A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications

Mehdi Belkhiria, Cédric Tedeschi

▶ **To cite this version:**

Mehdi Belkhiria, Cédric Tedeschi. A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications. Auto-DaSP 2019 - Third International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing, Aug 2019, Göttingen, Germany. pp.1-12. hal-02171172

HAL Id: hal-02171172

https://inria.hal.science/hal-02171172

Submitted on 2 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications

Mehdi Belkhiria Cédric Tedeschi

Univ Rennes, Inria, CNRS, IRISA, Rennes, France
mehdi.belkhiria@irisa.fr, cedric.tedeschi@inria.fr

Abstract. Stream Processing deals with the efficient, real-time processing of continuous streams of data. Stream Processing engines ease the development and deployment of such applications which are commonly pipelines of operators to be traversed by each data item. Due to the varying velocity of the streams, autoscaling is needed to dynamically adapt the number of instances of each operator. With the advent of geographically-dispersed computing platforms such as Fog platforms, operators are dispersed accordingly, and autoscaling needs to be decentralized as well. In this paper, we propose an algorithm allowing for scaling decisions to be taken and enforced in a fully-decentralized way. In particular, in spite of scaling actions being triggered concurrently, each operator maintains a view of its neighbours in the graph so as no data message is lost. The protocol is detailed and its correctness discussed. Its performance is captured through early simulation experiments.

1 Introduction

The need for near real-time processing of continuously produced data led to the development of the Stream Processing (SP) computing paradigm. A stream processing application is typically a graph of operators that each data item will traverse, data processing being pipelined. Stream Processing is becoming ubiquitous and is being applied in many domains, ranging from social media to military applications. Stream Processing Engines (SPEs) have been proposed to ease the development of these applications and their deployment over utility computing platforms [5, 14, 18]. From the data perspective, each data item traverses the graph of operators. From the processing perspective, each operator collects the data stream from its predecessors in the graph, applies its own transformation and creates a new stream of data, sent to its successors in the graph. Operators have different costs, for instance in terms of processing time and CPU utilization.

Parallelism within Stream Processing applications can be achieved in different manners. The first one, already mentioned, is pipeline parallelism: data items can be processed concurrently on different portions of the graph. The second one is data parallelism: if stateless or partitioned, an operator can process several data items at the same time, provided it is scaled accordingly. Because the input data stream generally varies in size over time, this scaling needs to be

adjusted dynamically in time so as to be able to ensure parallelism while minimising computing resource waste. This mechanism, referred to as *autoscaling*, is commonly designed and implemented in SPEs as a side system taking glocal decisions about the scaling (in or out) of operators based on real-time metrics collected about operators and the current stream velocity.

Assuming a dedicated, centralized autoscaling subsystem can become difficult in geographically dispersed platforms, such as Edge or Fog computing platforms. Such platforms are becoming realities and typically support mobile-based or IoT-based applications where Stream Processing is the adequate computing paradigm to be used [20]. Moving Stream Processing to the edge brings new challenges due to the very essance of these platforms: they gather a lot of small, geographically-dispersed computing resources. Accurately monitoring such platforms in a centralized fashion becomes difficult due to the constraints on both network and computing resources. In this paper, we assume a distributed deployment of the graph of operators. The infrastructure gathers geographically-dispersed compute nodes and each operator is placed over a compute node which is potentially distant from the compute nodes hosting its predecessor and successor operators in the graph. Consequently, each operator needs to maintain the addresses of the compute nodes hosting their neighbours in the graph. We also assume no central scaling authority is available and that operators take their own scaling decisions independently from each others. Then, as instances (or *replicas*) of operators appear and disappear at many places of the operators dynamically, one challenge is to be able to maintain on each operator a correct view of its predecessors and successors so that no message is lost: wrongly assume some node is still the host of one instance of one of our successor may cause one operator to send some message to a deleted instance, causing in turn data loss. In the following, we propose a fully-decentralized algorithm where scaling decisions are taken independently and the graph maintained so as to ensure no data message is lost. Operators exist in a dynamically adjusted number of *instances*. Each instance takes its own probabilistic scale-in or scale-out decisions, based on local monitoring so as to globally converge towards the *right* number of instances in regard to the current velocity level for this operator. Each time it decides a scaling operator, an instance also triggers a protocol to ensure the correct maintenance of the graph in spite of concurrent scale decisions.

