Aging-related Performance Anomalies in the Apache Storm Stream Processing System

Massimo Ficco, Roberto Pietrantuono, and Stefano Russo

Università degli Studi della Campania "Luigi Vanvitelli", Via Roma 29, I-81031 Aversa, IT

Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, IT

Abstract

Event stream processing has recently emerged as a popular paradigm for implementing high-volume distributed (near-)real time data processing applications. Several open source systems are today available, supporting the development of such applications, many of which developed with the technologies of the Apache Software Foundation. These so called *stream processors* are long-running complex software systems which may be affected by *software aging*, a well-known phenomenon among operation engineers, consisting of a progressive increase in the failure rate or in performance degradation of a software system over time.

We address the problem of identifying symptoms and sources of software aging in the Apache Storm event stream processing system; this helps to identify proper strategies to prevent or mitigate anomalous behaviors in production environments. To this aim, we present an experimental study investigating aging manifestations in a popular system, namely Apache Storm. Results show that Storm presents anomalous behaviors in long runs, which prevent some topologies from working continuously. These can be attributed to software aging, due to Storm internal resource management mechanisms influenced by the garbage collector and the memory assigned to worker processes. We discuss the aging-related Apache Storm behaviors, and we experiment *rejuvenation* actions, showing that they are actually able to remove them.

Keywords: Software aging, event stream processing, Apache Storm, cloud.

[1]Correspondence to: Massimo Ficco, e-mail: massimo.ficco@unicampania.it.

1. Introduction

Event stream processing (ESP) has recently emerged as a popular paradigm for implementing high-volume data processing applications. While traditional data processing models use to persist data to databases and then execute queries on the stored data, ESP applications perform complex queries on incoming streams of data to produce timely results in reaction to events observed in the processed data [1].

The development of ESP applications is supported by so called *distributed real time stream processing systems*, i.e., software technologies capable of deploying tasks (which are part of the stream processing application) over a cloud architecture or in general in a distributed execution environment. Stream processing systems find application in many fields, including real time analytics, online machine learning, continuous computation. One powerful such system is Apache Storm [2], a free and open-source platform, able to interoperate with lots of technologies belonging to the Apache ecosystem. Storm is used by many big companies - including Yahoo! and Twitter - for advanced real time distributed computation. Stream processing systems usually run for very long time; therefore, they may be affected by software aging.

Software aging is a phenomenon consisting of the performance degradation or of the increase of the failure rate of a program as it executes [3]. This is usually due to the accumulation of errors that leads the system-internal environment to a state in which such errors are propagated, causing the so-called aging-related failures [4]. Its common causes are memory leaks, data corruption accrual, unreleased file locks, round-off errors accumulation, unterminated threads, file-space fragmentation. Software aging has been demonstrated to affect many complex long-running systems, such as web servers [5], operating systems [7], and even safety-critical systems [8]. Software aging is usually a consequence of software faults, referred to as aging-related bugs – a class of faults may cause failures only after a long period of execution [4].

Software rejuvenation was proposed as a means to prevent or at least delay aging-related failures, hence to mitigate the impact of aging [9] [10]. In its simplest form, rejuvenation involves stopping and subsequently restarting the whole software application or parts of it, in a preventive manner. This allows removing the accrued error conditions, by refreshing software internal state. A number of rejuvenation techniques, at various level of granularity (concerning the entire system or even small parts of it), are now available, including: garbage collection, flushing of kernel system structure, preemptive rollback, re-initialization of data structures, memory defragmentation, micro-reboot, virtual machines-level rejuvenation [3]. Algorithms are also available for the optimal scheduling of rejuvenation, i.e. for the problem of *when* to apply rejuvenation [3].

This paper investigates symptoms and effects of software aging phenomena in the popular stream processing technology Apache Storm. Along with other compatible software, such as Apache Kafka (a distributed publish-subscribe messaging system) and ZooKeeper (a distributed configuration/synchronization system), Storm is widely used to set up ESP infrastructures. The investigation is based on experiments with a workload generator as test application, and measurements are taken to detect aging. The data gathered about memory consumption, throughput and the workload itself are analyzed to discover evidences of software aging afflicting the considered stream processing technology. Besides actually revealing aging phenomena, the experiments allow to spot potential causes of the observed anomalies, attributable to the garbage collection and to memory management. This in turn allows to propose and experiment a software rejuvenation solution.

The rest of the paper is organized as follows. Section 2 provides a description of ESP and of the most popular distributed stream processors. Section 3, describes in more detail the software aging phenomenon and the problems of detection and rejuvenation. Section 4 shows the hardware/software test-bed used for the experiments. Section 5 presents and discusses the experimental results. Finally, conclusions and directions of future work are presented in Section 6.

2. Event Stream Processors

Event stream processors are software platforms capable of manipulating unbounded streams of data, usually fed through sockets or publish/subscribe data distribution systems. Typically, messages are processed as soon as they arrive; parallel computations are achieved by distributing messages among multiple nodes. Messages in a stream can be collected within a temporal window to provide an output that is a function of more messages. The operation on messages include buffering, join, merge and aggregation. Several platforms are currently available; the most popular open-source ones include Tand Storm, Spark Streaming, Samza and Flink, all developed by the Apache Software Foundation.

According to [2], Storm is: "*a free and open source distributed real-time computation system, which makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing. It can be used with any programming language*". Various stream sources (e.g., queuing and databases technologies) can be plugged into Storm. Thanks to the Trident framework [11], it is also able to perform micro-batching operations, treating messages in a stream as batches gathered within temporal windows.

Spark Streaming [12] is not strictly categorized as a stream processor, as it actually performs micro-batch processing, yet it works with unbounded data streams. It does not provide latencies as low as those of Storm, while its performances are comparable to Trident.

As for Samza [13], the main difference with Storm lies in that Samza needs YARN [14]. YARN (Yet Another Resource Negotiator) is a cluster resource manager supporting the separation of the Hadoop Distributed File System (HDFS) from MapReduce, thus granting other system access to HDFS. Samza has a parallelism model which is simpler yet less configurable than Storm. Computation entities in the workflow need to be connected using the Apache Kafka publish/subscribe messaging system (Section 2.3).

Finally, Flink is a general-purpose platform for distributed stream and batch data processing [15], capable of running in standalone mode. It is fully compat-

ible with Hadoop (YARN, HDFS). According to the benchmarking performed by Yahoo!, Flink and Storm show similar performance and latency [16].

2.1. Apache Storm

The architecture of the Apache Storm stream processor [17] is based on the following entities (Fig. 1):

- A *Topology* is a directed acyclic graph, with nodes representing computations and edges data exchanges. Nodes are *spouts* or *bolts*;
- *Tuples* are ordered lists of (untyped) values produced by nodes. Storm needs to know how to serialize values to transfer tuples among nodes;
- *Streams* are unbounded sequences of tuples sent from a node to another. Apart from the very first nodes in a topology, which read from the external data sources, any node can accept more than one stream as input;
- *Spouts* are stream sources; they listen for incoming messages from external sources, and forward them, without performing computation, as their role is solely to emit tuples to the next type of nodes, i.e., the bolts;
- *Bolts* are entities which receive tuples, perform computations, and emit tuples. Tuple transformations include filtering, aggregation, and join;
- *Stream grouping* defines the way (tasks) the tuples are sent among bolts and spouts instances.

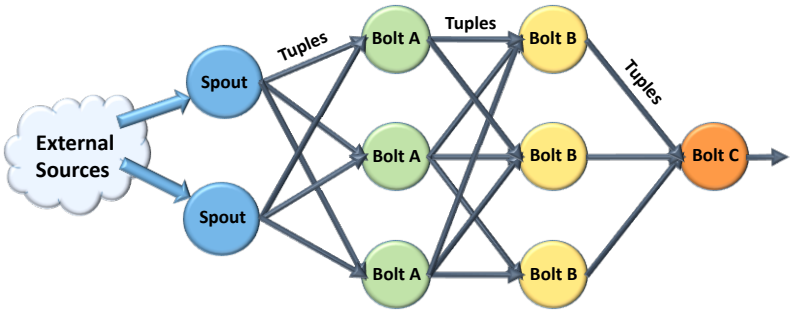

Figure 1: A Storm topology.

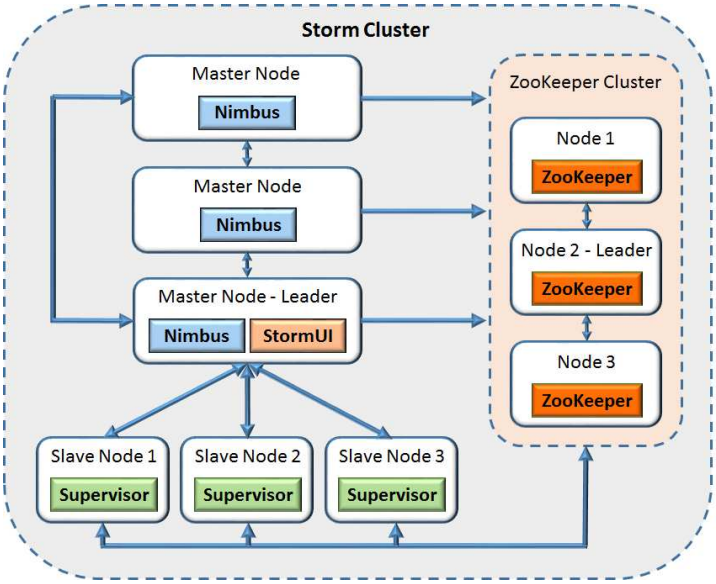

Figure 2: Apache Storm architecture.

2.2. Apache Storm architecture

The Storm architecture (Fig. 2) includes three core components: Nimbus, Supervisor and ZooKeeper:

- *Nimbus* [18] is a daemon which distributes the components of a topology onto computing nodes, assigns tasks to them, and monitors possible failures. It allows also to control the topologies life-cycle, providing means to start/kill/activate/deactivate and re-balance them. The master nodes of a Storm cluster are those running a Nimbus instance. Nimbus is designed to be "fail-fast", meaning that it self-destructs whenever any unexpected situation is encountered. Its state is kept in ZooKeeper or on disk; it is therefore suited for running under supervision, that is, by using a tool capable of restarting it whenever it crashes. In most cases, the Nimbus failure is temporary and it is restarted by the supervisor tool. However, in situations like a master node disk failure, Nimbus fails and becomes unreachable. Under these circumstances, the topologies keep running normally, but no new topology can be submitted. Moreover, the exist-

ing topologies cannot be killed/deactivated/activated, and if a Supervisor fails it cannot be reassigned, resulting in performance degradation or even topology failures. This is the reason why, starting from Storm version 1.0.0, Nimbus can be instantiated on more than one node [18], as shown in Fig. 2. This is possible thanks to ZooKeeper, that includes an election mechanism to select a leader instance. When the Nimbus leader fails, ZooKeeper starts an election to decide which survivor has to take its role.

- *Supervisor* is a fail-fast and stateless daemon managed by the Nimbus leader. It is enabled to instantiate and monitor worker processes responsible for the topologies execution, and to restart failed worker processes. If the worker process continuously fails to reboot and is unable to send heartbeat messages, Nimbus reschedules the worker. Finally, no worker processes are affected by the failure of Nimbus or Supervisor.

- *ZooKeeper* [19] is a framework providing centralized services for maintaining configuration information, naming, distributed synchronization and group membership. ZooKeeper supports coordination of distributed processes through a shared hierarchical name-space, implemented by a standard file system kept in memory, in order to assure high throughput and low latencies. Within Storm, ZooKeeper is used to achieve coordination between Nimbus and Supervisor nodes, and to monitor their states.

The user interface (StormUI) provides a tool to monitor a cluster in production.

