Workload Characterization for Software Aging Analysis

Antonio Bovenzi*, Domenico Cotroneo*, Roberto Pietrantuono*, Stefano Russo*†,
*Dipartimento di Informatica e Sistemistica, Universit di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy.
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Via Cinthia, 80126, Naples, Italy.
{antonio.bovenzi, cotroneo, roberto.pietrantuono, stefano.russo}@unina.it

***Abstract*—The phenomenon of software aging is increasingly recognized as a relevant problem of long-running systems. Numerous experiments have been carried out in the last decade to empirically analyze software aging. Such experiments, besides highlighting the relevance of the phenomenon, have shown that aging is tightly related to the applied workload. However, due to the differences among the experimented applications and among the experimental conditions, results of past studies are not comparable to each other. This prevent from drawing general conclusions (e.g., about the aging-workload relationship), and from comparing systems from the aging perspective.**

In this paper, we propose a procedure to carry out aging experiments in different applications for: *i)* assessing aging trend of the individual systems, as well as assessing differences among them (i.e., obtaining comparable results); *ii)* inferring workload-aging relationships from experiments performed on different applications, by highlighting the most relevant workload parameters. The procedure is applied, through a set of long-running experiments, to three real-scale software applications, namely Apache Web Server, James Mail Server, and CARDAMOM, a middleware for the development of air traffic control (ATC) systems.

***Keywords*-Software Aging, Empirical Study**

I. Introduction

The term *software aging* denotes a continued and growing degradation of software internal state during its operational life. This phenomenon is due to accrued error conditions that lead to progressive performance loss, eventually causing system hang or crash. Due to its cumulative property, it occurs more intensively in continuously and long-running running applications.

Until some years ago, software aging was judged as an occasional phenomenon experienced by few badly designed systems. Currently, as more and more software applications are reported to exhibit aging, it is recognized as a systematic non-negligible problem of long-running systems.

Past research efforts focused on predicting the time to failure of a system affected by aging, in order to trigger proper proactive recovery actions, known as *rejuvenation*, with an optimal schedule. Past studies can be distinguished in two classes. The former aims to estimate the optimal rejuvenation time by analytic modeling. It is referred to as *model-based* approach. The latter relies on measurements of the system's runtime health aiming to obtain predictions about impending aging failures. This is known as *Measurements-based* approach.

Among these, several studies reveal that aging dynamics are related to the *workload* applied to the system [12][13]. Aging bugs activation and resulting error accumulation depend on the way application is exercised; hence distinct workload patterns cause different behaviors in terms of aging dynamics. Such studies contributed to realize that a broader view on software aging needs to go beyond a mere *workload-independent* approach, since this turned out to be insufficient to describe the complexity of the phenomenon.

However, even if many of them reported relevant experiences, often on real-world applications, allowing researchers to claim the relevance of aging in today's systems, results of these analyses are often not comparable with each other, and hardly generalizable; workload parameters, when they are taken into account, are tied to the specific application under analysis.

This paper proposes a method to support a broad-scope analysis of software aging. We outline the sequence of steps to perform a workload-dependent aging analysis whose results can serve to practitioners as experimental samples for empirical analyses. Specifically, our goal is to:

- provide experimenters with a method to evaluate aging dynamics of their systems, producing results that can highlight aging of the specific application under analysis, but that can also be considered together with other studies' results for contributing to the body of empirical knowledge about the aging phenomenon. With time, this can lead researchers to *analyze the phenomenon of aging from a general perspective, independently from the specific application under analysis*;
- enable a *comparison among applications* in terms of aging, in order that experimenters can figure out how their application behave as compared to others (a sort of *aging benchmarking*);
- figure out, through a high-level workload characterization, how the aging phenomenon is tied to workload; i.e., *what application-independent workload features statistically affect the aging variation*;

The method is based on a workload characterization process that, starting from a high-level description, leads to design the list of experiments to perform on the target application(s). To validate the proposed method, we have designed and performed a series of long-running experiments on

three real-scale software applications, i.e., the Apache Web Server, the James Mail Server (the Java mail server), and CARDAMOM, a middleware used by air traffic control (ATC) applications. Results highlight aging trends in each of the proposed case study in different experimental conditions; moreover, due to the proposed characterization, it has been possible to compare the applications among each other, and to evidence the most statistically relevant workload parameters influencing aging dynamics.
The rest this paper is organized as follows: Section II surveys the existing literature about software aging, whereas Section III describes the steps of the proposed method. Section IV details the experimental study carried out by applying the proposed method, while Section V presents the obtained results. Finally, Section VI concludes the paper.