Related work is presented in Section 2. In Section 3, the system model used to describe applications and platforms considered is given. Our decentralized scaling protocol, including the scaling policy in both in and out cases, as well as a sketch of proof regarding correctness facing concurrency is presented in Section 4. Simulation results are given in Section 5.

2 Related Work

Autoscaling in stream processing has been the subject of a recent series of works [2,12] addressing i) the dynamic nature of the velocity of the data stream, and ii) the difficulty of estimating prior to execution the computation cost of the

operators, that can vary significantly from one operator to another. The scaling problem can be tackled either statically, *i.e.*, prior to the actual deployment of the application or online, so as to dynamically adapt the amount of computing power dedicated to each operator.

Static approaches typically rely on the prior-to-execution analysis of the graph so as to infer its *optimal* parallelization. Schneider et al. [17] propose a heuristic-based traversal of the graph so as to group operators together in different *parallel areas*, each area being a contiguous set of stateless operators, stateful operators being considered here again as not trivial to parallelize. While this static analysis is a necessary first step, it is unable to find a continuously accurate level of parallelism able when facing changes in the velocity of the incoming data stream. Accuracy refers here to the ability to find the right amount of instances, and avoid both over and under-provisioning.

Dynamic scaling generally relies on three operations: fusion, fission, and deletion [12]. *Fusion* refers to the merging of two contiguous operators hosted by two different compute nodes, into a single compute node. While this increases the load on the compute node chosen, *fusion* primarily targets the reduction of the network load by keeping within one node the traffic initially traversing the network links between the two nodes. Fusion is not a scaling action *per se*, and relates more to a consolidation of the placement of operators over the compute nodes. *Fission* (or *scale-out*) refers to operators' duplication: a new instance of operator gets started. It increases the level of parallelism of this operator provided the new thread or process spawned to support it leverages computing resources that were not fully used prior to the fission (Fission can rely over either vertical or horizontal scaling, again relating to a placement problem [15,16]). Note that, in practice, the fission mechanism is influenced by the *statefulness* of the operator: Maintaining the state of a stateful operator when it is fissioned requires to merge the partial states maintained independently over the instances. Statefulness is an issue in scaling but not our primary concern here. *Deletion* (or *scale-in*) is fission's inverse operation. It consists in removing running instances of a given operator, typically when the operator's incoming load gets reduced.

In practice, dynamic scaling systems typically rely on two elements [8, 10, 11, 19]: i) a centralized subsystem collecting up-to-date information about the network traffic and available resources, so as to be able to take relevant decisions to optimize a certain performance metric, and ii) a scaling policy to decide *when* to trigger a scale-out, scale-in or reconfiguration. Some of these works focus on monitoring the CPU utilization so as to detect bottlenecks and trigger a scaling-out phase, in particular for partitioned stateful operators, which requires to split and migrate the state of the operator between the evolving set of instances [8]. Designed as an extension of Storm [18], T-Storm [19] introduces a mechanism of dynamic load rebalance triggered periodically, with a focus on trying to reduce internode communication by grouping operators). Aniello et al. proposes a similar approach [1]. StreamCloud [11] provides a set of techniques to identify parallelizable zones of operators into which the whole graph is split, zones being delimited by stateful operators. The splitting algorithm shares some similarities

with the work in [17]: each zone can be parallelized independently. Yet, on top of this splitting mechanism, dynamic scheduling is introduced to balance the load at the entry point of each zone. Finally, some work combines fission and deletion so as to continuously satisfy the SASO properties (Settling time, Accuracy, Stability, Overshoot) [10]. The requirement is to be able to dynamically allocate the right amount instances ensuring the performance of the system (accuracy), that this number is reached quickly (settling time), that it does not oscillate artificially (stability) and that no resource is used uselessly (overshoot). While their objectives are similar to those of the present work, they still rely over a centralized authority to monitor the system, decide on the scaling operations and enforce them. The present work offers a decentralized vision of the problem.