A Storm topology is characterized by three main components: *worker processes*, *executors* and *tasks* [20]. Each topology can include several worker processes. A worker process runs one or more components of a topology (i.e., spouts or bolts). Each topology component is assigned one or more executors, which perform data processing tasks. Each computational node can includes one or more worker processes. These three parameters can be configured in order to obtain a defined parallelism degree. Finally, the number of executors and of worker processes can be changed run-time to perform a topology re-balance.

Internal queue-based messaging mechanisms enable communication among executors within a worker process (*intra-worker communication*), as well as among worker processes belonging to the same topology (*inter-worker communication*). Each executor has its own incoming and outgoing queues.

In order to achieve an *inter-topology communication*, external messaging system, such as Kafka or RabbitMQ, must be adopted, as shown in Fig. 3.

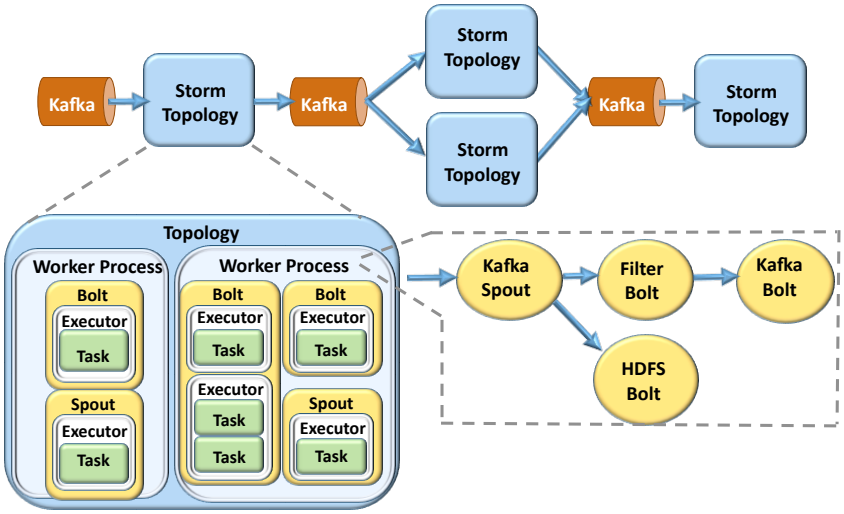

Figure 3: Storm topology communication.

2.3. Apache Kafka

Apache Kafka [21] is a scalable and distributed publish-subscribe messaging platform, used to develop real-time data pipelines and streaming applications. Like most publish-subscribe systems, Kafka manages feeds of messages in topics. A topic is a category (or feed name) of published messages. Topics can be partitioned and replicated across multiple nodes. Each partition is an ordered and immutable sequence of messages that is continually appended to a commit log. A sequential ID number uniquely identifies each message in a partition. The Kafka cluster retains all published messages for a configurable time interval. Messages by a producer are appended to a topic partition in the order they are sent. Consumer see messages in the order they are stored in the log.

3. Software Aging

Software aging is a phenomenon often affecting long-running systems, consisting of the gradual *increase of the failure rate* and/or *decrease of performance* during execution. It is due to the accumulation of erroneous conditions in the system state or to the consumption of resources, such as physical memory [4]. It can be attributed to "elusive" software bugs, i.e., bugs (e.g., memory leaks) that, when triggered, do not immediately cause a software failure, but manifest only after some time, making the system to slowly degrade its performance and eventually fail. These bugs are usually subtle or expensive to expose and remove during testing and debugging, as their manifestation may require a long time [22]. Several types of systems have been shown to suffer from software aging, including: web servers [5, 6], operating systems [7], web applications [23], the Java Virtual Machine [24], data base management systems [25], cloud computing [26] and virtualization environments [27], data centers [28].

Software aging effects can be detected by means of *aging indicators* [4], typically, system variables that can be directly measured and related to aging. Examples are: system resources usage, such as *free physical memory, used swap space, file and process tables size*, or user-perceived performance indicators, like *response time or throughput*. Many studies address the problems of the *detection* and of the *prediction* of the Time To Aging Failure (TTAF), within which a preventive action should be taken. The main strategies to estimate *when* the system will become unavailable due to aging are [3]: *model-based*, where analytic models are used to describe the phenomenon and to estimate the TTAF based on estimated parameters; *measurement-based*, where observed field data are used to infer the real trend of aging that is occurring, and predict the TTAF; *hybrid techniques*, where field data are used to feed analytical models.

Model-based techniques can be applied to a wide range of systems, and they may provide more general findings than measurement-based approaches. However, they can be less effective, since they have some simplifying assumptions, such as the one that the distributions characterizing the system behavior (e.g.,

the time-to-failure distribution) are known. Measurement-based studies forecast software aging based on direct measurements (e.g., on time series analysis and machine learning), and provide empirical data about software aging phenomena. Their advantage is that software aging forecasting can adapt to the current condition of the system (e.g., the current operational profile, which may not have been foreseen before operation), and can accurately predict the occurrence of aging phenomena. On the other hand, measurement-based approaches may be not easily generalizable to other systems, since they exploit aspects related to the nature of the considered system. Hybrid models try to combine both techniques, feeding models online by field data.

As it is often too expensive to fix aging-related bugs during testing, or even infeasible if bugs are within third-party code, libraries or within the OS itself, the typical countermeasure is a runtime proactive fault tolerance technique named *software rejuvenation*. Rejuvenation was defined in [9] as the *preemptive rollback of continuously running applications to prevent failures in the future*. It removes aging effects and prevents aging-related failures, without requiring knowledge of the location of aging-related bugs, or even the very fact of their existence. The objective of rejuvenation is to avoid - or at least postpone - aging-related failures, and reduce the overall downtime and related cost. It has to be applied carefully, assuring that the downtime cost due to rejuvenation (during which an application may be unavailable) is lower than the cost of *unscheduled* downtime due to failures that would occur otherwise.

Rejuvenation strategies aim at determining *when* and *how* to perform rejuvenation. The software aging analysis techniques mentioned for determining the expected TTAF are used for the purpose of *rejuvenation schedule*, i.e. the planning of when to rejuvenate during execution. As for how to rejuvenate, *application-specific actions* can be applied, i.e., techniques that take advantage of special feature of the application domain or architecture, or *application-generic* actions, i.e., techniques that restart the system or its parts and that are not specific to a particular class of systems, classified in [29] as: application restart, OS reboot, virtual machine monitor and restart, and cluster fail-over.

4. Stream processing test application

4.1. Overview

The experimental stream processing application has been designed to be *simple* and *realistic*. Simple means that the application does neither perform complex processing, nor uses any sophisticated external components. This reduces the chance of introducing any aging bug in the test program itself, so that the possible presence of aging could not be attributed to it. Moreover, the application itself has been tested in order to increase the confidence in its correctness. This is a similar approach as in past studies [24]. Realistic means that the application processes a real workload, rather than a synthetic one.

The application (Fig. 4) consists of three processing steps, each implemented by a topology. The communication among different topologies is managed by the publish/subscribe messaging subsystem, provided by Apache Kafka. The design choice of having each processing step implemented by a topology is common among developers of stream processing topologies. In fact, introducing a messaging layer among logically separated topologies allows improving fault tolerance, if the messaging system is enabled to store messages on disk. For the purpose of this study, it also simplifies the detection and diagnosis of a possible aging phenomenon. The next subsections provide a description of each topology.

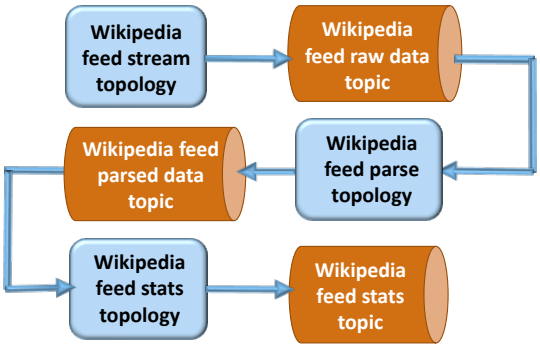

Figure 4: The experimental application.

4.2. Feed stream topology

The first component is the `wikipedia-feed-stream-topology`, shown in Fig. 5. It connects the application to the Wikipedia Internet Relay Chat (IRC) server at the address *irc.wikimedia.org*:6667, joining all IRC channels of incoming messages (e.g., *#en.wikipedia*, *#it.wikipedia*, *#de.wikipedia*, etc.). This operation is done by the `feedStreamerSpout`, which uses an out-of-the-box and IRC library. The messages coming from the IRC server are pushed into a FIFO queue. Storm calls the *nextTuple*() method of the `feedStreamerSpout` to pop a message out of the FIFO queue, and sends it to the `feedWriterKafkaBolt`. This bolt writes the message in a Kafka topic named `wikipedia-feed-raw-data`, so that it can be processed by the next topology.

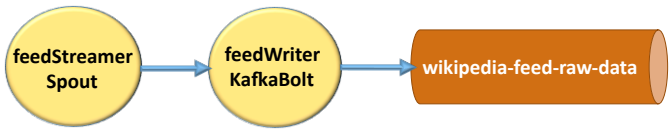

Figure 5: The `wikipedia-feed-stream-topology` component.

As the application has been developed with the main purpose of stress testing Storm, the `wikipedia-feed-stream-topology` can be used to increase the workload, by setting a parameter named message replication factor (MRF). It ensures that a message stored in the FIFO queue is not popped out until it is emitted a number of times equal to MRF.

4.3. Feed parsing topology

The second topology (Fig. 6) processes the messages coming from the IRC server, which include the last modifications made to a Wikipedia page, users login, new accounts creation, abuses, and so on. Such messages provided by the `feedStreamerKafkaSpout` have a precise format, which is parsed by the `feedParserBolt`. The information gathered through this process is then converted to a JSON message and stored in a Kafka topic (named `wikipedia-feed-parsed-data`) by the `parsedFeedWriterKafkaBolt`.

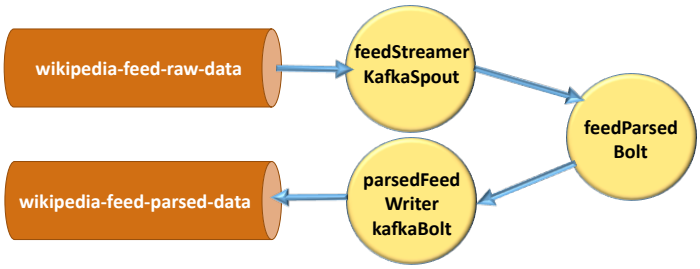

Figure 6: The `wikipedia-feed-parse-topology` component.

4.4. Feed statistics topology

The last topology provides statistics on the parsed messages, computed by gathering them in temporal windows of 10 seconds. This is accomplished by the tumbling window bolt `feed-StatsCalculatorBolt` (Fig. 7). Incoming messages are provided by the Kafka spout `parsedFeedReaderKafka`, which reads the data in the `wikipedia-feed-parsed-data` topic, written by the `wikipedia-feed-parse-topology`. As `feedStatsCalculatorBolt` can be configured with more tasks, incoming messages might be distributed among tasks in a wrong way: if the stream grouping is not chosen appropriately, messages belonging to the same channels (e.g., *#it.wikipedia*) could be assigned to different tasks, thus making the statistics incorrect. For this reason, the component `feedStatsPreprocessorBolt` has been connected to the `feedStatsCalculatorBolt` to provides messages to its tasks in the proper way, by using fields grouping. The last bolt `feedStatsWriterKafkaBolt` writes the statistics in a topic named `wikipedia-feed-stats`.

