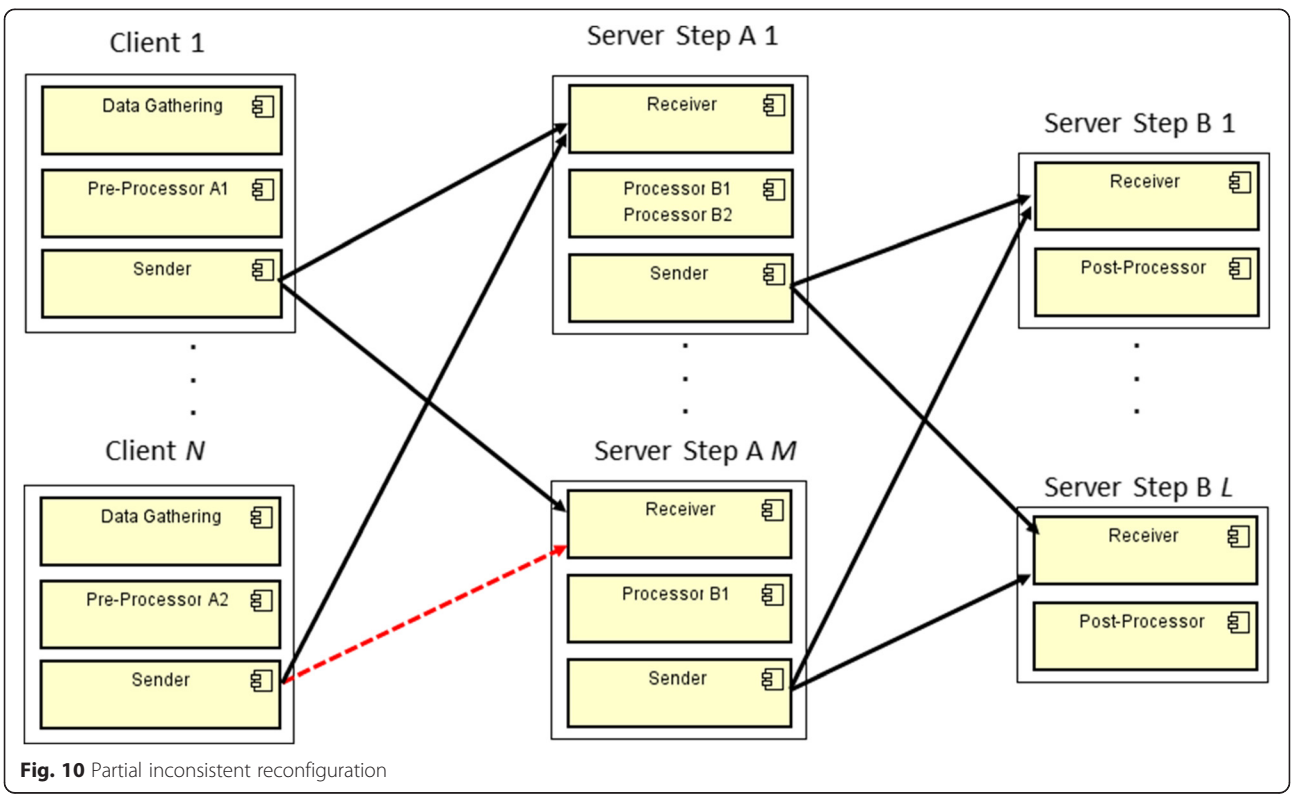

Fig. 10 Partial inconsistent reconfiguration

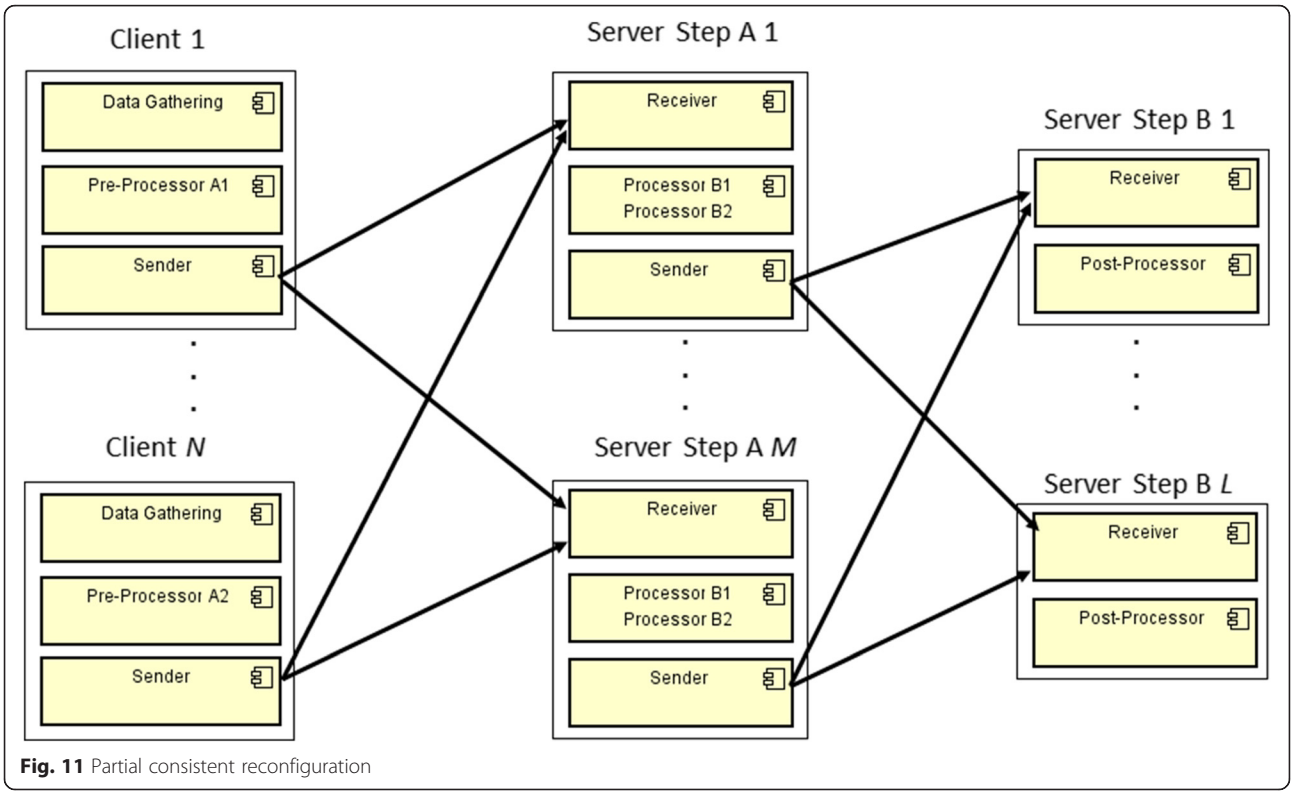

Fig. 11 Partial consistent reconfiguration

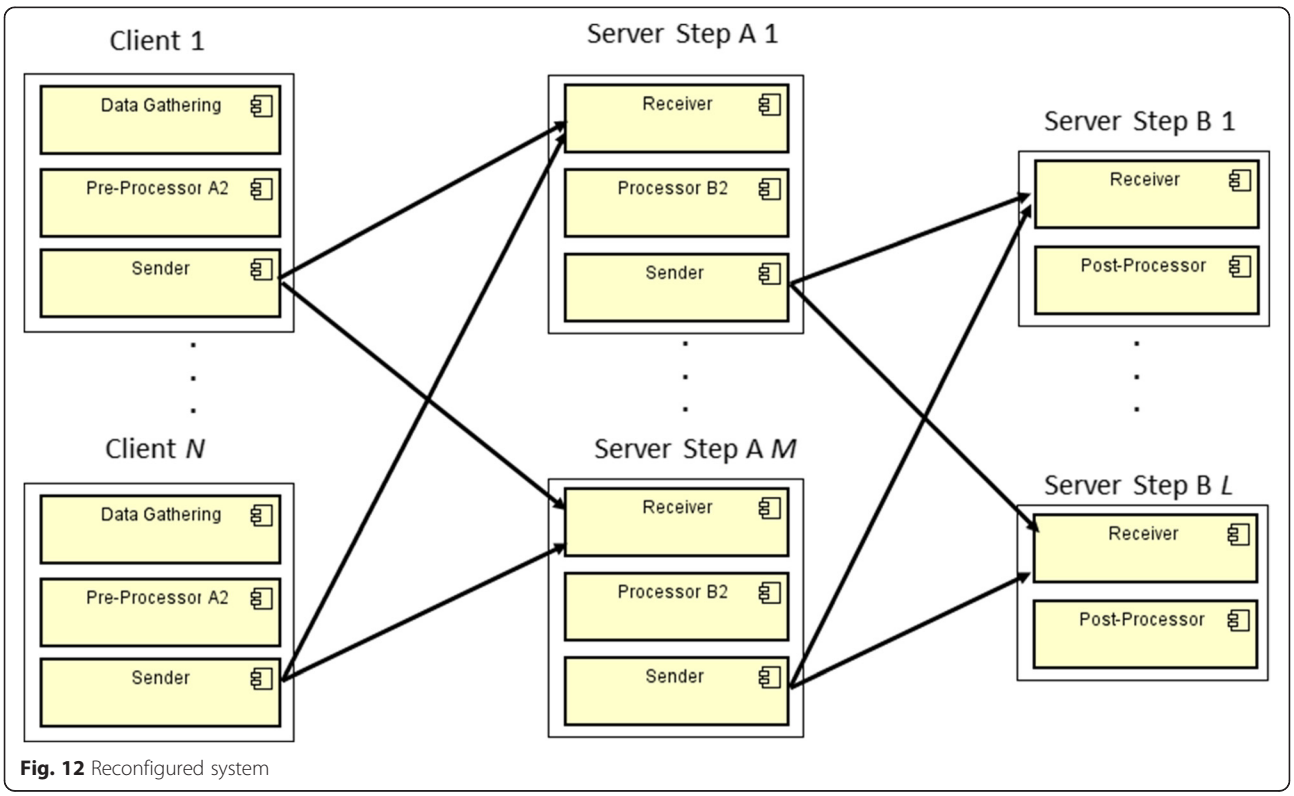

Fig. 12 Reconfigured system

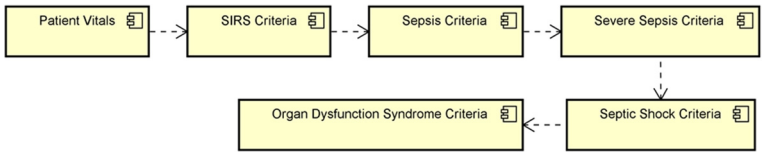

Fig. 13 Prototype application's component model using the outdated criteria to identify septic shock

dependent component instances, R is removed from the system. Thus, in order to remove Processor B1, whenever a client removes Pre-Processor A1 due to such request from the Reconfiguration Manager, the client 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, Processor B1 is removed from such server. Then, Processor B1 is removed safely and gradually from the entire system. For such, we assume that each component knows its upstream dependencies. Each marker carries the information about the component version that was removed from a node. In addition, the Reconfiguration Manager informs to each node the amount of markers that it has to receive before removing its component. For instance, whenever Pre-Processor A1 is to be removed, the Reconfiguration Manager informs the servers that there are *N* clients (i.e., each server has to receive *N* markers from clients, notifying that Pre-Processor A1 is no more on the system, to remove the Processor B1 component safely).

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, such as in Fig. 11, if a client node X becomes unavailable before updating its Pre-Processor instance to version A2, the servers at step A should postpone the removal of Processor 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 [52] is not the focus of this work.

Although the system is partially reconfigured while Client Node X does not finish its reconfiguration, the system is able to evolve to Pre-Processor A3 and Processor B3. This is possible because each component is able to have many f^{select}, f^{update} and f^{output} functions. Thus, the Pre-Processor and Processor 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.

3.5 Prototype implementation

As a proof of concept, we have implemented our proposal using the Java programming language for Java and Android platforms. Even though Android uses the Java programming language, it executes upon the Android runtime (ART) virtual machine, while Java uses HotSpot virtual machine for desktop applications. Android also does not provide the full Java API (Application Programming Interface) and provides some specific APIs for handling the *classloader*, for instance.