Decentralizing the management of stream processing frameworks has been the subject of different works [4, 6, 7, 13, 16]. DEPAS [4] is not specifically targeted at stream processing and focus on a multi-cloud infrastructure with local schedulers taking decisions independently. The similarity between DEPAS and the present work stands in that autonomous instances take scaling decisions based on a probabilistic policy. Yet, our main focus is also different: we are mainly interested in providing a graph maintenance algorithm minimizing downtime. More specifically targeted at stream processing, Pietzuch et al. [16] proposed a Stream-Based Overlay Network (SBON) that allows to map stream processing operators over the physical network. Hochreinter et al. [13] devise an architectural model to deploy distributed stream processing applications. Finally, Cardellini et al. [6, 7] proposed a hierarchical approach to the autoscaling problem, following a hierarchical approach combining a threshold-based local scaling deision with a central coordination mechanism to solve conflicts between decisions taken independently and limit the number of reconfigurations.

Autoscaling generally assumes a pause-and-restart: when a scaling operation takes place, the application is paused. It gets restarted once the reconfiguration is over. Reconfiguration is needed in particular when dealing with the scaling of partitioned stateful operators which requires to split and migrate its state dynamically. In the following, assuming stateless operators, we devise a fully-decentralized autoscaling protocol that does not require to pause data processing during reconfigurations. While making the problem easier, assuming stateless operators appear to be a reasonable first step. To our knowledge, no such fully-decentralized proper protocol was proposed assuming neither stateful, nor stateless protocols.

3 System Model

Platform Model. We consider a distributed system composed of an unbounded set of (geographically dispersed) homogeneous compute nodes. These nodes can be either physical nodes or virtual machines. We abstract out the allocation of a new node through the `createNode()` primitive. In practice, *homogeneity* means that all virtual machines allocated have the same size. Compute nodes are assumed reliable: they can be deallocated, but cannot crash. Nodes communicate

in a partially synchronous model [9] using FIFO reliable channels: A message reaches its destination in a finite time, and two messages sent through the same channel are processed in the same order they were sent. Sending a message is done through the `send(type,ctnt,dest)` non-blocking method. `type` denotes the message type and `ctnt` its content. `ctnt` actual structure varies depenting on `type`. `dest` is the address of the destination node. The higher-level communication primitive `sendAll(type, ctnt, dests)` sends the same message to all nodes in `dests`.

Application Model. We consider stream processing applications represented as directed pipelines in which vertices represent operators to be applied on each input record and edges represent streams between these operators. We assume stateless operators. At starting time, each operator is launched on one particular compute nodes, and each compute node hosts a single replica. Then, the scaling mechanism can add or remove replicas. Each replica of an operator is referred to as an *operator instance* (OI) in the following. OIs running the same operator are referred to as *siblings*. The load of an operator is shared equally between all of its instances. Each operator O_i can exist in several instances OI_ij where i is the id of the operator and j the id of the instance. In the example of Fig. 1, the pipeline is made of three operators. At some point, scaling out introduced two new instances for the middle operator. The application follows a purely distributed configuration: due to the geographic dispersion of nodes and for the sake of scalability, the view of the graph on each instance is limited to the instances of their successor and predecessor operators.

4 Scaling Algorithm

The algorithm proposed and described in this section enables each OI to decide locally and independently when to get duplicated or deleted. The algorithm is run periodically on each OI (with possibly different frequencies). The algorithm starts with the decision phase in which the OI checks its current load. Assuming OIs are homogeneous and the load fairly distributed amongst instances, OIs are able to take uncoordinated decisions leading to a global accurate number of instances. Once an OI decides to get duplicated or deleted, it actually executes the action planned and ensures its neighbours are informed of it. Section 4.1 details the decision process and Section 4.2 gives the details of the maintenance protocol enforcing the decision taken.

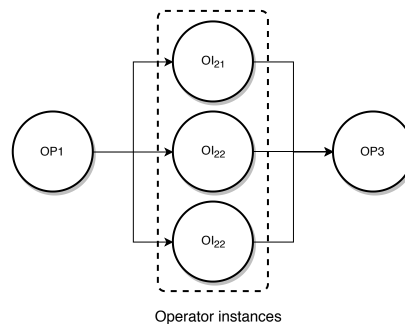

Fig. 1. Scaling a 3-stage pipeline.