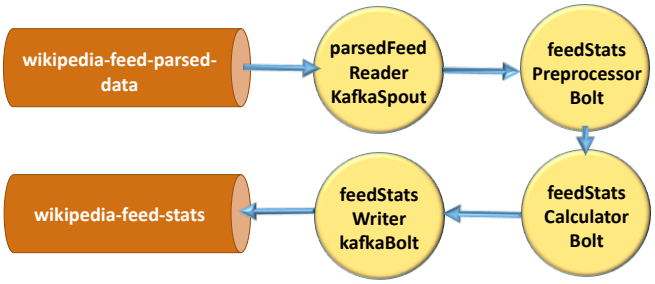

Figure 7: The `wikipedia-feed-stats-topology` component.

5. Experiments

5.1. Testbed

The server used for the experiments is a desktop PC equipped with an Intel Core i7 CPU (2.8 GHz), 4 GB RAM and a 500 GB SATA-3 hard disk. Initially, the system is reset and a fresh installation of the debian-8.4.0-netinst OS (64 bit) is done. The graphical interface and auxiliary software are not activated. In this way, it is ensured that there are no useless services running in background. The following programs are installed: (*i*) Oracle Java Standard Edition Development Kit 8u91; (*ii*) Apache Kafka v.0.10.0.0; (*iii*) Apache ZooKeeper v.3.4.8; (*iv*) Apache Storm v.1.0.1; and (*v*) SSH (for remotely monitoring the system). The measures of interest related to the system resources usage and to the user-perceived performance are collected by *bash* scripts reading from the */proc* filesystem, by using the *vmstat* utility provided by the OS, and by reading from Storm logs details about emitted requests and their responses.

5.2. Experiments, metrics, analysis method

We investigate the following research questions:

- **RQ1**: Does Apache Storm suffer from aging, and, if so, to what extent?
- **RQ2**: Is Apache Storm subject to software aging phenomena under stressful conditions, and, if yes, to what extent?
- **RQ3**: How does Apache Storm manage the exhaustion of available memory under extreme conditions or under its limit (i.e., maximum usable capacity) conditions?

To address them, four long-running experiments have been designed. The first two aim at investigating the possible presence of software aging under a real (unaltered) workload (Experiment 1, Section 5.3) and an amplification of the real workload by a 10x stress factor (Experiment 2, Section 5.4). The second two experiments (Section 5.5) investigate how Storm manages the exhaustion of available memory by considering extreme cases of very high work loads (a 500x

stress factor in Experiment 3) and under the maximum usable capacity (Experiment 4), zooming into anomalies related to the Storm memory management mechanisms. A prototypal rejuvenation action is also implemented and tested. For each experiment, we adopt a measurement-based technique, by gathering data and analyzing the resulting time series. The analysis is performed both at system-level (the macroscopic system resources which age) and at process level (to identify those processes more responsible for resource consumption and user-perceived performance degradation, if any) – thus in the following we distinguish *global-level* and *process-level* analysis.

We point out explicitly that Experiment 1 uses a real workload, as input data come dynamically from the actual Wikipedia IRC server channels, while experiments 2-4 are based on a replication of real data which preserves the pattern of requests, while mimicking the presence of more users.

The main aging indicators we consider regard both the user-perceived performance (in terms of throughput and latency) and the resource depletion in terms of real memory consumption. These are the typical aspects considered in software aging studies [3], and we consider them as *direct* aging indicators[2]. Regarding memory consumption, we consider a summary metric to account for the caching and buffering effect on the amount of memory consumed:

$$Global_{MC} = TM - MemFree - Cached - Buffers \tag{1}$$

where TM is the total memory. The page cache contains a copy of recently accessed files in kernel memory. Since it can get all the free memory not allocated by the kernel or user processes, its memory consumption is quite large and would bias the analysis; therefore, it is subtracted from total memory. Buffers also stores temporary data which can be freed if needed, hence it is subtracted, too. An increasing trend of the above metric over time is useful to detect software aging phenomena impacting memory depletion.

[2]A gradual and monotonic decrease of throughput or increase of latency or of memory consumption over time is considered a software aging phenomenon.

The performance indicator is the *topology throughput* (TT), which is the metric perceived by the user submitting the tasks to the system. It is the percentage of the incoming tuples have been successfully processed by the topology in a certain time interval. It is measured separately for the three topologies. The incoming *request rate* is the *number of emitted tuples per second* taken at the first topology, namely the `wikipedia-feed-stream-topology`. Then, the throughput is measured in all the three topologies of interest.

Finally, the latency (LAT) is measured for each of the three topologies as time taken to start processing a request.

As for process-level indicators, the memory consumption metric considered sums up the resident set size $VmRSS$ - the amount of RAM space in use, for private and shared areas - and the swapped-out space $VmSwap$ used by anonymous private data (shared memory swap usage is not included)[3]:

$$Proc_{MC} = VmRSS + VmSwap. \tag{2}$$

Beside these *direct* aging indicators, we consider further metrics, both at *global level* (Table 1) and at *process level* (Table 2), which are *indirect* indicators, as their trends do not necessarily imply the presence of software aging. They are useful to explain the aging dynamics, and to identify the main contributors to the direct indicators.

The monitored processes are those at the core of the architecture, namely:

- The `Nimbus` and `Supervisor` daemons;
- The `Zookeeper` and `Kafka` processes;
- The `Storm-ui` process, namely, the storm user interface;
- The `Redis` server, which is a key value store used to implement state persistence of bolts across a topology;
- The three worker processes instantiating the `feed-stream`, `feed-parse` and `feed-stats` topologies.

[3]A recent study has shown, with reference to the web caching proxy Squid, that *heap usage* may be a better indicator than the resident set size (RSS) for memory leakage-related aging [30]. Nonertheless, RSS is typically used for detection of aging related to memory leakage.

Table 1: Global-level indicators monitored during tests. Data collected from the */proc* filesystem (CPUIdle), logs of Storm (for TT and LAT), and through the *vmstat* utility.

Global-level software aging indicators	
Indicator	**Description**
CPUIdle	CPU Idle time (seconds per sample)[4]
NReads/NWrites	Number of reads/writes (per sample)
ReadSectors/WrittenSectors	Read/Written sectors (per sample)
ReadTime/WriteTime	Time spent (in ms) reading/writing from the disk (per sample)
ioTime	Time spent (in s) waiting for I/O to complete (per sample)
Swpd	Amount of swapped memory
Swap_in/out	Memory per second swapped in from/out to disk
Block_in/out	Blocks per second received from/sent to a block device
MemFree	Amount of idle memory (KB)
Cached	Memory used as cache (KB)
Buffers	Memory used as buffers (KB)
Interrupts	Interrupts per second, including the clock
ContextSwitches	Context switches per second
$Global_{MC}$	Global memory consumption (KB)
Throughput	Applicative throughout, for the three topologies (%)
Latency	Applicative latency, for the three topologies (ms per sample)

Table 2: Process-level indicators monitored during tests. Data collected from */proc/<PID>/status*, */proc/<PID>/smaps* and */proc/<PID>/io*.

Process-level software aging indicators	
Indicator	**Description**
CPU	Total CPU time: sum of times spent running non-kernel code, kernel code, idle, and waiting for IO
VmRSS	Resident set size: amount of physical memory (private and shared pages) the process is currently using
VmSize	VM size: Amount of virtual memory available to the application (KB)
VmSwap	Swapped-out VM size by anonymous private pages (KB)
PSS	Proportional Set Size: private pages plus shared pages each divided by the n. of sharing processes (KB)
USS	Unique Set Size: private pages (KB)
rchar/wchar	Number of bytes read/written (per sample) from storage It is unaffected by whether or not actual physical disk IO was required (the read might have been satisfied from pagecache)
syscr/syscw	Number of read/write system calls (per sample)
read/write_bytes	Number of real read/written bytes (per sample) from storage
canceled_write_bytes	Number of bytes not written due to page cache truncation (per sample)
$Process_{MC}$	Process memory consumption (KB)

Differently from conventional controlled experiments in software aging studies, we need to consider that the workload is real and variable – only the replication factor is controlled in experiments 2-4. Workload-dependent analysis requires not only to compute the trend of the indicators, but also to relate them to the workload. Indeed, increase of memory consumption or decrease of throughput may well be due to the load increase, in which case the performance degradation is not a symptom of software aging – it is an expected behaviour. However, progressive and continuative depletion of memory and/or decrease of throughput under a constant or decreasing request rate is symptom of aging.

To account for this, we act as follows. We adopt the conventional Mann-Kendall test (MKT) [31] with 95% confidence level to estimate the trends of aging indicators. The analysis is typically conducted by means of observation windows, which define time intervals within which to apply MKT. Let us consider one summary indicator for memory consumption (MC), one for throughput (TT), one for latency (LAT), and the incoming request rate (WL). If the MKT does not detect any trend for at least one of these indicators in the current observation window, the window is expanded to consider more samples. When the MKT succeeds for at least one variable, the window slides over; the new window starts from the first sample just after the previous window.

The effectiveness of the Mann-Kendall test for aging detection has been investigated in [32]: Machida et al. have shown experimentally that MKT suffers from high rates of false positives, so it is possible for it to indicate software aging even where there is no aging. This limitation of MKT can be contrasted by increasing the amount of data considered in the test, at the cost of increasing the time to detect aging. Despite its limit, MKT remains the most widely adopted test to detect aging. However, the choice of the observation window is crucial, and it has to be performed carefully. In our study, the "sufficiently large" initial size for the observation window has been defined based on a set of preliminary tests and a manual inspection results, which suggested setting it as high as 120 minutes, with an expansion factor of 10 minutes.

In a window with at least one trend, the following cases are of interest:

- There is an increasing trend in WL and an increasing trend in MC and/or a decreasing one in TT and/or an increasing trend in LAT. We consider this a non-aging behaviour, since the decrease is somehow "expected" because of the workload increase;

- There is no trend or a decreasing trend in WL and an increasing trend in MC and/or a decreasing trend in TT and/or an increasing trend in LAT. We consider this behaviour a potential aging phenomenon, as the behaviour is not "expected" in that window, because of the increase of the

workload. The large minimum size of the window minimizes the chance that this behaviour is due to the "history" of previous windows. Of course, if two or all of the three indicators (MC, TT, LAT) have an aging trend, the likelihood that there is an actual aging occurring is higher.

All the other cases do not give clues about possible aging (i.e., if neither MC nor TT nor LAT have aging trends, then we conclude that there is no aging in that window). We count how many times, in a given experiment, the above cases happen. This approach allows investigating how much the workload affects the memory growth, the topologies throughput and their latency. Thus, for instance, suppose that the MKT test is applied over 50 time windows across all the time series. Consider the case of the MC aging indicator. if the MKT applied to the MC series notifies an increasing trend AND the MKT applied to the input WL series, in the same time windows, notifies a decreasing or no trend, we count a potential aging behaviour. If, in the mentioned example, this happens 10 times, then we note a percentage of 20% for the MC indicator, meaning that in 20% of cases there has been a potential aging phenomenon related to the memory consumption.

5.3. Experiment 1 (RQ1)

5.3.1. Global-level analysis

The first experiment addresses research question RQ1; it consists in a relatively long-running test of 72 hours. The message replication factor is set to 1 – i.e., the workload is left unmodified - and the sampling period is 1 minute. The workload occurring during the experimental period is showed in Fig. 8.

Figures 9-15 show the global memory consumption, the throughput and latency of the three involved topologies. A workload-independent analysis – i.e., looking for a global trend over the entire time series – highlights a trend on memory consumption (2.28 KB/minute), and no trend in the throughput indicators (none of the trends is significant at 95%, or it is 0 with confidence greater than 95%). At latency level, there is a very slight trend for the `feed-stream` topology

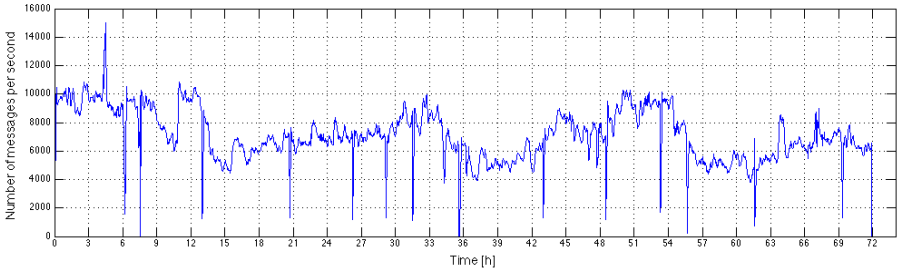

Figure 8: The input workload, test 1

(amounting to 1.58e-09 ms increase per request) and for the `feed-parse` topology (6.12e-10 ms increase per request). The per process analysis will highlight if this is somehow associated with the memory consumption at process level.