II. Background and Related work

Software aging refers to the accumulation of errors during long-running application execution, which may lead to *i)* performance degradation, and *ii)* application/system hang or crash. This phenomenon has been observed in various operational systems (e.g., web servers [2], middleware [3], spacecraft systems [4], military systems [5]), causing serious damages such as loss of money or human lives. It is due to the activation of the so-called aging-related bugs, a particular class of software bugs, which manifest their effect only after a long period of time from their activation [1]. Some examples of aging bugs are: memory leaks (memory allocated portions of a process but no longer used or usable), unterminated threads, poor management of shared resources, data corruption, unreleased file-locks, accumulation of numerical errors (e.g., round- off and truncation), and disk fragmentation.

Rejuvenation techniques are widely adopted to mitigate software aging effects, by preventing the system from failing. Their purpose is to restore a "clean state" of the system by releasing OS resources and removing error accumulation. Some common examples are garbage collectors (e.g., in Java Virtual Machines) and process recycling (e.g., in Microsoft IIS 5.0). In other cases, rejuvenation techniques result in partial or total restarting of the system: application restart, node reboot and/or activation of a standby spare.
Research studies on software aging, [7][8][10], try to the figure out the optimal time for scheduling the rejuvenation actions. This is typically done by either *Analytical approaches*, in which the optimal rejuvenation schedule is determined by models, or by *Measurements-based approaches*, in which the optima schedule is determined by statistical analyses on data collected from system execution. In the former case, stochastic processes (e.g., Markov Decision Processes, Semi-Markovian process, Stochastic Petri Nets), representing the system's states, are adopted to model the software affected by aging, through several distributions [7], [8]. Instead, the work presented in [11], and then in [12], is an example of the measurements-based approach; in this case authors report results of an analysis where the 33% of outages were due to aging. Other examples of measurement-based analyses are about performance degradation of OLTP server [9], and of the Apache Web Server [2]. Trivedi et *al.* [10] proposed also a combined solution by composing modelling with a measurement-based approach.
Several works have investigated the dependence of software aging on the workload applied to the system, by showing the importance of considering the workload in models that control rejuvenation activities. In [12], authors present an analysis, extending the work in [11], that takes into account some workload parameters, such as the number of CPU context switches and the number of system call invocations. Their results confirm that aging trends are related to the *workload states*. A comparison with [11] (on the same case study) shows that the workload-driven methodology is by far more powerful and useful to model aging phenomenon. However, workload indicators used in [12] are internal (i.e., system-level) parameters, and are hardly controllable factors. A work that addresses the impact of application-level workload on aging trends is presented in [14]. Authors applies the Design of Experiments approach to draw the effect of controllable application-level workload parameters on aging. However, their focus was on one specific application, i.e., Apache Web Server, and so they consider application-specific workload parameters (e.g., the page size, the page type, http requests rate).
Madeira et *al.* [15] highlighted the presence of aging in a Java-based SOAP server, highlighting that aging trends are related to the workload distribution. In [16] some best practices are provided to build empirical models for Time To Exhaustion (TTE) prediction. These best practices also address the selection of workload variables. Our previous works [6] [17] report the presence of aging trends both at JVM and OS level; results showed relationship between workload parameters, such as method invocation frequency and object allocation frequency, and aging trends.
However, in all the cases parameters are specific either to the system (such as in [12]), or to the particular application under analysis (such as Apache, JVM, Linux OS), leaving the final considerations valid only for that specific case. Results of these works would be much more useful if they were comparable to each other, since they would enable more general analysis of aging-workload relationship.
Our aim is to define a methodological approach to obtain comparable results about aging of different applications, i.e., results produced under comparable workload conditions.

III. Definition of the Experimental Procedure

The proposed procedure allows: $i)$ to carry out experimental campaigns to assess aging trends, whose results are comparable to each other, and $ii)$ to infer workload-aging relationships, and highlight the most relevant workload parameters influencing aging dynamics. Figure 1 synthesizes the procedure's steps, which are described in the following subsections.