4.1 Scaling decision

Duplication decision. Let C denote the capacity of the nodes, i.e. the number of records they can process per time unit. Let n_t denote the replication factor (the number of OIs) for this operator at time t, i.e. the number of current OIs running this operator.[1] Let l_t the current load experienced by the OI, *i.e*, the number of records received during the last time unit. Finally, r, with $0 < r \leq 1$ denotes the desired load level of operators, typically a parameter set by the user. It represents the targeted ratio between load and capacity of each node. The objective for an OI is to find the replication factor to be applied to itself so all OIs for this operator globally reach a load level of r. The current load for an operator can be locally estimated as $L_t = l_t \times n_t$. Each node contributes to the needed scaling equally following a *local replication factor*. The desired load for an OI is $r \times C$. The number of OIs to support L_t while ensuring that each OI reaches the desired load is then $\frac{L_t}{r \times C}$. Thus the number of nodes to add globally is $n_{diff} = \frac{L_t}{r \times C} - n_t$. Locally, this target is translated into the local replication factor to be applied $p = \frac{n_{diff}}{n_t}$. If $p < 1$, it is interpreted as a duplication probability: the node will get duplicated with probability p. Otherwise, the node will get duplicated $\lfloor p \rfloor$ times and then one final time with probability $p - \lfloor p \rfloor$.

Deletion decision. The inverse decision, triggered when the load is below a certain threshold, follows the same principle. Yet, the factor calculated in this case is a probability. Note that there is a risk that all OIs for a given operator take this decision at the approximate same time, leading to a collective termination, and to the disappearance of this operator. This problem is solved by introducing a particular node (called the *operator keeper*) that cannot terminates itself whatever its load. The deletion/duplication factor is materialized through the `getProbability(C, r, l_t, n_t)` function. The `applyProba(p: real)` function transforms a probabability into a boolean stating whether the deletion or duplication action will actually take place.

4.2 Scaling protocol

Algorithm 1 gives the pseudo-code of the protocol triggered once the duplication decision has been taken. It takes two extra inputs: i)$thres_{\uparrow}$, the value above which the load level triggers the duplication policy, and ii) the list of successors and predecessors of the current OI. The first part of the algorithm consists in calculating the amount of duplication needed to reach the targeted load ratio r (in Lines 2-4). From Lines 5 to Lines 7, the calculated amount of nodes get started. Newly spawned OIs are not yet `active`: they are idle, waiting for a message of the current node to initialize its neighbors and start processing incoming data, which is stored in some entry queue in the meantime. The current node, in Lines 9-11, spreads the information about the new nodes to its own neighbors.

[1] In real settings, while a node i does not need to maintain the set of its siblings, the information n_t can be sent by one instance of node i's predecessor which knows n_t since maintaining a view of node i's OIs which are its successors.

A counter of the expected number of responses is initialized. To validate the duplication and actually initialize the new, initially idle, nodes, the OI needs to collect the acknowledgement of all of its neighbors.

Algorithm 1 Scale-out protocol.

```
Input: thres↑: threshold
Input: succs, preds: arrays of successors and predecessors
 1: procedure opScaleOut()
 2:   p ← getProbability(C, r, l_t, n_t)
 3:   newAddrs ← []
 4:   n ← ⌊p⌋ + applyProba(p)
 5:   if n > 1 then
 6:     for i ← 1 to n do
 7:       newAddrs.add(createNode())
 8:     end for
 9:     sendInformation("duplication", succs, preds, newAddrs)
10:     nbAck ← 0
11:     nbAckExpected ← |succs| + |preds|
12:   end if
13: upon receipt of ("duplication", addrs) from p
14:   if p ∈ succs then
15:     if isActive then
16:       succs = succs ∪ addrs
17:     else
18:       succsToAdd = succsToAdd ∪ addrs
19:     end if
20:   else if p ∈ preds
21:     if isActive then
22:       preds = preds ∪ addrs
23:     else
24:       predsToAdd = predsToAdd ∪ addrs
25:     end if
26:   send("duplication_ack", p)
27: upon receipt of ("duplication_ack")
28:   nbAck + +
29:   if nbAck = nbAckExpected then
30:     for all newSibling in newAddrs do
31:       send("start", succs, preds, newSibling)
32:     end for
33:   end if
34: upon receipt of ("start", succs_, preds_) from p
35:   succs = succs_ ∪ succsToAdd \ succsToDelete
36:   preds = preds_ ∪ predsToAdd \ predsToDelete
37:   isActive ← true
```