Table 3 lists the results of the workload-dependent correspondence analysis: it shows the percentage of time the aging indicators have a trend concordant or discordant with the workload trend. The highest percentage is in the latency indicator of the `feedstats` topology, which in 36.11% of the cases shows a trend together with a decreasing (or no) trend of the WL. The global trend of the entire time series is negative (and very small); thus, despite there are windows where latency increases unexpectedly, the phenomenon is to be considered negligible (indeed, there is no impact on throughput). The other indicator with a relatively high percentage is for memory consumption, whose trend is increasing in 38.89% of the observation windows, along with a non-increasing WL trend in 27.78% of the cases. The global trend for MC is 2.28 KB per minute.

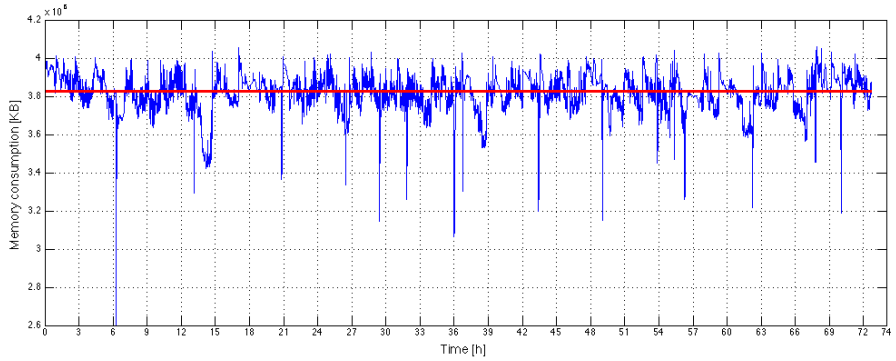

Figure 9: Global memory consumption, test 1

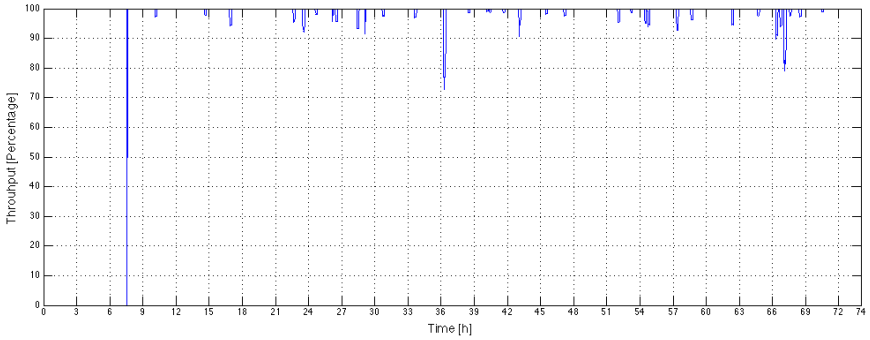

Figure 10: Throughput of `wikipedia-feed-stream-topology`, test 1. (Y-axis scale: 0-100).

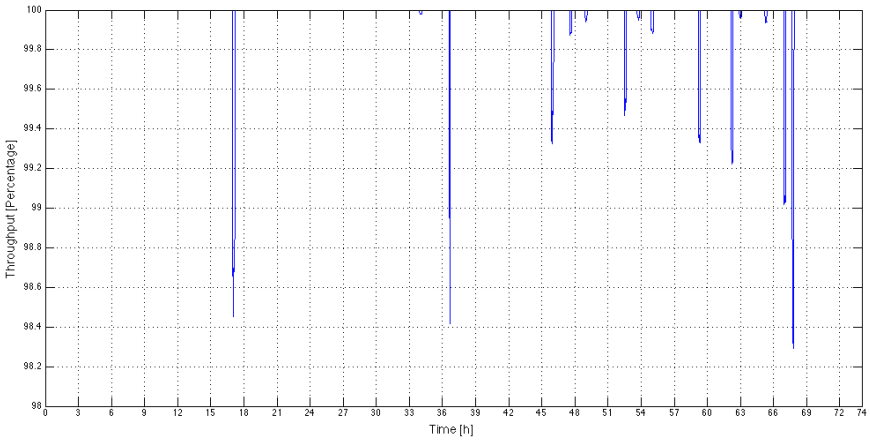

Figure 11: Throughput of `wikipedia-feed-parse-topology`, test 1 (Y-axis scale: 98-100).

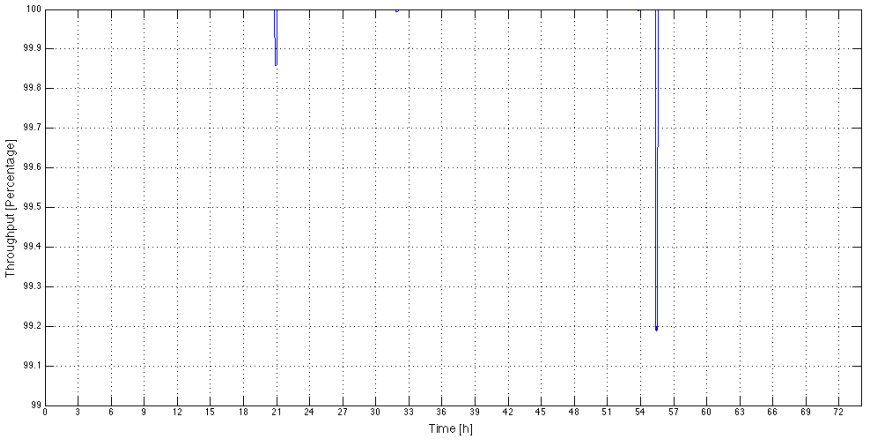

Figure 12: Throughput of `wikipedia-feed-stats-topology`, test 1 (Y-axis scale: 99-100).

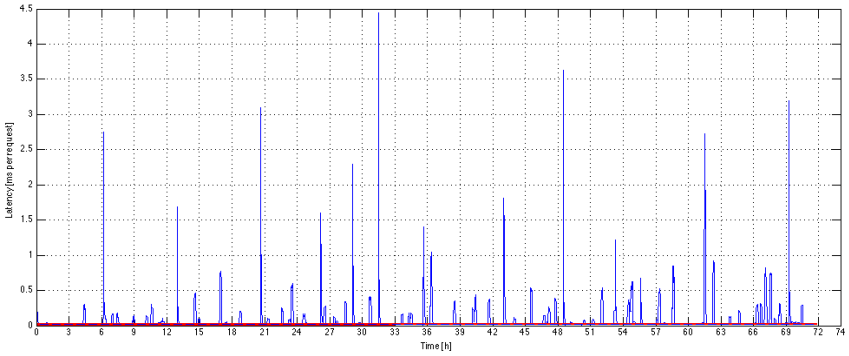

Figure 13: Latency per request of `wikipedia-feed-stream-topology`, test 1.

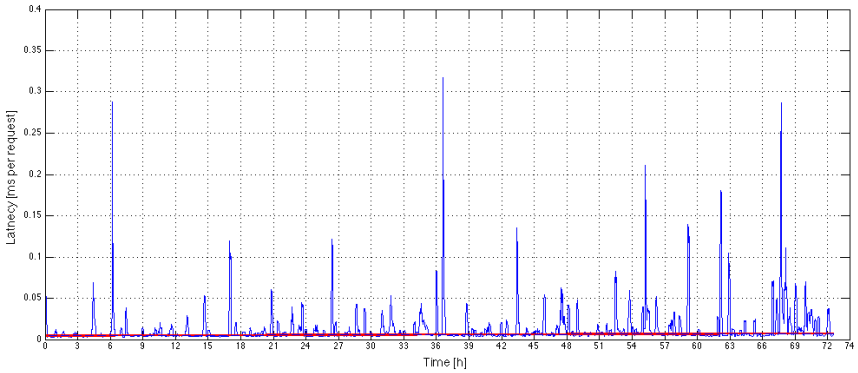

Figure 14: Latency per request of `wikipedia-feed-parse-topology`, test 1.

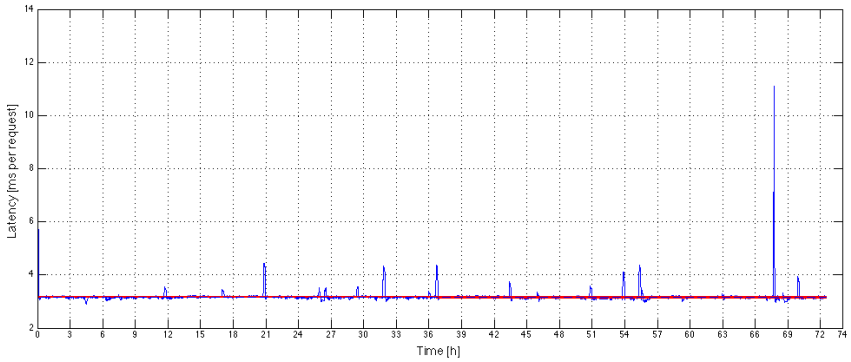

Figure 15: Latency per request of `wikipedia-feed-stats-topology`, test 1.

Table 4 lists the global trend for each remaining *indirect* indicator, if significant with *p-value* < 0.05 (it is considered zero otherwise). There is a slight trend in the disk writing operations, whereas the CPU- and memory-related indicators confirm the stable behaviour of Storm under the considered load.

Table 3: Correspondence analysis between workload and aging indicators, test 1. Percentage of occurrence of trends in the aging indicator (row) in correspondence to the WL trend (column)

WL Trend / *Aging Trend*	**Increasing, decreasing or no WL trend** (Independent of WL)	**No or decreasing WL trend**
Increasing MC	38.89%	27.78%
Decreasing TT	13.89% (`stream` topology) 8.33% (`parse` topology) 5.55 % (`stats` topology)	8.33% (`stream` topology) 2.78% (`parse` topology) 5.55% (`stats` topology)
Increasing LAT	25% (`stream` topology) 22.22% (`parse` topology) 41.67% (`stats` topology)	25% (`stream` topology) 16.67% (`parse` topology) 36.11% (`stats` topology)
Increasing MC AND Decreasing TT	2.78% 0.00% 0.00%	2.78% 0.00% 0.00%
Increasing MC AND LAT	5.56% 2.78% 22.22%	5.56% 0.00% 19.44%
Decreasing TT AND Increasing LAT	0.00% 5.56% 2.78%	0.00% 2.78% 2.78%
ALL: Increasing MC AND Decreasing TT AND Increasing LAT	0.00% 0.00% 0.00%	0.00% 0.00% 0.00%

Table 4: Global trend and WL-aging, test 1.

Indicator	**Trend**
Global-level Aging Indicators (Indicator's *um* per minute – e.g., KB/min for $Global_{MC}$)	
Direct indicators	
$Global_{MC}$	2.28e+0
TT *stream*	0.0e+0
TT *parse*	0.0e+0
TT *stats*	0.0e+0
LAT *stream*	1.58e-9
LAT *parse*	6.12e-10
LAT *stats*	-3.71e-9
Indirect indicators	
CPUIdle	2.4e-3
NReads/NWrites	0.0e+0/9.33e-3
ReadSectors/WrittenSectors	-2.41e+0/4-44e+0
ReadTime/WriteTime	1.17e+0/3.93e+0
ioTime	4.38e-4
Swpd	0.0e+0
Swap_in/out	-9.14e-4/-2.5e-3
Block_in/out	1.10e-3/5.63e-2
MemFree	9.19e-1
Cached	-2.07e+0
Buffers	3.25e-1
Interrupts	2.78e-2
ContextSwitches	2-84e-2

5.3.2. Process-level analysis

We analyze now the direct aging indicators for each process over the entire time series. Then, the relation of the process aging indicator $Proc_{MC}$ with the workload is considered. Table 5 summarizes the results.