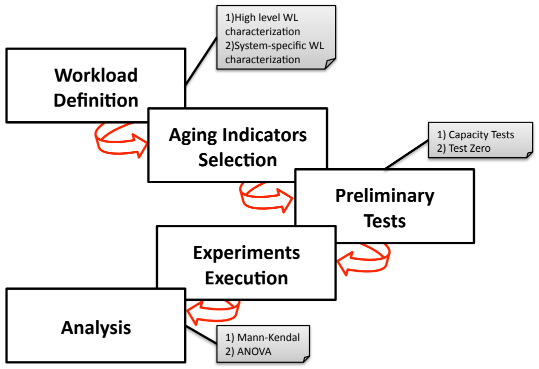


Figure 1: Steps of the proposed procedure.

A. *Workload Definition*

The goal of the workload characterization is to have a first set of parameters *whose ability to describe the load is as much application-independent as possible*. Starting from the reviewed papers that investigate aging-workload dependency, we have identified some key requirements for the workload definition.

- *Comparability*: the parameters describing workload of different applications should be comparable; therefore parameters should be described from a *high-level* perspective *without referring to application-specific inputs*.
- *Basic Description*: application's behavior varies according to the applied workload, depending on the *amount* and *type* of work. At a first level of abstraction, characterization should include these two basic macro-dimensions (cf. with [14]), which are then exploded in successive refinement steps.
- *Specialization Ability*: it should be possible to specialize the high-level application-independent parameters into application-dependent workload parameters, so that practitioners can simply setup their experiments starting from the high-level workload description.
- *Realistic*: workload parameter values should fall inside realistic ranges, e.g., actually observed during operation, in order to avoid drawing misleading conclusions.
- *Practicability*: the number of workload parameters should be kept reasonably low, in order to allow experimenters to design a representative and cost-effective campaign. This number should account for the budget (time and machine resources) available for experiments.

To define workload parameters according to the outlined requirements, we first proceed by defining a *high-level workload characterization*; then, this is refined into an *application-dependent workload characterization*, in order to enable the actual experiments execution on real case-studies.

High-level workload characterization

We assume that, at high level, the load imposed to an application can be represented as a generic *request of service*, considered from the user point of view, characterized by a *request type*, among a set of types, and by input/output parameters. A request of service is characterized by the following parameters:

Intensity. It represents the stress level of the application. In absolute value, it is measured as number of requests per second. In relative value, it is measured as percentage with respect to the maximum capacity of the system.

Parameters Size. It indicates the value of the overall size of exchanged data in input/output. This indicator represents the amount of data processed by a request, and it can have an impact on software aging. This value will vary from a MIN to a MAX value, determined by real observations in the field.

Types of Request. It indicates how many different requests can be served by the application ($e.g.$, sending or receiving a message, executing a query); this may affect aging bugs activation, and thus aging manifestation, since different parts of the application code can be exercised in response to different request types.

There is a trade-off between the number of request types to take into account, and the number of experiments. Thus, if it is necessary to reduce the number of requests, different types should be merged into one type, heightening their level of abstraction (for instance, *open a connection* and *sending a message* could be merged into "sending a message" requests).

Variation of Request Type. Given a request of type T, this parameter represents the probability that the next request will be of a type different from T. In other words, this index represents the variability of request types. This probability can be represented by a first-order Discrete Time Markov Chain (DTMC), where states represent request types; from a state, the DTMC evolves according to the type of the next request. Transition probability from state i to state j represents the probability that the next request will be of type j, given that the current request is of type i. The probability that a request is repeated is therefore the transition probability from state i to itself, $p_{i,i}$. The *Variation of Request Type* is therefore $1 - p_{i,i}$; this will vary from a MIN to a MAX level. Once the $p_{i,i}$ value is chosen, we assume that the other transition probabilities are equally distributed (i.e., considering that each row must add up to 1, $p_{i,j} = (1 - p_{i,i})/n$, for $j = 1, \ldots, n$ with n denoting the number of request types).

To summarize, the above parameters indicate: *i)* how many requests are, in the average, submitted to the system; *ii)* the average size of data the system must process in a request; *iii)* what types of request are submitted to the system; *iv)* how frequently request types vary.