In Lines 14-19, the case of a *duplication* message coming from a successor is processed: the addresses received are new predecessors and are added to the corresponding set. If the node receiving the message is itself not yet *active*, i.e., it is itself a new node waiting for its *start* message, it will store the new neighbour in a particular *succsToAdd* set containing future neighbors: the node may store incoming data but cannot yet send data to its successors to avoid lost tuples, as reviewed in Section 4.3. Then, in Lines 20-25, the case of a duplication message received from a predecessor is processed similarly. Finally, the node acknowledges the message to the duplicating node by sending a *duplication_ack* message. Once all acknowledgements have been received by the duplicating OI, the new nodes can become active and start processing records. To this end, in

Line 31 of Algorithm 1, the duplicating OI sends a *start* message to all of its new siblings. On receipt — refer to Lines 35-36) — the new siblings initialize the sets of their neighbors by combining the sets sent by the duplicating OI and the possible information received in the meantime, stored in $*ToAdd$ and $*ToDelete$ variables.

Let us now review the termination protocol, detailed in Algorithm 2. It is very similar to the scale-out protocol. Algorithm 2 first shows how the current OI, before self-termination ensures that every node pertained by the deletion (its neighbours) is informed of it. On receipt of this upcoming termination information, we again have to consider two cases, depending whether the receiving node is active or not: if it is, then the node is simply removed from the list of its neighbors (either from *pred* or *succ*) and an acknowledgement is sent back. Otherwise, the node is stored in a *to be deleted* set of nodes, that will be taken into account at starting time. The final step consists, on the node about to terminate, to count the number of acknowledgements. As discussed in Section 4.3, the terminating node must wait for all the acknowledgement of the nodes it considers as neighbors. Once it is done, it flushes its data queue and triggers its own termination.

Algorithm 2 Scale-in protocol.

```
Input: thres↓: threshold
 1: procedure operatorScale − In()
 2:   p ← getProbability(C, r, l_t, n_t)
 3:   if applyProba(p) then
 4:     sendInformation("deletion", succs, preds, me)
 5:     nbAck ← 0
 6:     nbAckExpected ← |succs| + |preds|
 7:   end if
 8: upon receipt of ("deletion", addr) from p
 9:   if P ∈ succs then
10:     if isActive then
11:       succs ← succs \ addr
12:     else
13:       succsToDelete ← succsToDelete ∪ addr
14:     end if
15:   else if p ∈ preds
16:     if isActive then
17:       preds ← preds \ addr
18:     else
19:       predsToDelete ← predsToDelete ∪ addr
20:     end if
21:   send("deletion_ack", p)
22: upon receipt of ("deletion_ack")
23:   nbAck + +
24:   if nbAck = nbAckExpected then
25:     // wait current tuples to be processed
26:     terminate()
27:   end if
```

The global algorithm checks periodically the current load *vs* the thresholds and starts the corresponding algorithm as needed, each OI, except the operator keepers, doing that independently at possibly different times.

4.3 Correctness

A graph is said *stable* when for every OI, the set of its successors is equal to the set of OI having it as a predecessor, and the same goes reversing successors and predecessors. In such a situation, following the assumptions that nodes are reliable and that messages reach their destination in a finite time, no tuple is lost.

Let us now review the possible perturbations in this graph. The simplest case is a single duplication triggered in a stable graph. Recall that new nodes are first spawned (through `createNode()`) and become `active` once they have received a *start* message. Yet, a spawned not-yet-active node can store incoming data messages: becoming active means that it will start process them and send the result to its successors. Note that a node needs to know where to send messages (its successors) but does not need to know its predecessors to receive messages from them. For instance, using message-oriented middleware, does not require a node to know where the messages are coming from to receive them. We simply need to be sure that nodes to which messages are sent have been spawned. Clearly, starting from a stable graph, successors of the new instances are already running when the new instances become active. Also, when predecessors of the new instances receive the notification about them, new instances are already spawne, since calls to `createNode()` are made (and return) before the information is sent to the predecessors. After this period of instability, everyone has received and updated its sets of neighbours so the graph becomes stable again. Having multiple concurrent duplication processes on different OIs of an operator does not bring any difficulty, OIs processing messages one by one.