Table 5: Global trend (indicator *um* per minute) of process-level indicator, and WL-MC correspondences. Trends not significant at $p - value$ <0.05 are set to 0.0e+0.

	Processes				
Indicator	`Nimbus`	`Superv`	`Zook`	`Kafka`	`Storm-ui`
CPU	-5.74e-4	0.0e+0	2.73e-4	-3.08e-2	0.0e+0
$VmSize$	0.0e+0	0.0e+0	0.0e+0	-7.45e+0	0.0e+0
$VmRSS$	1.60e+0	-1.03e+1	-7.47e+0	-2.49e+1	-1.78E+1
$VmSwap$	2.23e+1	1.27e+1	8.30e+0	3.27e+1	1.61e+1
PSS	0.0e+0	-1.06e+1	-7.59e+0	-2.49e+1	-1.70e+1
USS	0.0e+0	-1.07e+1	-7.67e+0	-2.50e+1	-1.72e+1
$rchar$	-1.65e-1	-1.66e+1	2.73e+0	-3.24e+4	0.0e+0
$wchar$	6.35e-1	0.0e+0	2.45e+0	-3.23e+4	5.59e-2
$syscr$	3.7e-3	0.0e+0	2.5e-3	-1.73e+1	0.0e+0
$syscw$	5.6e-3	0.0e+0	0.0e+0	-1.10e+1	0.0e+0
$readB$	1.70e+0	-2.37e+0	0.0e+0	0.0e+0	-5.54e+0
$writeB$	3.69e+0	0.0e+0	0.0e+0	-2.28e+4	0.0e+0
$cancWriteB$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	0.0e+0
$Proc_{MC}$	**4.01e+0**	**1.17e+0**	**1.09e+0**	**2.09e+0**	**-1.46e+0**
% of time slots where WL is not increasing and $Proc_{MC}$ is increasing					
$Proc_{MC}$-WL (%slots)	**30.56%**	**16.67%**	**16.67%**	**33.33%**	**27.78%**

Indicator	`Redis`	`stream-top`	`parse-top`	`stats-top`	
CPU	0.0e+0	5.87e-2	-1.5e-1	3.17e-2	
$VmSize$	0.0e+0	0.0e+0	1.80e-1	2.08e+0	
$VmRSS$	0.0e+0	1.10e+1	2.36e+1	-1.11e+1	
$VmSwap$	0.0e+0	0.0e+0	1.31e+1	1.68e+1	
PSS	0.0e+0	1.07e+1	2.36e+1	-1.12e+1	
USS	0.0e+0	1.08e+1	2.36e+1	-1.14e+1	
$rchar$	5.17e-1	-3.02e+1	-8.61e+3	0.0e+0	
$wchar$	0.0e+0	-4.8e+3	-1.32e+4	2.96e+0	
$syscr$	0.0e+0	-7.37e+0	-1.09e+1	0.0e+0	
$syscw$	0.0e+0	-6.72e+0	-3.67e+0	0.0e+0	
$readB$	0.0e+0	-2.29e+0	0.0e+0	-2.51e+1	
$writeB$	0.0e+0	-2.92e+0	-5.48e+0	-2.52e+1	
$cancWriteB$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	
$Proc_{MC}$	**0.0e+0**	**1.26e+1**	**3.52e+1**	**4.47e+0**	
% of time slots where WL is not increasing and $Proc_{MC}$ is increasing					
$Proc_{MC}$-WL (%slots)	**0.0%**	**52.77%**	**25%**	**38.89%**	

It is worth noting the relevant contribution in absolute terms of the `feed-stream` and `feed-parse` topologies to the memory consumption (these had also the trend on Latency). Considering the correspondence with a non-increasing workload, the main processes are the above two topologies, followed by core components *Kafka* and *Nimbus*. These trends are however insufficient to cause the entire system to age in a noticeable way, as highlighted by the analysis in the previous Section.

5.4. Experiment 2 (RQ2)

The second experiment addresses research question RQ2, investigating if the aging behaviour changes under a workload by an order of magnitude heavier. The replication factor is set to 10.

5.4.1. Global-level analysis

The workload during the experimental period is shown in Fig. 16.

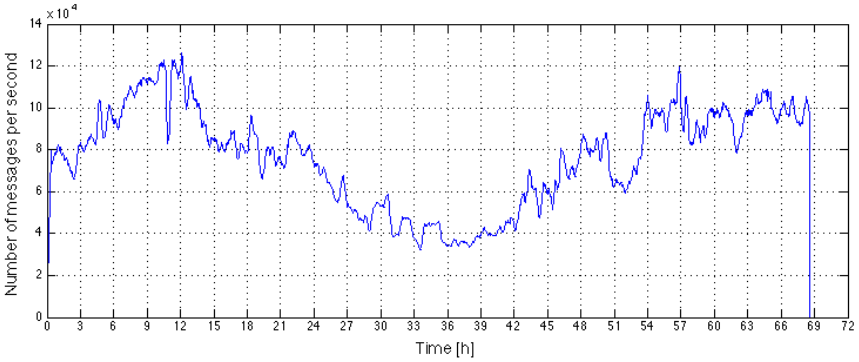

Figure 16: The input workload, test 2.

Figures 17-23 show the global memory consumption, the throughput and latency of the three topologies. A workload-independent analysis – of a global trend over the entire time series - highlights a remarkable trend in the memory consumption, amounting to an increase of 451 KB per minute compared to the previous case; the throughput, however, is still unaffected (namely, Storm manages to serve incoming requests), but the latency of the responses has a positive global trend amounting to an increase of 11 ms per request for the last output topology, `wikipedia-feed-stats-topology` (first part of Table 7).

Table 6 lists the results of the workload-dependent correspondence analysis: it shows the percentage of times the aging indicators have a concordant or discordant trend with the input workload trend (i.e., the request rate).

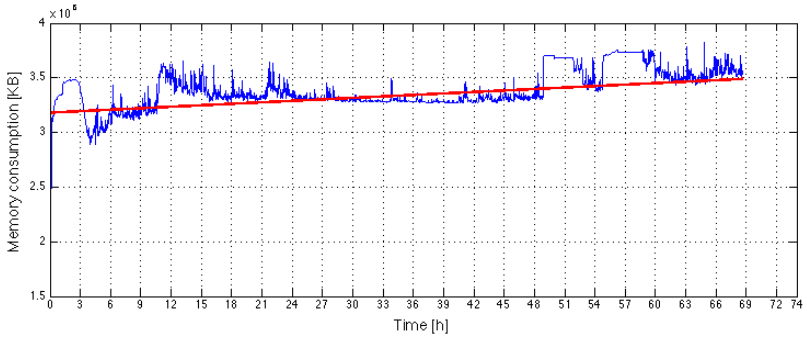

Figure 17: Global memory consumption, test 2.

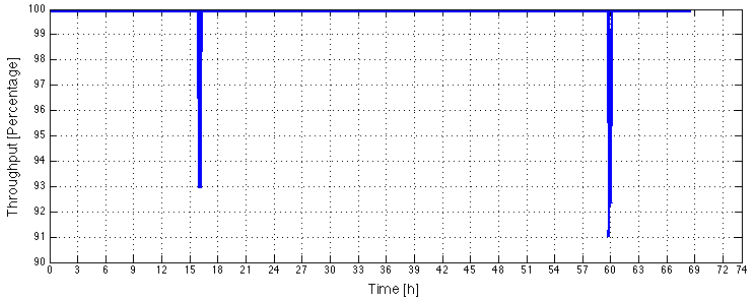

Figure 18: Throughput of `wikipedia-feed-stream-topology`. (Y-axis scale: 90-100.)

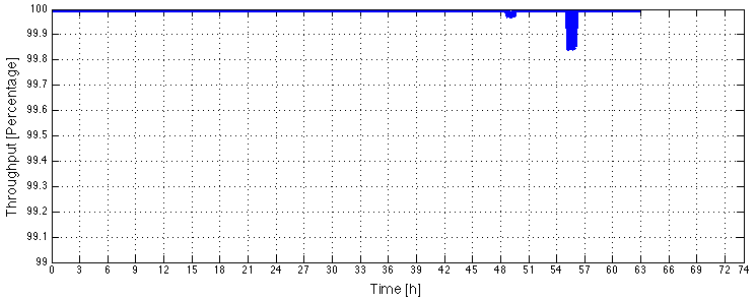

Figure 19: Throughput of `wikipedia-feed-parse-topology`. (Y-axis scale: 99-100.)

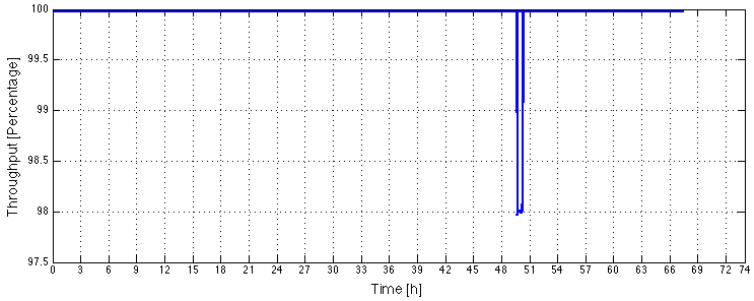

Figure 20: Throughput of `wikipedia-feed-stats-topology`. (Y-axis scale: 97.5-100.)

The results show that there is not a high percentage of observation windows where the aging indicators show aging trends. This means that the trends have a decreasing/increasing pattern (with a pattern that, visually, seems to be the opposite one of the workload), but, considering the entire time series, the overall trends are increasing. Moreover, in the windows where such an aging trend is present (visually, from hour 15 to 39), the WL has in most cases a non-increasing trend (namely, what we defined an aging situation).

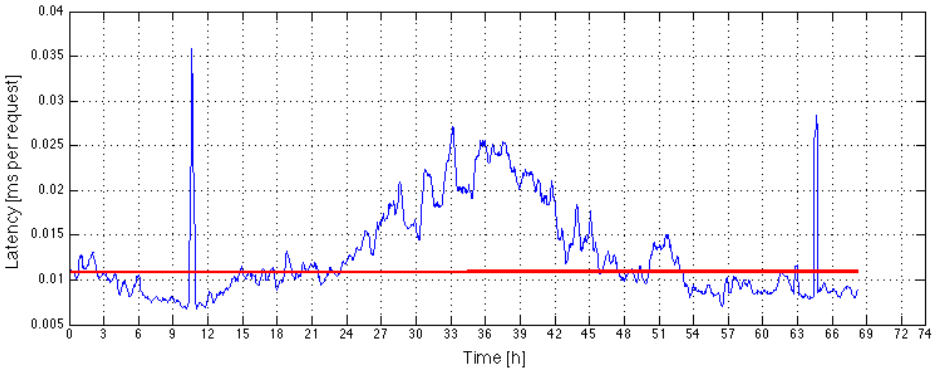

Figure 21: Latency per request of `wikipedia-feed-stream-topology`, test 2.