Some reconfiguration capabilities are implemented using Java/Android reflection in order to enable us to add dynamically new JARs (Java Archives) into the application's *classloader* and to instantiate components. Most of our implementation uses software engineering designs (e.g., interfaces and abstract classes) and programming techniques (e.g., Java generic types) in order to reduce the use of computational reflection, since it introduces a considerable overhead at runtime [53, 54].

4 Evaluation

In this section, we present the evaluation results regarding our approach. Using our prototype implementation, we have compared the time required to reconfigure the system (i.e., update time) and disruption of our approach against those of the quiescence based approach. On an second experiment, we also have measured the update

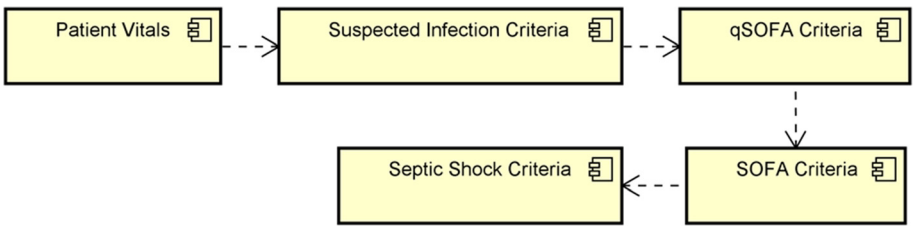

Fig. 14 Prototype application's component model using the latest criteria to identify septic shock

Table 1 Parameters of the evaluation scenarios

# CNs	Tuple production rate (tuples/s)
3	150, 1,500 and 15,000
30	150, 1,500 and 15,000
300	150, 1,500 and 15,000

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. Finally, concerning to mobile disconnections, we have emulated disconnections to verify the amount of time required to complete a reconfiguration after an MN becomes available again.

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 Reconfiguration Manager. 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 [55, 56].

4.1 Prototype application

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 [57, 58] 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 [59].

The prototype application, shown in Fig. 13, defines the severity of sepsis and septic shock using the criteria provided by [60]. This application verifies the patient vitals to identify if the patient is having a septic shock. The Patient Vitals component generates the data (i.e., patient vitals) that is processed by the hospital's system in the cloud. The remaining components verify if the patient's vitals meet the criteria that indicate a septic shock. The only component that runs at the CN is the Patient Vitals, all other components run at the PNs. However, since February 23, 2016, these criteria are no longer recommended for the diagnosis of septic shock [60]. The new recommended criteria to identify septic shock [61], shown in Fig. 14, uses the qSOFA (quick Sequential [Sepsis-related] Organ Failure Assessment) and SOFA (Sequential [Sepsis-related] Organ Failure Assessment) scores to determine if a patient is having a septic shock. Thus, this is an example where the entire monitoring system must be updated to use the latest criteria to identify such serious medical condition.

4.2 Performance experiments

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. We emulated each scenario 5 times and the confidence level for all results is 95 %. The JAR file that encapsulates each deployed component has nearly 4 KB (kilobytes). The first reconfiguration performed is optimizing the system in Fig. 13 to discard the tuples that do not meet a criteria (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. Although such reconfigurations may not be carried out in practice, they were applied only for the purpose of performance evaluation.

Table 1 shows the configuration parameters that we applied to evaluate the approach. We also evaluated the overhead that our prototype imposes while no

Table 2 Update time for each evaluated scenario

# CNs	Tuple Production Rate (tuples/s)	Static Dependency Management		Dynamic Dependency Management	
		Update Time (ms)	Confidence Interval (ms)	Update Time (ms)	Confidence Interval (ms)
3	150	24.29	+/−7.61	24.07	+/−5.98
30	150	24.18	+/−6.34	25.20	+/−4.45
300	150	24.88	+/−3.22	24.75	+/−3.74
3	1,500	25.18	+/−3.78	25.2	+/−4.43
30	1,500	24.60	+/−5.74	21.36	+/−3.74
300	1,500	25.62	+/−3.63	23.62	+/−2.72
3	15,000	25.05	+/−4.60	25.63	+/−4.61
30	15,000	26.87	+/−5.99	26.27	+/−4.32
300	15,000	26.69	+/−6.27	26.48	+/−5.68

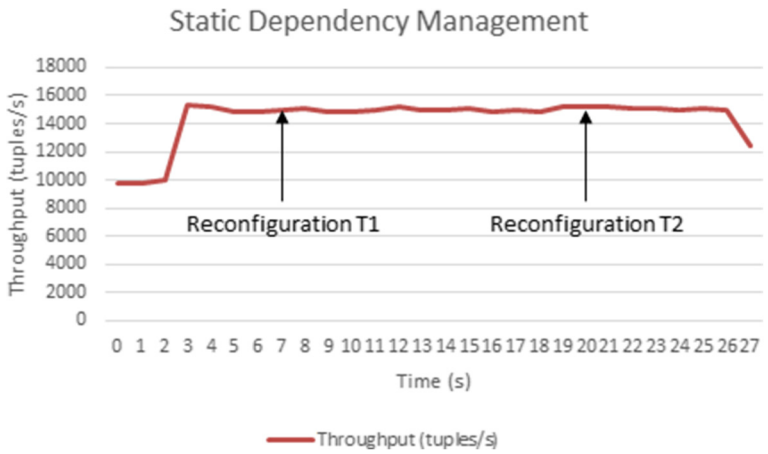

Fig. 15 Throughput in the scenario with 300 CNs, 15,000 tuples/s and static dependency management

reconfiguration is performed. All experiments were performed using the static approach of the dependency management.

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.

4.2.1 Update time

The update time experiment measured the Round-trip Delay (RTD), which encompasses the time interval from the instant of time the Reconfiguration Manager 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 Reconfiguration Manager 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 2). 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 24.07 ms in the scenario with three CNs, production rate of 150 tuples/s to 26.69 ms 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 24.07 ms to 26.48 ms in the same scenarios.