Let us now study the deletion of a single node at a time starting from a stable graph. Messages could be lost in case the predecessors of the deleted node keep sending message to it. As per the algorithm, to trigger the actual termination (calling `terminate`()), a node needs to receive acks from its predecessors. These acks are sent only after the `deletion` message has been received. What we assume here is that before sending the `deletion_ack` message, an OI communicates with its data processing layer so as to inform it of the upcoming deletion. The data processing layer takes it into account by stopping emitting messages to the about-to-be-deleted OI. Yet, the last message sent contains a particular *last message* stamp. The `terminate`() primitive is assumed to return only after these specifically marked messages has been received from each predecessor, ensuring no message is lost. Neighbors of the deleting node are informed of the deletion and their sets of neighbours are updated, so the graph becomes stable again. Multiple concurrent deletions do not bring any more difficulty.

Let us now study the case of having concurrent duplication and deletion. If triggered by nodes that are not neighbours , this does not bring any particular difficulty. If they are triggered by nodes that are not neighbours, this is not a problem either. A more difficult case to check is when two neighbouring nodes N1 and N2 take these antagonist actions. Say N2 is amongst the successors of N1. Assume N1 triggers a duplication while N2 triggers its own termination. Consequently, N1 sends a `duplication` message while N2 sends a `deletion`

message at the approximate same time. Let us assume that N2's deletion message takes far longer to reach N1 than N1's message to reach N2. Assuming channels are FIFO we distinguish two cases: The first case is when N2 sends the `deletion` message before processing N1's duplication message. In this case, due to the FIFO assumption, N1 will first receive N2's deletion message and remove N2 from its set of successors, so that, once N1 receives all of the `duplication_ack` from its neighbours (including N2), N1 will send the starting message to the new OI with a set of successors not including N2, leading both N1 and the new OI to not consider N2 as a successor. The second case is when N2 processes the `duplication` message sent by M1 before sending its own `deletion` message. In this case, N2 sends the *duplication_ack* message before the `deletion` message. So they will arrive in this order on N1. On receipt of the first message, N1 still considers N2 as a neighbour for the future OI and may send it to the new OI at starting time. Yet it is not a problem, as N2 now knows about the new OI and will send its deletion message also to it. While not yet active, the new OI will receive the deletion message and keep the information that N2 is to be deleted from the successors at starting time, as enforced by Line 13 in Algorithm 1.

The case of two concurrent duplications is simpler. In case each duplication message arrives before the other one is processed, each node will learn its new neighbour independently anyway and start their new OI with the information of that new neighbours' OI. The case of two concurrent deletions can be solved similarly.

5 Simulation results

In this section, we present early simulation results of our protocol. We developed a discrete-time simulator in Java. Each time step t sees the following operations: a subset of the nodes test the conditions for triggering a scaling operation. In case the protocol is initiated, the first message (`duplication` or `deletion`) is received by the neighbours of the initiating node. The following steps are as follows: messages sent at step t are processed at step $t+1$ and new resulting messages are sent as per the protocol, to be processed at time $t+2$, and so on. Remark that a scale-out operation spans three steps, and a scale-in one spans two. The variation of the workload is modelled by a stochastic process, mimicking a Brownian motion, which allows us to evaluate our algorithm with a quick yet swift variation of the workload. The graph tested is a pipeline composed of 5 operators, each operator having a workload evolving independently. Initially, each operator is duplicated on 7 OIs. Compute nodes hosting OIs have a processing capacity of processing 500 tuples per time step. The other parameters are: $r = 0.7$, $thres_{\downarrow} = 0.8$, and $thres_{\uparrow} = 0.6$. Nodes try to start the scaling protocol every 5 steps.

Our algorithm's ability to quickly adapt to the load's variations and reach an adequate number of instances through local decisions is illustrated by Fig. 2. The blue curve shows the load (aggregated number of tuples received by the nodes, whatever their position in the pipeline), and the red curve shows how the global number of OIs evolved during the experiment. Firstly, we observe that

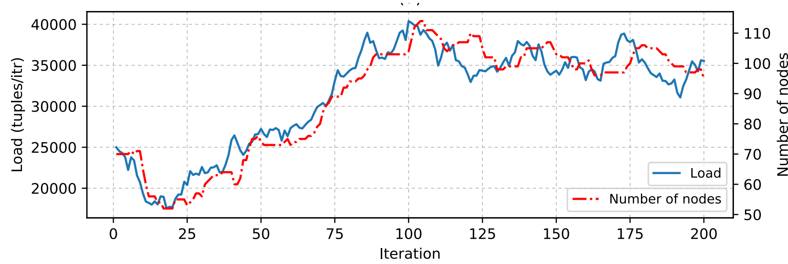

Fig. 2. Number of nodes *vs* load.