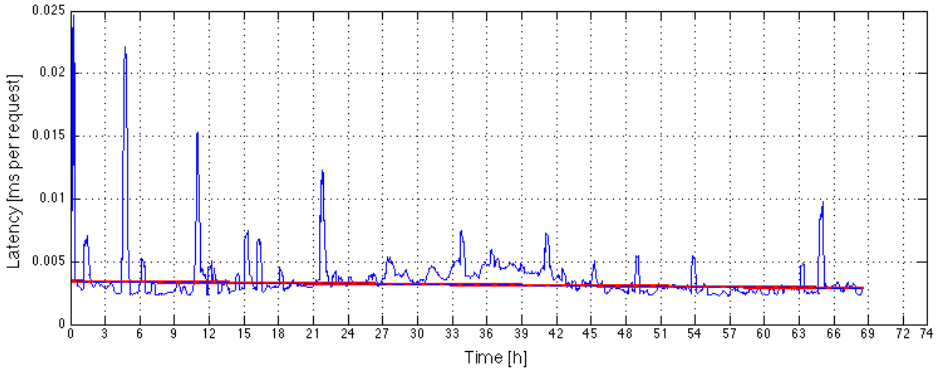

Figure 22: Latency per request of `wikipedia-feed-parse-topology`, test 2.

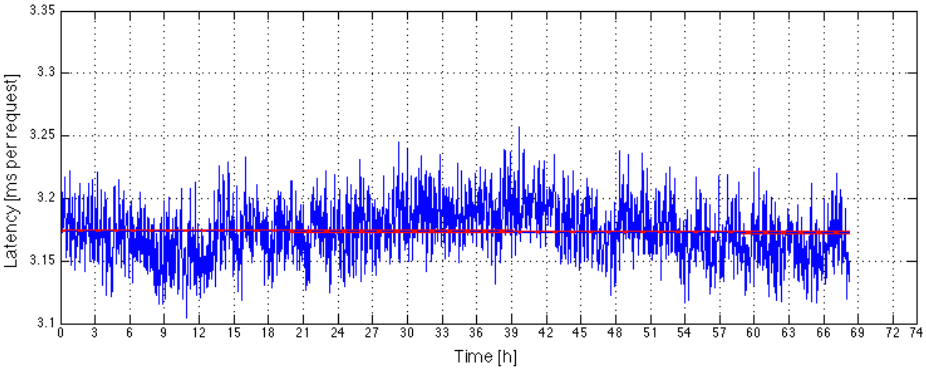

Figure 23: Latency per request of `wikipedia-feed-stats-topology`, test 2.

Table 6: Correspondence analysis between workload and aging indicators, test 2. Percentage of occurrence of trends in the aging indicator (row) in correspondence to the WL trend (column)

WL Trend / *Aging Trend*	**Increasing, decreasing or no WL trend** (Independent of WL)	**No or decreasing WL trend**
Increasing MC	27.78 %	16.67 %
Decreasing TT	5.55% (`stream` topology) 0.0% (`parse` topology) 0.0 % (`stats` topology)	5.55% (`stream` topology) 0.0% (`parse` topology) 0.0% (`stats` topology)
Increasing LAT	27.78% (`stream` topology) 38.89 % (`parse` topology) 22.22% (`stats` topology)	27.78% (`stream` topology) 27.78% (`parse` topology) 16.67% (`stats` topology)
Increasing MC AND Decreasing TT	2.78% 0.0% 0.0 %	2.78% 0.0% 0.0%
Increasing MC AND LAT	2.78% 2.78 % 5.56 %	2.78% 0.0 % 5.55%
Decreasing TT AND Increasing LAT	0.0% 0.0% 0.0%	0.0% 0.0% 0.0 %
ALL: Increasing MC AND Decreasing TT AND Increasing LAT	0.0% 0.0% 0.0%	0.0% 0.0 % 0.0%

Table 7 reports, for both *direct* and *indirect* indicators, the global trend, if it is significant with *p-value* < 0.05 (it is considered equal to zero otherwise). The indicators highlight a positive significant trends on disk activity (writing activity), as well as the contribution to memory trend given by the cache and buffers. There seems to be no stress at CPU level.

Table 7: Global trend and WL-aging, test 2

Indicator	**Trend**
Global-level Aging Indicators (Indicator's *um* per minute – e.g., KB/min for $Global_{MC}$)	
Direct indicators	
$Global_{MC}$	4.51e+3
TT *stream*	0.0e+0
TT *parse*	0.0e+0
TT *stats*	0.0e+0
LAT *stream*	0.0e+0
LAT *parse*	-2.68e-7
LAT *stats*	1.11e+1
Indirect indicators	
CPUIdle	1.30e-3
NReads/NWrites	0.0e+0/3.20e-3
ReadSectors/WrittenSectors	0.0e+0/2.27e-1
ReadTime/WriteTime	0.0e+0/9.92e-2
ioTime	0.0e+0
Swpd	0.0e+0
Swap_in/out	0.0e+0/2.61e-3
Block_in/out	-8.98e-4/0.0e+0
MemFree	-2.58e+2
Cached	8.26e+1
Buffers	9.50e+1
Interrupts	-1.30e-1
ContextSwitches	-1.31e-1

5.4.2. Process-level analysis

Table 8 summarizes the results at process level, considering both the aging indicators over the entire time series, and then the relation of the process aging indicator $Proc_{MC}$ with the workload.

Table 8: Global trend (Indicator's *um* per minute) of process-level indicator, and WL-MC correspondences. Trends not significant at $p - value$¡0.05 are set to 0.0e+0.

	Processes				
Indicator	`Nimbus`	`Superv`	`Zook`	`Kafka`	`Storm-ui`
CPU	-5.1e-3	0.0e+0	0.0e+0	-1.84e-2	-6.43e-4
$VmSize$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	1.97e+0
$VmRSS$	1.87e+2	-1.74e+0	-3.01e+0	-6.22e+1	-4.02e+0
$VmSwap$	0.0e+0	0.0e+0	0.0e+0	7.18e+1	0.0e+0
PSS	1.90e+2	2.70e-1	1.64e-2	-6.03e+1	-1.14e+0
USS	1.89e+2	-1.31e-1	-5.62e-1	-6.04e+1	-1.85e+0
$rchar$	2.57e+0	0.0e+0	9.43e+0	0.0e+0	0.0e+0
$wchar$	0.0e+0	1.11e-2	12.14e+0	0.0e+0	3.62e-2
$syscr$	0.0e+0	0.0e+0	1.50e-3	0.0e+0	0.0e+0
$syscw$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	0.0e+0
$readB$	0.0e+0	0.0e+0	0.0e+0	-1.23e+3	0.0e+0
$writeB$	-2.21e+0	0.0e+0	8.92e+0	0.0e+0	0.0e+0
$cancWriteB$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	0.0e+0
$Proc_{MC}$	**1.91e+2**	**-1.74e+0**	**-2.36e+0**	**3.14e+0**	**-4.02e+0**
% of time slots where WL is not increasing and $Proc_{MC}$ is increasing					
$Proc_{MC}$-WL (%slots)	**47.06%**	**17.65%**	**27.78%**	**50.00%**	**22.22%**

Indicator	`Redis`	`stream-top`	`parse-top`	`stats-top`	
CPU	0.0e+0	-4.85e-2	-2.31e-1	1.27e-1	
$VmSize$	0.0e+0	4.98e-1	0.0e+0	0.0e+0	
$VmRSS$	-3.74e-1	3.11e+0	-3.29e+1	3.16e+1	
$VmSwap$	0.0e+0	0.0e+0	2.60e-3	0.0e+0	
PSS	0.0e+0	3.45e+0	-3.20e+1	4.59e+0	
USS	-1.23e-2	3.02e+0	-3-23e+1	4.00e+0	
$rchar$	1.22e+0	0.0e+0	0.0e+0	-1.30e-3	
$wchar$	0.0e+0	0.0e+0	0.0e+0	6.22e+0	
$syscr$	0.0e+0	0.0e+0	0.0e+0	-3.73e-1	
$syscw$	0.0e+0	0.0e+0	0.0e+0	-1.2e-3	
$readB$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	
$writeB$	0.0e+0	-2.44e+0	0.0e+0	0.0e+0	
$cancWriteB$	0.0e+0	0.0e+0	0.0e+0	0.0e+0	
$Proc_{MC}$	**-3.53e-1**	**3.12e+0**	**-3.29e+1**	**3.16e+1**	
% of time slots where WL is not increasing and $Proc_{MC}$ is increasing					
$Proc_{MC}$-WL (%slots)	**0.00%**	**50.00%**	**33.33%**	**88.89%**	

In absolute terms, the processes mainly contributing to the aging trend are `Nimbus`, where an increasing amount of 191 KB per minute are consumed, and the last topology, `wikipedia-feed-stats-topology`. This is in line with the latency result, suggesting the `wikipedia-feed-parse-topology` as a bottleneck. If we consider the workload-dependent analysis, the percentages of windows where there is an aging trend along with a non-increasing workload of

`wikipedia-feed-parse-topology` is as high as the 88.89%. This means that almost always the memory consumption of `wikipedia-feedparse-topology` increases and the WL does not increase. In conjunction with the latency trend in the same process, this reveals an issue in that process. Secondarily, it is worth noting the role of `Nimbus`, with a percentage of 47.06%, and again `Kafka`, like in the previous test, where the 50% of times there is an aging behaviour.

5.5. Experiments 3 and 4 (RQ3)

The following two experiments refer to research question RQ3; they investigate how Storm manages the exhaustion of available memory by considering extreme cases of very high workloads (Experiment 3) and under the maximum usable capacity (Experiment 4). We will look at specific anomalies related to the Storm memory management mechanisms in such situations. A prototypal rejuvenation action is also implemented and tested.

In Experiment 3, we set the message replication factor to 500, to emulate a highly stressful condition. As Fig. 24 shows, the `wikipedia-feed-stream-topology` memory consumption indicator is affected by several drops, corresponding to the times when the topology is reset by Storm. Such a procedure is triggered by Storm each time Nimbus does not receive the worker process heartbeat within a predefined timeout interval (30 seconds).

At first glance, all these resets happen after the memory consumption grows beyond a (variable) threshold. Moreover, in many cases there are sudden increments of the WL, and the memory consumption grows much faster.

By analyzing topology logs, it is possible to notice the presence of many Full GC operations preceding the topology reset. This is because the workload is too heavy for the server memory resources, and the garbage collector has to continuously free the memory. When a Full GC begins, all the other threads are stopped, so that the operation can be completed as soon as possible. Depending on the size of the memory and on its level of fragmentation, this procedure takes considerable time (up to minutes). Besides stopping the topology processing, this behavior prevents the heartbeat thread from sending its signals to Storm.

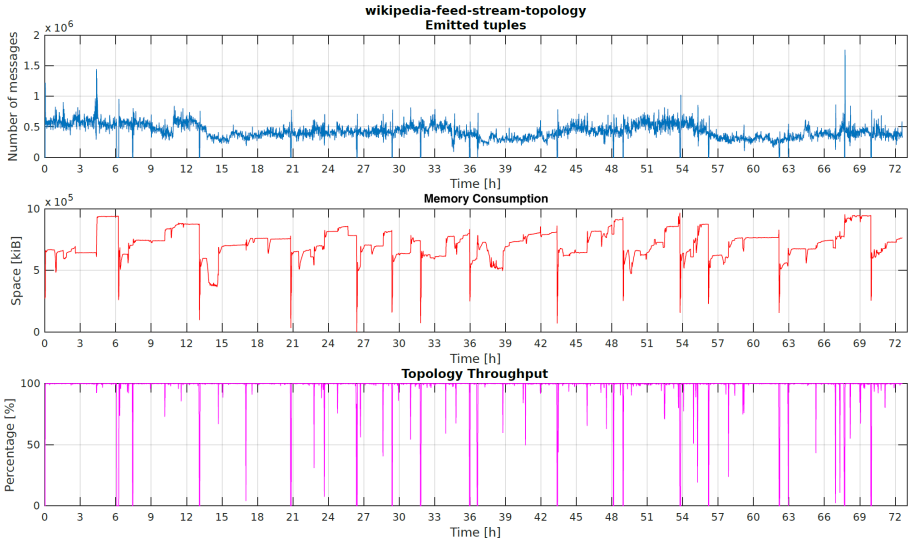

Figure 24: Performance drops of the `wikipedia-feed-stream-topology`.