4.2.2 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

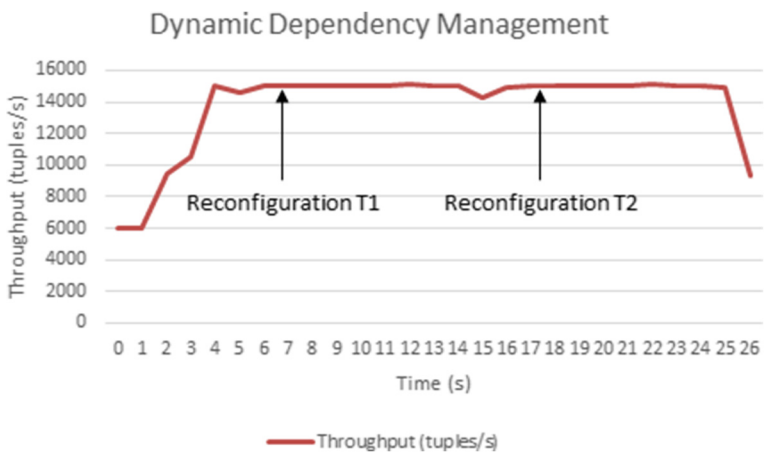

Fig. 16 Throughput in the scenario with 300 CNs, 15,000 tuples/s and dynamic dependency management

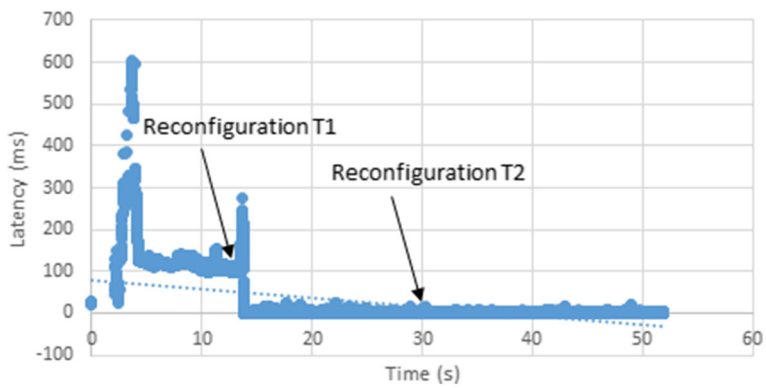

Fig. 17 Latency in the scenario with 300 CNs, 15,000 tuples/s and static dependency management

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 (see Figs. 15 and 16), the service disruption related to the throughput was negligible. The throughput for the static dependency management (in Fig. 15) 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 (Fig. 16), 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.

According to the experiments (see Figs. 17 and 18), 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 (≈2 ms) in both cases.

4.2.3 Overhead

We also measured the overhead that our mechanism imposes on the prototype application, described in Section 4.1,

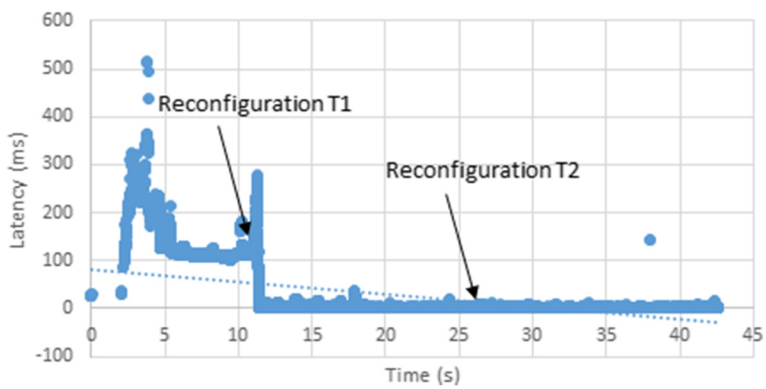

Fig. 18 Latency in the scenario with 300 CNs, 15,000 tuples/s and dynamic dependency management

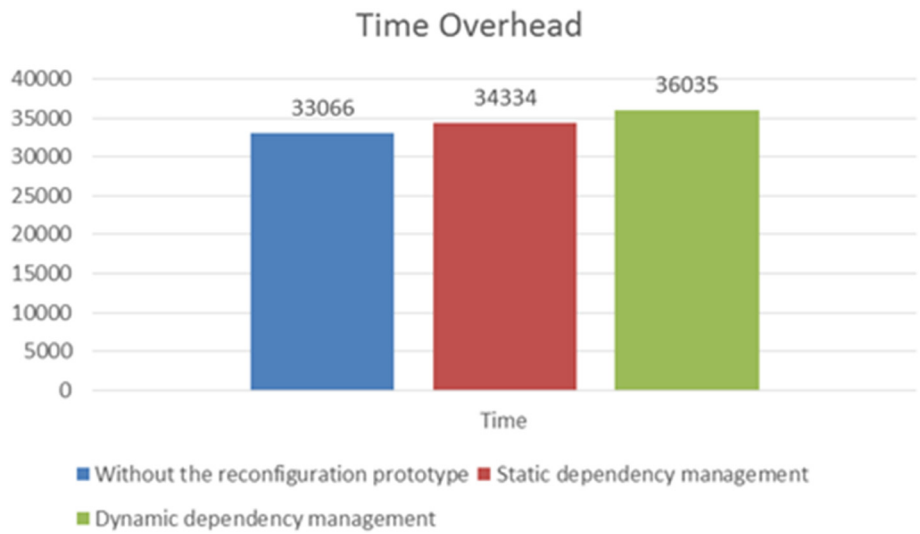

Fig. 19 Overhead in terms of time imposed by the reconfiguration prototype

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 (see Fig. 19), the static dependency management imposed 3.83 % of overhead while the dynamic one imposed 8.98 %. The throughput (see Fig. 20) was reduced by 2.38 and 2.84 % using the static and dynamic dependency management approaches, respectively. Finally, the latency (see Fig. 21) 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.