the number of nodes decreases with the decline of the workload during the first 25 iterations. Then, it increases until reaching the peak of 114 nodes at iteration 104 quickly after the load itself reached the peak of 40396 tuples per step at iteration 100. Then, the load (and consequently the number of OIs) does not fluctuate significantly. Secondly, we observe that the number of nodes can scale quickly. The delay between a variation in the load and the adaptation can be small. Nodes, without coordination, based on decisions taken locally, are able to start or remove nodes in a *batch* fashion, the burden of starting or removing these nodes being shared by the existing nodes.

More simulation results are available in a research report [3], which also gives few hints about the network overhead incurred by the protocol.

6 Conclusion

This paper presents a fully decentralized autoscaling algorithm for stream processing applications to be deployed over a geographically-dispersed set of resources. The algorithm relies on independent, local autoscaling decisions taken by operators having only a partial view of the load and maintaining only a local view of the graph. Future work will consist in relaxing some of the assumptions regarding the algorithm, in particular the fault model and the statelessness of operators. On the practical side, the prototype of a decentralized stream processing engine is being developed, including the scaling algorithm presented.

References

1. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS'13). pp. 207–218. ACM, Arlington, USA (2013)
2. de Assunção, M.D., Veith, A.D.S., Buyya, R.: Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. J. Network and Computer Applications **103**, 1–17 (2018)
3. Belkhiria, M., Tedeschi, C.: Decentralized Scaling for Stream Processing Engines (May 2019), `https://hal.inria.fr/hal-02127609`, working paper or preprint

4. Calcavecchia, N.M., Caprarescu, B.A., Di Nitto, E., Dubois, D.J., Petcu, D.: DEPAS: a Decentralized Probabilistic Algorithm for Auto-scaling. Computing **94**(8), 701–730
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
6. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Distributed QoS-aware Scheduling in Storm. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. pp. 344–347. DEBS '15, ACM, New York, NY, USA (2015)
7. Cardellini, V., Presti, F.L., Nardelli, M., Russo, G.R.: Decentralized self-adaptation for elastic data stream processing. Future Generation Comp. Syst. **87**, 171–185 (2018)
8. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: ACM SIGMOD'13). pp. 725–736. ACM, New York, USA (2013)
9. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (Apr 1988)
10. Gedik, B., Schneider, S., Hirzel, M., Wu, K.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1447–1463 (2014)
11. Gulisano, V., Jimnez-Peris, R., Patio-Martnez, M., Soriente, C., Valduriez, P.: Streamcloud: An elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems **23**(12), 2351–2365 (Dec 2012)
12. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4), 46:1–46:34 (Mar 2014)
13. Hochreiner, C., Vgler, M., Schulte, S., Dustdar, S.: Elastic stream processing for the internet of things. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). pp. 100–107 (June 2016). https://doi.org/10.1109/CLOUD.2016.0023
14. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 239–250. SIGMOD '15, ACM, New York, NY, USA (2015)
15. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: Resource-aware scheduling in storm. In: Proceedings of the 16th Annual Middleware Conference. pp. 149–161. Middleware '15, ACM, New York, NY, USA (2015)
16. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: 22nd International Conference on Data Engineering (ICDE'06). pp. 49–49 (April 2006)
17. Schneider, S., Hirzel, M., Gedik, B., Wu, K.: Auto-parallelizing Stateful Distributed Streaming Applications. In: International Conference on Parallel Architectures and Compilation Techniques, PACT '12. pp. 53–64. Minneapolis, USA (Sep 2012)
18. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.V.: Storm@twitter. In: International Conference on Management of Data (SIGMOD 2014). pp. 147–156. Snowbird, USA (Jun 2014)
19. Xu, J., Chen, Z., Tang, J., Su, S.: T-storm: Traffic-aware online scheduling in storm. In: IEEE 34th International Conference on Distributed Computing Systems (2014)
20. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., Jue, J.P.: All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey. CoRR **abs/1808.05283** (2018)