This is the reason why Storm kills and restarts the topology - causing a high number of lost tuples. The stream feeding topology takes minutes to establish a new connection to all IRC wikipedia server channels. Fig. 25 shows that the topology downtime is about 3-4 minutes. In the meantime, messages are lost.

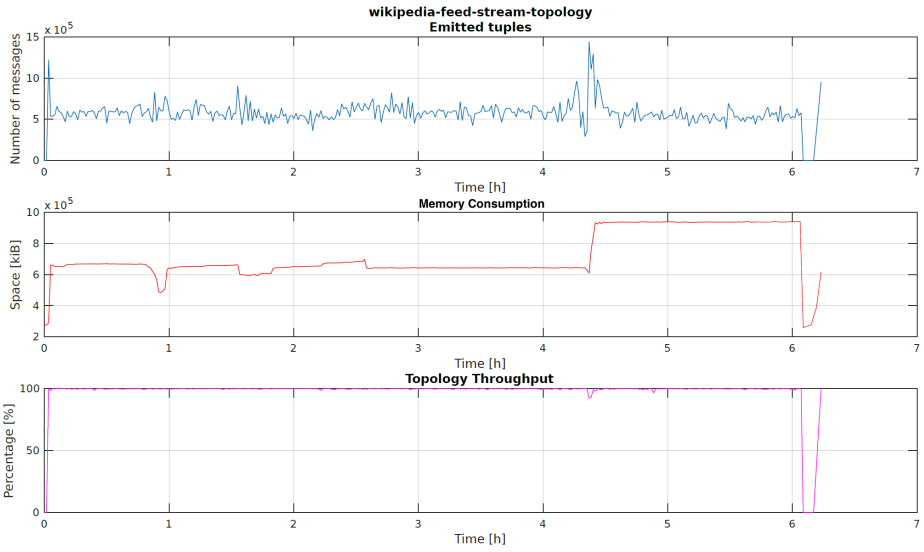

Figure 25: Run of the `wikipedia-feed-stream-topology` before its first reset by Storm.

Several strategies can avoid this undesirable behavior. An obvious one is to provide more memory. However, cleaning large memory areas can take a long time, and the solution may not be adequate. An alternative is to increase the heartbeat timeout value. But Storm would take longer to detect the failure of a topology, resulting in unnecessary downtime. A further solution is to dynamically change the timeout value based on both the memory consumed and the workload. This is however not practicable, because the timeout value is set when Storm is launched – a modification of its source code would be required.

We opted for the replication of the stream feeding topology, to replace it just before it is reset by Storm. The approach consists of simply monitoring the memory consumption of the topology, checking if it exceed a threshold. Since a restarted topology takes about 3 minutes to start emitting tuples again, it is convenient to trigger this failover procedure as soon as a threshold is exceeded. The old topology must be killed just before the new one starts to emit tuples, so that few messages are lost and no duplicates are produced. A threshold value is given by the mean (about 700 MB) of the memory time series of Fig. 24. The procedure is implemented by a '*restarting agent*', which periodically monitors the topology memory consumption. Fig. 26 shows the difference between a Storm topology restart and one made by our restarting agent. The former takes a little more than 2 minutes to complete, while the latter takes 30 seconds.

Fig. 27 shows the behavior of `wikipedia-feed-parse-topology`, which is reset by Storm in three different occasions. These restarts are much faster than the `wikipedia-feed-stream-topology`, as the parsing topology just connects to Kafka to start emitting tuples. Considering that Kafka ensures the messages persistence, it may be justified to let Storm decide when a reset is needed. In this case, the restarting agent could lead to more frequent (and useless) resets.

The `wikipedia-feed-stats-topology` is reset just once, after it spent a long time with its memory at the maximum value (Fig. 28). Once again, the reset is caused by the heartbeat thread, which does not send signals to Nimbus because of highly frequent Full GCs that overload the system. This topology turned out to be the most robust one against heavy workloads.

Figure 26: Comparison between a Storm-made reset and the restarting agent approach.

Experiment 4 consists of additional tests to identify the message replication factor that does not overload the topologies. More memory (1 GB instead of 800 MB) is given to each topology, in order to prevent situations like those illustrated above. Consistently, we do not observe any topology reset. However, other malfunctions regarding both the `wikipedia-feed-stats-` and the `wikipedia-feed-parse-` topologies came out. In particular, the test shows sudden breaks of tuples processing, which went undetected by Storm.

Fig. 29 suggests that memory exhaustion is the possible cause of the anomaly. The GC log files reveal that a Full GC is done a few seconds before the topology stops working. Moreover, many Full GCs are done (every 30 seconds, approximately), so, once again, the memory given to this topology is not enough. However, the seriousness of the anomaly is because the problem causes only the crash of the executors, while the heartbeat thread is still working – so it is a silent failure. This prevents Storm from revealing the issue and performing a reset – a highly undesirable behavior. To prove that this behavior is consistent over time, the topology is manually reset; after about the same time (90 minutes), the topology stops processing tuples and Storm does not detect it.

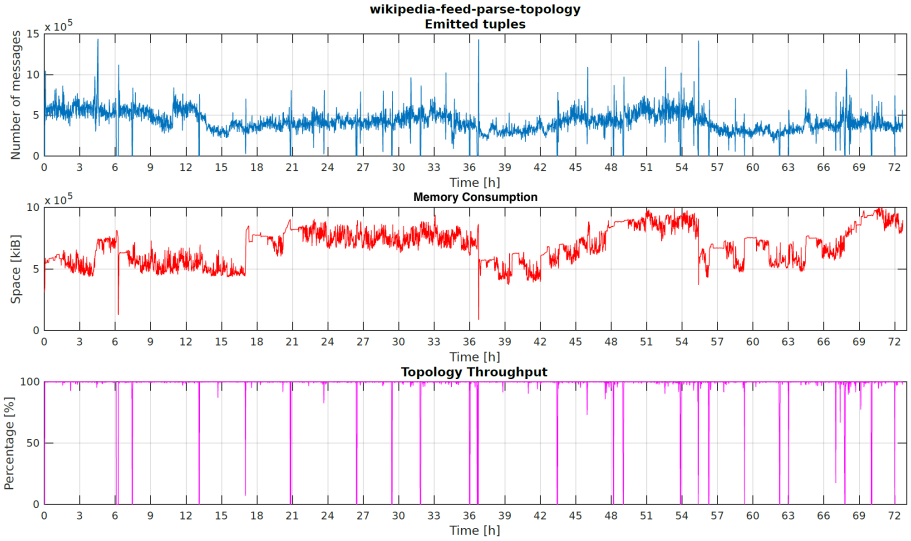

Figure 27: Emitted tuples, memory consumption and throughput of the `wikipedia-feed-parse-topology`.

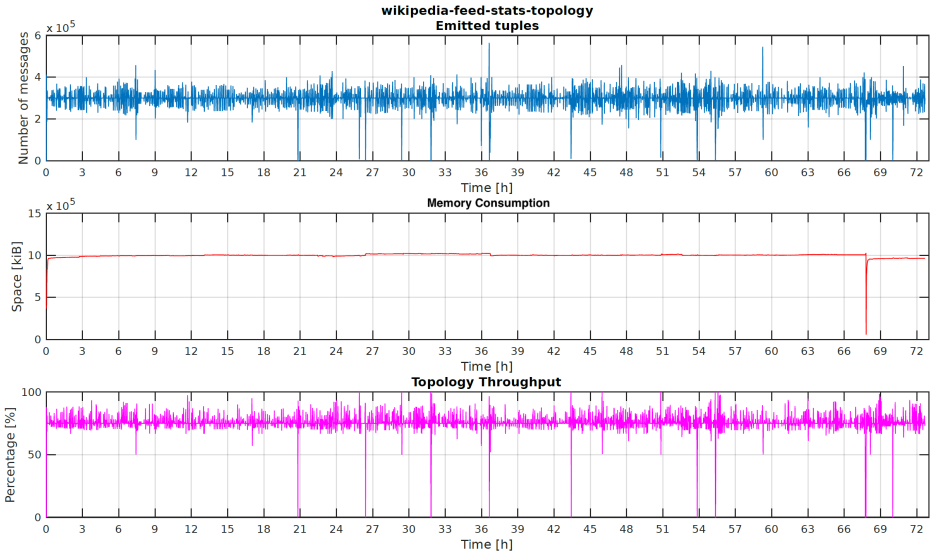

Figure 28: Emitted tuples, memory consumption and throughput of the `wikipedia-feed-stats-topology`.

The `feed-stats` topology exhibits a similar behavior (Fig. 30). After about 6 hours, it suddenly stops to process tuples. The GC log files contain similar information as for the `feed-parse` topology. This might be due to the Storm internal communication mechanisms: they are based on external libraries, which

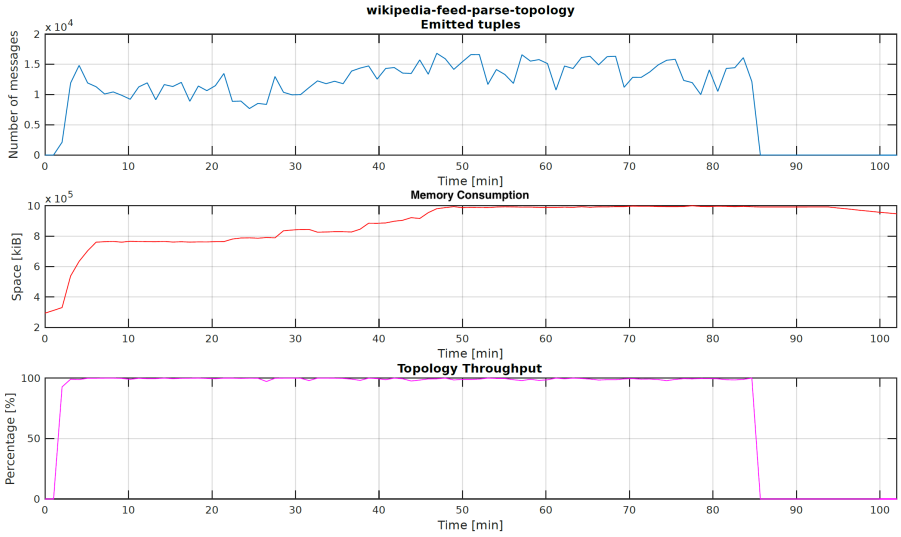

Figure 29: Tuple processing stopped with the heartbeat enabled.

use the Java Unsafe API, which in turn allows to allocate memory without invoking the GC (as per default). If confirmed, this hypothesis – to be further investigated - would mean that the GC collects the unsafely allocated memory, thus blocking the threads waiting for objects stored in that particular area.