4.2.4 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 the Reconfiguration Manager receives an acknowledgment informing that the CN completed the execution of the reconfiguration. As the number of CNs and the tuple

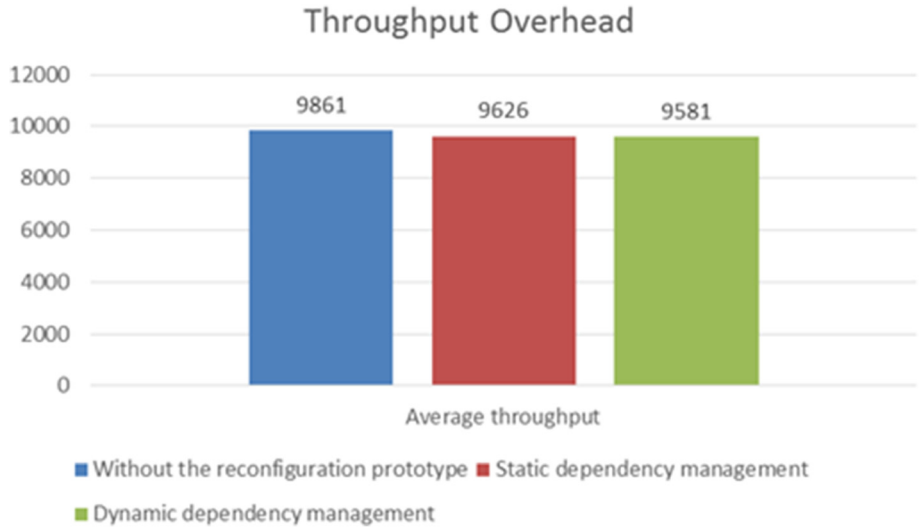

Fig. 20 Overhead in terms of throughput imposed by the reconfiguration prototype

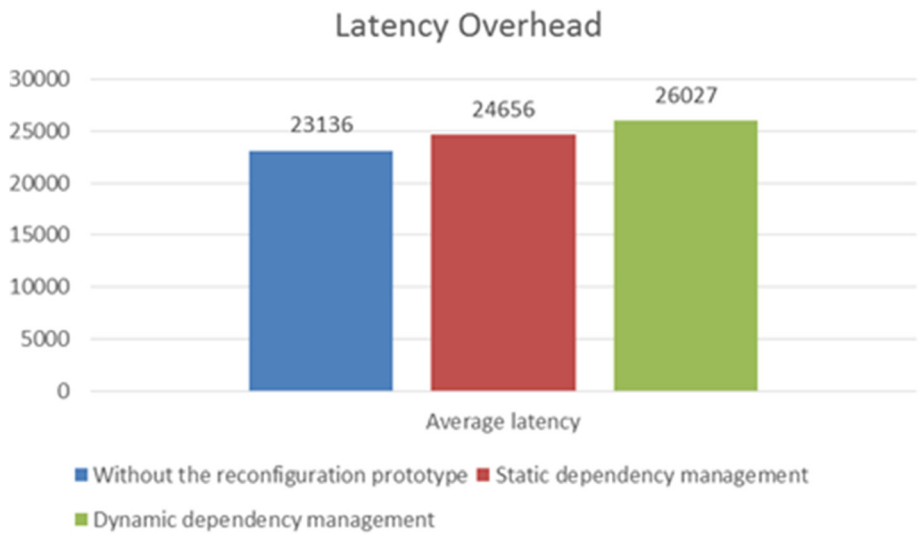

Fig. 21 Overhead in terms of tuples' latency imposed by the reconfiguration prototype

production rate have minor impact on the update time (see Section 4.2.1), we conducted this experiment with 300 CNs and 15,000 tuples/s. Table 1 shows that, as soon as the CN reconnects, it took 31.50 ms to complete the reconfiguration. The experiments show that our mechanism performs consistent reconfigurations in few milliseconds and supports mobile CN reconnections, as well as it does not disrupt the system (Table 3).

4.3 Quiescent vs non-quiescent approach

In order to compare our system with another version that utilizes the quiescent approach and to show the benefits of our approach, we evaluated the prototype application using both approaches in order to measure the update time, throughput and latency. Our approach was evaluated using the static dependency management and we varied the number of CNs (i.e., patients) from 334 to 3,000, where each CN sends a tuple every second. The number of CNs was chosen taking into consideration the maximum capacity of huge hospitals (i.e., among the largest hospital in the world, the Chris Hani Baragwanath Hospital and the Clinical Centre of Serbia have 3,200 and 3,500 beds, respectively [62, 63]). Considering the scenario of the update of the septic shock evaluation criteria, the reconfiguration performed on the prototype application should replace the components implementing the outdated criteria by the components of the new criteria (i.e., in terms of the prototype application, a change from Figs. 13 and 14).

Table 3 Reconnection time

# CNs	Tuple Production Rate (tuples/s)	Reconnection Time (ms)	Confidence Interval (ms)
300	15,000	31.50	+/−1.43

Analyzing the results of Table 4, the update time using the quiescent approach ranges from 583 ms to 737 ms, while our approach produces fairly stable update times (ranging from 25.72 ms to 27.30 ms). Figure 22 shows that the quiescence has a higher impact on the throughput since the system has to be blocked. More specifically, the quiescent approach causes reduction of 88.70 % (from 1,000 tuples/s to 113 tuples/s) and 61.55 % (from 2,463 tuples/s to 947 tuples/s) in throughput in the scenarios with 1,000 and 3,000 CNs, respectively. On the other hand, our approach causes very small impact on the system's throughput.

While quiescence interfere the tuples' latency for almost two seconds and requires up to 1,126 ms to process a tuple in the scenario with 3,000 CNs (see Fig. 23), our approach interfere the tuples' latency for less than half a second and requires at most 150 ms to process a tuple (see Fig. 24), which is 86.68 % lower than the quiescent latency.

5 Conclusion and future work

In this paper, we propose and validate a non-quiescent approach for dynamic reconfiguration that preserves global system consistency in distributed data stream systems. Unlike many works that require blocking the

Table 4 Update time for the quiescent and non-quiescent approaches

# CNs	Update Time (ms)	
	Quiescent Approach	Non-Quiescent Approach
334	583.46	25.72
1,000	629.62	26.92
3,000	737.28	27.30

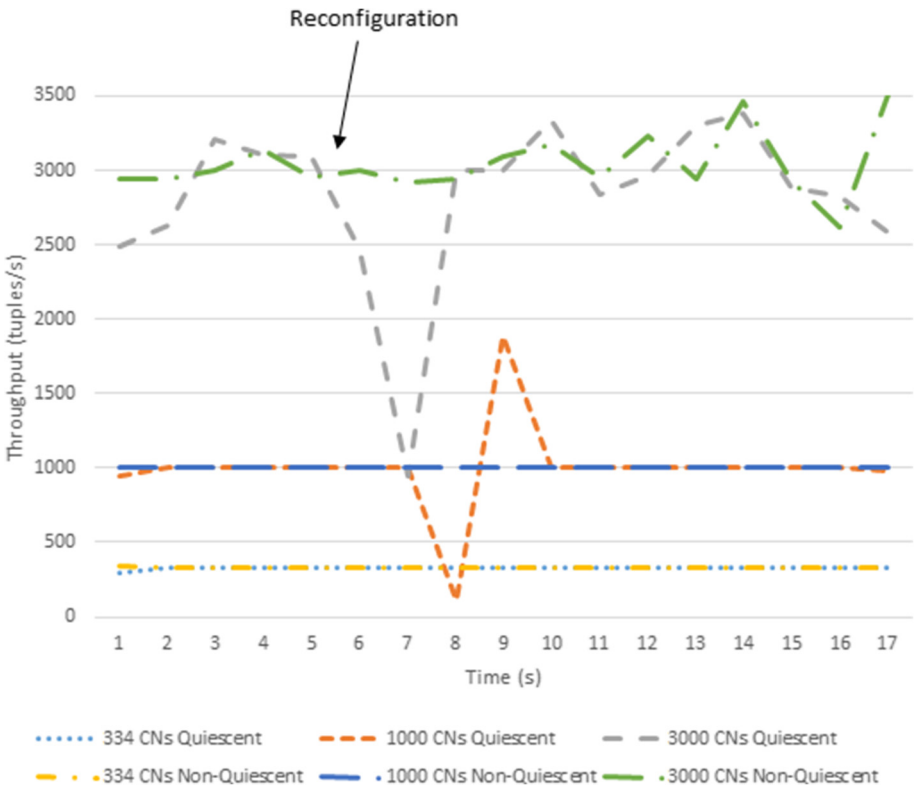

Fig. 22 Throughput comparison between quiescence and our approach

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. Apart from the consistency, our proposal supports stateful components and handles nodes that may disconnect and reconnect at any time. Hence, the main contributions of this paper are (i) a mechanism to enable non-quiescent reconfiguration of distributed data stream systems, and (ii) a prototype middleware that implements the mechanism.

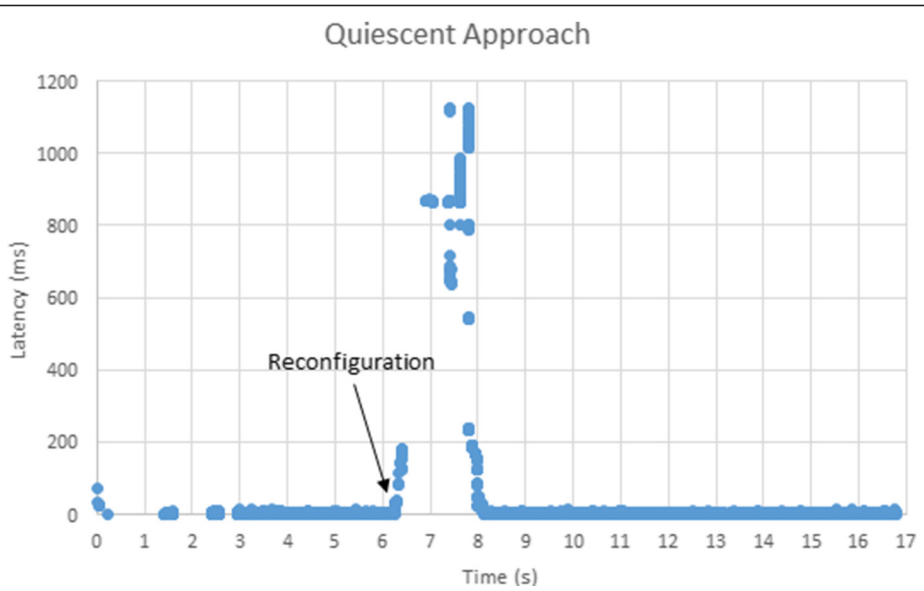

Fig. 23 Latency in the scenario with 3,000 CNs using the quiescent approach

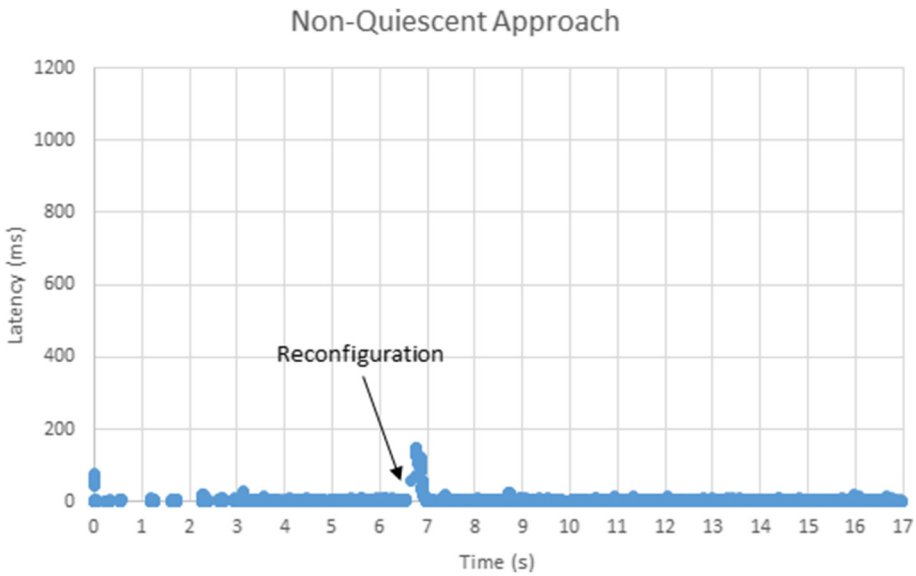

Fig. 24 Latency in the scenario with 3,000 CNs using our non-quiescent approach

Several studies have been conducted in the field of middleware for reconfigurable applications; however, most notable efforts do not take into account problems such as intermittent connectivity of nodes and non-blocking reconfigurations in distributed stream systems. 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.

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 24.07 ms, 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. Another important result is the capability of completing a reconfiguration when a CN reconnects, in which our prototype implementation took 31.50 ms to complete the reconfiguration after the CN's reconnection. For the future, we expect to advance our work along the following lines: (i) explore the topic of reconfiguration policies (i.e., strategies of how to reconfigure the system), (ii) using a more realistic and complex evaluation application to better evaluate the proof of concept prototype and, for instance, to understand the relationship between the reconfiguration size and the evaluated metrics, and (iii) supporting legacy nodes that cannot perform a reconfiguration due to some limitation.