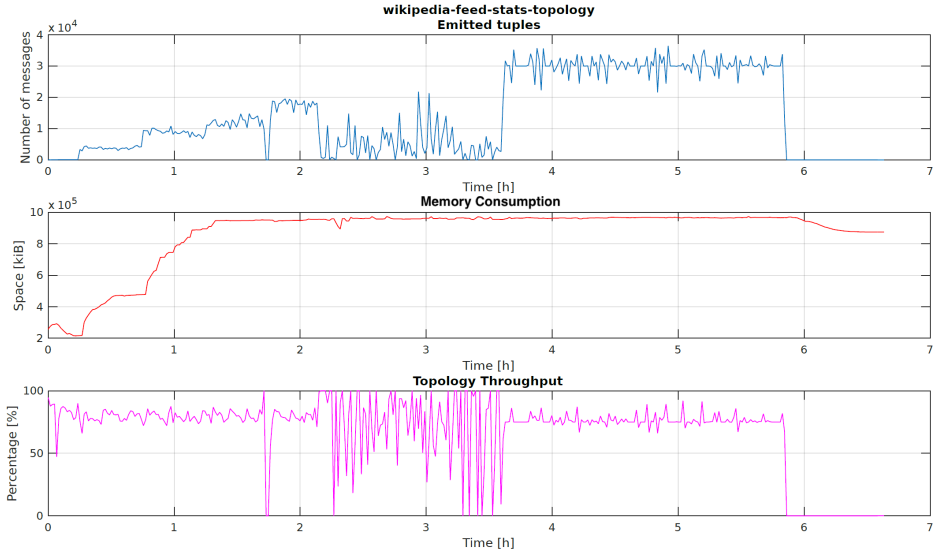

Figure 30: Anomaly behavior of `wikipedia-feed-stats-topology`.

A rejuvenation solution consists in adding a mechanism to the proposed restarting agent, to enable detecting the illustrated anomalies. The agent periodically monitors topology throughput by means of queries to Storm UI. If the throughput is null within the time interval, the restarting agent checks if the "logically previous" topology is working. For instance, if the `wikipedia-feed-parsed-data` TT is zero, then the restarting agent checks if the one belonging to the `wikipedia-feed-stream-topology` is non-zero. In that case, it means that the `feed-parse` topology stopped working and must be restarted. A different situation is when the zero TT is exhibited by the `feed-stream` topology. This could be due to three causes: (1) the connection is down, (2) the IRC server is down, (3) the topology stopped working. The restarting agent can check the first two conditions: no action is taken if one of them holds. If they are both false, the topology stopped working and must be reset.

A further approach for avoiding the reset of the topologies makes use of cloud computing. Storm may be deployed on a cloud infrastructure as illustrated in Fig. 31. In this way, when the workload increases, exploiting the auto-scaling mechanism, the virtual machine hosting the worker processes can be replicated according to a specified service level agreement [33]. However, this solution requires a partial change of the application. In particular, the `wikipedia-feed-stream-topology` should be removed, since its possible replication would only lead to doubling the messages written in the `wikipedia-feed-raw-data` topic. In any case, the best way for following the cloud computing approach would be to consider the implementation of auto-scaling mechanisms directly into Storm.

6. Conclusions

This work has investigated software aging issues in Apache Storm, a popular event stream processing system. Storm is a robust stream processor: even under heavy workload, it keeps working at the best of its possibility. In case of light workloads, Storm yields roughly constant throughput and latency and a

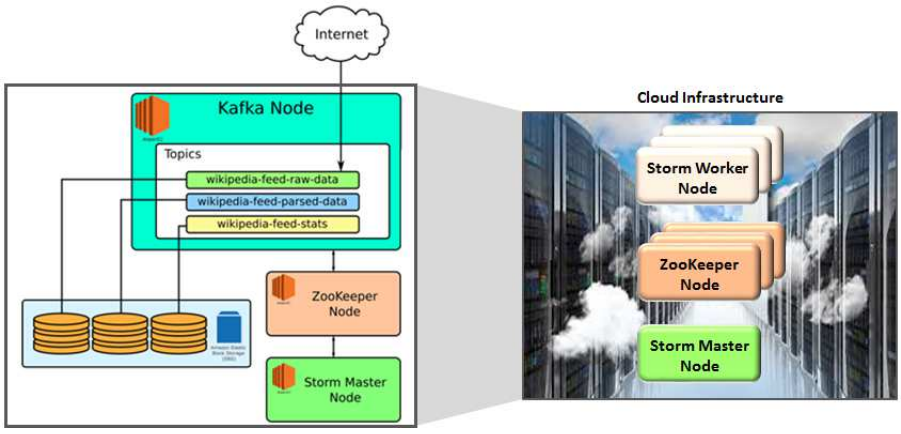

Figure 31: Proposed solution for the overload problem using cloud computing services.

negligible memory consumption trend. Under heavier workload (real workload amplified by a factor of 10x), aging turned out to be more evident in terms of memory consumption, affecting also the user perception in terms of response latency. Processes of the Storm architecture were also analyzed to pinpoint possible contributors, revealing the prominent role of some of them such as `Kafka` and `Nimbus`. Further experiments highlighted that the platform is subject to aging-related anomalous behaviors under extreme workload conditions, wherein Garbage collection plays a key role, which prevent some topologies from working continuously and fail silently. For addressing the aforementioned issues, a cloud-based solution is advocated. We found that in some cases Storm frequently kills the topologies, which are just doing garbage collection, although they are not actually dead. It has been shown that these resets are mainly due to the server lack of memory. Such shortage forces the worker processes GCs to run continuously in order to free the scarce memory available to the topology. This prevents the topology heartbeat thread from sending signals to Storm, which kills it after a certain timeout. In order to solve these problems, a rejuvenation action based on a *restarting agent* has been designed, implemented and tested, and other possible approaches have been discussed.

As specific aging manifestation, a possible Storm vulnerability has been also highlighted, due to the use of GC when running Java topologies. Such mech-

anism is well-known and absolutely robust, but since Storm uses it with its default parameters, in some cases it could not perform as expected. Therefore, the acknowledged aging-related anomalies could lead to undesirable behaviors in production environments if not taken into account. Solutions like the proposed *restarting agent* should be considered or, even better, the mentioned auto-scaling mechanisms should be implemented directly in Storm.

It is worth to point out that Storm is indeed a complex software – hence, the tests conducted in this work cover just a portion of all its aspects. For example, no experiments are made to test the exactly-once message processing mechanisms, or that considers the utilization of synthetic workload (e.g., to simulate other workload patterns), which could uncover some other critical issues in the system. On the other hand, the goal of the presented work was to run a *case study* to highlight how aging behavior in Storm *can* occur and to give insights into this specific instance of stream processors: indeed, further long-running tests need to be run in order to corroborate our findings and statistically generalize results.

Acknowledgment

The work by Pietrantuono and Russo has been supported by MIUR within the GAUSS project (CUP E52F16002700001) of the PRIN 2015 program.

References

[1] M. Cherniack et al. Scalable Distributed Stream Processing. Proc. *1st Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[2] Apache Storm - A distributed stream processing computation framework. Available at: http://storm.apache.org [accessed Sep. 2016].

[3] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems*, Vol. 10, No. 1, 2014.

[4] M. Grottke, R. Matias, and K. Trivedi. The fundamentals of software aging. Proc. *1st Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2008, IEEE.

[5] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi. Analysis of software aging in a web server. IEEE Transactions on Reliability, Vol. 55, No. 3, 2006.

[6] J. Zhao, K. S. Trivedi, M. Grottke, J. Alonso and Y. Wang. Ensuring the Performance of Apache HTTP Server Affected by Aging IEEE Transactions on Dependable and Secure Computing, Vol. 11, No. 2, pp. 130-141, 2014.

[7] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. Software Aging Analysis of the Linux Operating System. Proc. *21st Int. Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 71-80, IEEE.

[8] E. Marshall. Fatal Error: How Patriot Overlooked a Scud. Science, Vol. 255, No. 5050, pp. 1347-1347, 1992.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. Proc. *25th Int. Symposium on Fault-Tolerant Computing (FTCS)*, 1995, pp. 381-390, IEEE.

[10] T. Dohi, K. Goseva-Popstojanova, K. Vaidyanathan, K. Trivedi, and S. Osaki. Software rejuvenation: modeling and applications. In: *Handbook of Reliability Engineering*, 2003, pp. 245-263, Springer.

[11] Trident Tutorial. Available at: http://storm.apache.org/releases/1.0.1/Trident-tutorial.html [accessed July 2017].

[12] Apache Spark Streaming framework. Available at: http://spark.apache.org/streaming [accessed July 2017].

[13] Samza - A distributed stream processing framework. Available at: http://samza.apache.org [accessed July 2017].

[14] Apache Hadoop YARN (Yet Another Resource Negotiator). Available at: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at [accessed Nov. 2016]

[15] Flink - An open source platform for distributed stream and batch data processing. Available at: http://flink.apache.org [accessed July 2017].

[16] Yahoo! Benchmarks Apache Flink, Spark and Storm. Available at: www.infoq.com/news/2015/12/yahoo-flink-spark-storm [accessed July 2017].

[17] S.T. Allen, M. Jankowski, and P. Pathirana. Storm Applied: Strategies for Real-time Event Processing. Manning Publications, Greenwich, CT, 2015.

[18] Apache Storm. Highly available Nimbus design. Available at: http://storm.apache.org/releases/1.0.1/nimbus-ha-design.html [accessed July 2017].

[19] Apache ZooKeeper. Available at: http://zookeeper.apache.org [accessed July 2017].

[20] Apache Storm. Understanding the parallelism of a storm topology. Available at: http://storm.apache.org/releases/1.0.1/Understanding-the-parallelism-of-a-Storm-topology.html [accessed Nov. 2016].

[21] Apache Kafka - A distributed streaming platform. Available at: http://kafka.apache.org [accessed July 2017].

[22] D. Cotroneo, R. Pietrantuono, S. Russo and K. Trivedi. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. Journal of Systems and Software, Vol. 113, pp. 27-43, 2016.

[23] L.M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak. Using virtualization to improve software rejuvenation. Proc. *6th Int. Symposium on Network Computing and Applications (NCA)*, 2007, pp. 33-42, IEEE.

[24] D. Cotroneo, S. Orlando, R. Pietrantuono, and S. Russo. A Measurement-based Aging Analysis of the Java Virtual Machine. Software Testing Verification and Reliability, Vol. 23, pp. 199-239, 2013.

[25] A. Bovenzi, D. Cotroneo, R. Pietrantuono and S. Russo. On the Aging Effects Due to Concurrency Bugs: A Case Study on MySQL. Proc. *23rd Int. Symposium on Software Reliability Engineering (ISSRE)*, 2012, pp. 211-220, IEEE.

[26] J. Araujo, R. Matos, P. Maciel, R. Matias and I. Beicker. Experimental evaluation of software aging effects on the Eucalyptus cloud computing infrastructure. Proc. *Middleware 2011 Industry Track Workshop*, 2011, Article No. 4, ACM.

[27] F. Machida, D.S. Kim and K. Trivedi. Modeling and analysis of software rejuvenation in a server virtualized system. Proc. *2nd Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2010, pp. 1-6, IEEE.

[28] F. Machida, D.S. Kim, J.S. Park and K. Trivedi. Toward optimal virtual machine placement and rejuvenation scheduling in a virtualized data center. Proc. *1st Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2008, pp. 1-3, IEEE.

[29] J. Alonso, R. Matias, E. Vicente, A.M. Carvalho and K. Trivedi. A Comparative Evaluation of Software Rejuvenation Strategies. Proc. *3rd Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2011, pp. 26-31, IEEE.

[30] R. Matias, G.O. de Sena, A. Andrzejak and K. Trivedi. Software Aging Detection Based on Differential Analysis: An Experimental Study. Proc. *8th Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2016, pp. 71-77, IEEE.

[31] Mann-Kendall Test For Monotonic Trend. Available at: http://vsp.pnnl.gov/help/Vsample/Design_Trend_Mann_Kendall.htm [accessed July 2017].

[32] F. Machida, A. Andrzejak, R. Matias and E. Vicente. On the Effectiveness of Mann-Kendall Test for Detection of Software Aging. Proc. *5th Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2013, pp. 269-274, IEEE.

[33] N. Bessis, S. Sotiriadis, V. Cristea, and F. Pop. Modelling Requirements for Enabling Meta-scheduling in Inter-Clouds and Inter-Enterprises View Document. Proc. *3rd Int. Conf. on Intelligent Networking and Collaborative Systems (INCoS)*, 2011, pp. 149-156.