Authors' contributions

ROV is the main contributor of this work, which was undertaken as part of his Ph.D. studies. ROV has participated in the design of this study, designed and implemented the prototype and conducted the evaluation experiments. IV has contributed to the conception of this study. ME, which is supervisor of ROV, have made substantial contributions to the conception and design of the work, and drafted the manuscript. ROV and ME wrote the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Received: 12 February 2016 Accepted: 21 July 2016
Published online: 09 August 2016

References

1. Giuffrida C, Kuijsten A, Tanenbaum AS. Safe and automatic live update for operating systems. ACM SIGPLAN Not. 2013;48:279.
2. Giuffrida C, Iorgulescu C, Tanenbaum AS. Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs. In: Proceedings of the 15th International Middleware Conference. New York: ACM; 2014. p. 133–44.
3. Ma X, Baresi L, Ghezzi C, Panzica La Manna V, Lu J. 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. New York: ACM Press; 2011. p. 245.
4. Ertel S, Felber P. A framework for the dynamic evolution of highly-available dataflow programs. In: Proceedings of the 15th International Middleware Conference on - Middleware'14. New York: ACM Press; 2014. p. 157–68.
5. Andova S, Groenewegen LPJ, de Vink EP. Dynamic adaptation with distributed control in Paradigm. Sci Comput Program. 2014;94:333–61.
6. Hayden CM, Smith EK, Hicks M, Foster JS. State Transfer for Clear and Efficient Runtime Updates. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops. Washington: IEEE Computer Society; 2011. p. 179–84.
7. Hayden CM, Smith EK, Denchev M, Hicks M, Foster JS. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM; 2012. p. 249–64.
8. Kon F. Automatic Configuration of Component-Based Distributed Systems. 2000.
9. Blair G, Bencomo N, France RB. Models@ run.time. IEEE Computer. 2009;42:22–7.
10. Kakousis K, Paspallis N, Papadopoulos GA. A survey of software adaptation in mobile and ubiquitous computing. Enterprise Information Systems. 2010;4:355–89.
11. Li W. QoS assurance for dynamic reconfiguration of component-based software systems. IEEE Trans Softw Eng. 2012;38:658–76.
12. Escoffier C, Bourret P, Lalanda P. Describing Dynamism in Service Dependencies: Industrial Experience and Feedbacks. In: Proceedings of the 2013 IEEE International Conference on Services Computing. Washington: IEEE Computer Society; 2013. p. 328–35.
13. Ramirez AJ, Cheng BHC. Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS'10. New York: ACM Press; 2010. p. 49–58.
14. Costa-Soria C. Dynamic Evolution and Reconfiguration of Software Architectures Through Aspects. 2011.
15. Kramer J, Magee J. The evolving philosophers problem: dynamic change management. IEEE Trans Softw Eng. 1990;16:1293–306.
16. Ghafari M, Jamshidi P, Shahbazi S, Haghighi H. 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. New York: ACM Press; 2012. p. 177.
17. Zhang J, Cheng BHC. Model-based Development of Dynamically Adaptive Software. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06). New York: ACM; 2006. p. 371–80.
18. Vasconcelos, R.O., Endler, M., Gomes, B. de T.P., Silva, F.J. da S. e: Design and Evaluation of an Autonomous Load Balancing System for Mobile Data Stream Processing Based on a Data Centric Publish Subscribe Approach. International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS). 2014;5:19.
19. Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. ACM SIGMOD Rec. 2005;34:42–7.
20. Cugola G, Margara A. Processing flows of information: from data stream to complex event processing. ACM Computing Surveys (CSUR). 2012;44:1–62.
21. Vandewoude Y, Ebraert P, Berbers Y, D'Hondt T. Tranquility: a Low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans Softw Eng. 2007;33:856–68.
22. Ganti R, Ye F, Lei H. Mobile crowdsensing: current state and future challenges. IEEE Commun Mag. 2011;49:32–9.
23. Eisenman SB, Miluzzo E, Lane ND, Peterson RA, Ahn G-S, Campbell AT. The BikeNet mobile sensing system for cyclist experience mapping. In: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems - SenSys'07. New York: ACM Press; 2007. p. 87.
24. DailyMail: Google's war on potholes: Patent reveals plans for cars that detect uneven road surfaces to plot the smoothest routes, http://www.dailymail.co.uk/sciencetech/article-3211572/Google-s-war-potholes-Patent-reveals-plans-cars-detect-uneven-road-surfaces-plot-smoothest-routes.html. Accessed 20 Oct 2015.
25. Schweppe H, Zimmermann A, Grill D. Flexible on-board stream processing for automotive sensor data. IEEE Transac Indus Inform. 2010;6:81–92.
26. Golab L, Özsu MT. Issues in data stream management. ACM SIGMOD Rec. 2003;32:5–14.
27. Babcock B, Babu S, Datar M, Motwani R, Widom J. 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. New York: ACM Press; 2002. p. 1.
28. Cherniack M, Balakrishnan H, Balazinska M, Carney D, Çetintemel U, Xing Y, Zdonik S. Scalable distributed stream processing. In: In CIDR. Asilomar, California: ACM; 2003.
29. Vasconcelos RO, Endler M. A Dynamic Load Balancing Mechanism for Data Stream Processing on DDS Systems. M.Sc Dissertation, Departamento de Informática, PUC-Rio - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro. 2013;74. http://www2.dbd.puc-rio.br/pergamum/biblioteca/php/mostrateses.php?open=1&arqtese=1112660_2013_Indice.html.
30. IBM: Stream Computing Platforms, Applications, and Analytics, http://researcher.ibm.com/researcher/view_group.php?id=2531. Accessed Oct 14 2015.
31. Jacques-Silva G, Gedik B, Wagle R, Wu K-L, Kumar V. Building user-defined runtime adaptation routines for stream processing applications. Proceedings of the VLDB Endowment. 2012;5:1826–37.
32. Gedik B, Andrade H. A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere streams. Softw Pract Exper. 2012;42:1363–91.
33. Vasconcelos RO, Endler M, Gomes B, Silva F. Autonomous load balancing of data stream processing and mobile communications in scalable data distribution systems. Int J Adv Intell Syst (IARIA). 2013;6:300–17.
34. Turaga D, Andrade H, Gedik B, Venkatramani C, Verscheure O, Harris JD, Cox J, Szewczyk W, Jones P. Design principles for developing stream processing applications. Software—Practice & Experience - Focus on Selected PhD Literature Reviews in the Practical Aspects of Software Technology. 2010;40:1073–104.
35. Kleiminger W, Kalyvianaki E, Pietzuch P. Balancing load in stream processing with the cloud. In: 2011 IEEE 27th International Conference on Data Engineering Workshops. New York: IEEE; 2011. p. 16–21.
36. EsperTech: Esper Enterprise Edition: Enterprise ready Event Processing and CEP platform, http://www.espertech.com/products/esperee.php. Accessed 19 Oct 2015.
37. Zhang Q, Cheng L, Boutaba R. Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl. 2010;1:7–18.
38. Cook DJ, Das SK. Pervasive computing at scale: transforming the state of the art. Pervasive Mobile Comput. 2012;8:22–35.
39. Kshemkalyani AD, Singhal M. Distributed Computing: Principles, Algorithms, and Systems. New York, NY, USA: Cambridge University Press; 2011.
40. Chen H, Yu J, Hang C, Zang B, Yew P-C. Dynamic software updating using a relaxed consistency model. IEEE Trans Softw Eng. 2011;37:679–94.
41. Hicks M, Nettles S. Dynamic software updating. ACM Trans Program Lang Syst. 2005;27:1049–96.
42. Subramanian S, Hicks M, McKinley KS. Dynamic Software Updates: A VM-centric Approach. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM; 2009. p. 1–12.
43. Makris K, Ryu KD. Dynamic and Adaptive Updates of Non-quiescent Subsystems in Commodity Operating System Kernels. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. New York: ACM; 2007. p. 327–40.
44. Morrison JP. Flow-Based Programming, 2nd Edition: A New Approach to Application Development. Paramount, CA: CreateSpace; 2010.
45. Nouali-Taboudjemat N, Chehbour F, Drias H. On Performance Evaluation and Design of Atomic Commit Protocols for Mobile Transactions. Distrib Parallel Databases. 2010;27:53–94.
46. Vasconcelos RO, Vasconcelos I, Endler M. A Middleware for Managing Dynamic Software Adaptation. In: 13th International Workshop on Adaptive and Reflective Middleware (ARM 2014). Bordeaux: In conjunction with ACM/IFIP/USENIX ACM International Middleware Conference 2014; 2014. p. 6.
47. Ghafari M, Jamshidi P, Shahbazi S, Haghighi H. Safe Stopping of Running Component-Based Distributed Systems: Challenges and Research Gaps. In: Proceedings of the 2012 IEEE 21st International Workshop on Enabling

Technologies: Infrastructure for Collaborative Enterprises. Washington: IEEE Computer Society; 2012. p. 66–71.
48. Giuffrida C, Tanenbaum AS. Cooperative update: a new model for dependable live update. In: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp'09. New York: ACM Press; 2009. p. 6.
49. Pina L, Veiga L, Hicks M. Rubah: DSU for Java on a stock JVM. ACM SIGPLAN Not. 2014;49:103–19.
50. Lamport L. Time, clocks, and the ordering of events in a distributed system. Commun ACM. 1978;21:558–65.
51. Chandy KM, Lamport L. Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst. 1985;3:63–75.
52. Bakshi A, Talaei-Khoei A, Ray P. Adaptive policy framework: a systematic review. J Netw Comput Appl. 2013;36:1261–71.
53. Forax, R., Duris, E., Roussel, G (2005). Reflection-based implementation of Java extensions: the double-dispatch use-case. In: Proceedings of the 2005 ACM symposium on Applied computing - SAC'05. 1409.
54. Ortin F, Conde P, Fernandez-Lanvin D, Izquierdo R. The runtime performance of invokedynamic: an evaluation with a java library. IEEE Softw. 2014;31:82–90.
55. David L, Vasconcelos R, Alves L, Andre R, Baptista G, Endler M. A Communication Middleware for Scalable Real-Time Mobile Collaboration. In: IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). New York: IEEE; 2012. p. 54–9.
56. David L, Vasconcelos R, Alves L, André R, Endler M. A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes. J Internet Serv Appl (JISA). 2013;4:16.
57. Lee SI, Ghasemzadeh H, Mortazavi B, Lan M, Alshurafa N, Ong M, Sarrafzadeh M. Remote patient monitoring: what impact can data analytics have on cost? In: Proceedings of the 4th Conference on Wireless Health - WH'13. New York: ACM Press; 2013. p. 1–8.
58. Forbes: 4 Interesting Tech Trends In Patient Monitoring, http://www.forbes.com/sites/robertszczerba/2014/12/10/4-interesting-tech-trends-in-patient-monitoring. Accssed 14 July 2016.
59. Catalyst, H.: The Year of Healthcare Data Analytics, https://www.healthcatalyst.com/2014-Year-Healthcare-Data-Analytics. Accessed 14 July 2016.
60. MDCalc: SIRS, Sepsis, and Septic Shock Criteria, http://www.mdcalc.com/sirs-sepsis-and-septic-shock-criteria. Accessed 15 Mar 2016.
61. Singer M, Deutschman CS, Seymour CW, Shankar-Hari M, Annane D, Bauer M, Bellomo R, Bernard GR, Chiche J-D, Coopersmith CM, Hotchkiss RS, Levy MM, Marshall JC, Martin GS, Opal SM, Rubenfeld GD, van der Poll T, Vincent J-L, Angus DC. The third international consensus definitions for sepsis and septic shock (Sepsis-3). JAMA. 2016;315:801–10.
62. Hospital, C.H.B.: The Chris Hani Baragwanath Hospital, South Africa, https://www.chrishanibaragwanathhospital.co.za/. Accessed 15 July 2016.
63. Saturn: Clinical Centre of Serbia – (CCS), http://www.saturn-project.eu/about-saturn/consortium-partners/clinical-centre-of-serbia-ccs/. Accessed 15 July 2016.