Application-dependent workload characterization.
Since the goal is to conduct real experiments, these high-level workload parameters need to be refined into application-dependent ones. In this stage, the experimenter has to characterize the parameters with respect to the application(s) involved in the experimental campaign.
The *intensity* of request is viewed, at high level, as the number of service requests per unit time. We need to know what "request" means for each application. For instance, if we consider a Web Server, it can be the number of page requests per second, as quantified in several works [14],[18]. Hence, the *Types of Request* parameter values have to be determined. This is done by considering the most relevant requests for each application, which can be retrieved from documentation and/or from field data. Requests can be merged if the number is too high (as described above). The *Variation of Request Type* parameter will then refer to the established types of request. As for the *Parameters size*, values should be representative of the actual size of parameters exchanged between user and application (i.e., *realistic* requirement). A practical way to determine the MIN and MAX values, i.e., the *Low* and *High* levels, is to consider field data regarding the application under study.

B. Aging Indicators Selection

After the workload definition, the second step concerns with the selection of aging indicators. Aging effects can be observed during the system execution, by monitoring the so-called *aging indicators*. Aging indicators are explanatory variables that, individually or in combination, suggest if the system state is degrading or not [23]. Past studies showed that software aging manifests itself mainly as resource depletion (typically, memory depletion) and performance degradation [22], [14], [15], which are therefore the most common indicators in the literature. While memory depletion can be considered an application-independent measure, performance degradation can be measured in several ways depending on the application (e.g., response time, round trip time, number of served requests per second). Moreover, it should be considered that aging dynamics may be different among the considered applications: in order to carry out comparable analyses we cannot consider absolute values, but the increase (or decrease) of aging must be measured by *relative* values. In other words, data must be normalized. Along with the absolute aging indicators, also relative aging indicators need to be considered, e.g., by normalizing data with respect to the minimum observed aging for that system.

C. Preliminary tests

In the previous steps, workload parameters to be set in the experiments and aging indicators to monitor are defined. Before executing the experiments, some tests should be performed in order to determine the limits of the applications under test, and complete the previous steps.
For instance, the measurement of the *intensity* parameter has been hypothesized as the percentage of the maximum system's capacity. Thus, experimenter should determine what is the maximum capacity for the system under test, i.e., s/he has to perform a *Capacity test*. This implies: *i)* determining a metric to measure the throughput of the system, and then *ii)* soliciting the system with an increasing load, in terms of *intensity*, until a knee in the throughput curve is reached. The knee indicates the limit of the system's capacity, since beyond such limit the system is no longer able to serve requests properly (i.e., as the request rate increases beyond that limit, its throughput does not increase anymore, as it would be expected, but it remains the same or even decreases). Since different limits can be reached with different request types, capacity tests have to be performed per each request type. Then, either the average of the observed limits, or, to be conservative, the minimum of such limits, is chosen as the system's capacity. The latter choice assures that, during experiments, the system will not fail due to the system's capacity exceeded; this helps distinguishing failures due to aging, from failures due to excessive loads.
One more issue is related to the time duration (T) of experiments. An experiment should last for an amount of time at least sufficient to observe a significant trend in data. Such time is system-dependent; a pre-defined experimental time for all the applications would imply either a useless experiment, if T is too low, because of insufficient number of samples to determine a trend, or an expensive experiment, if T is too high. Thus, an additional preliminary test is needed to estimate the best experimental time: we name it *Test zero*. Under the assumption that the less intensive workload parameters, the lower aging trends are, the *Test zero* aims to investigate if there is a trend with the least stressful workload. Least stressful means that parameters are set at their minimum level, including the *intensity* parameter. With samples of this test, the experimental time for each system is evaluated, by using an algorithm developed and presented in our previous study [3]. The algorithm determines the minimum time in order to observe statistically significant trends for a given response variable. It takes the desired error that one can tolerate and samples of the *Test zero*, as inputs; then it estimates if the number of samples collected up to a given time t is sufficient to have a significant trend (e.g., at 95% confidence level). When more response variables are involved, the minimum experimental time T is the maximum of times obtained for the considered response variables. Since the *Test zero* considers the least stressful workload, all the experiments with more stressful workload will exhibit, under the mentioned assumption, aging trends within T. Note that the formulated assumption is reasonable and confirmed by previous works [3], [14]. Output of this test may be used for aging indicator normalization.

D. Experiments and Data Analysis

Design of Experiments (DoE).
After preliminary tests, the experiments planning is carried out by the Design of Experiments (DoE) technique. The DoE [19] is a systematic approach to the investigation of a system or a process. A series of measurement experiments are designed, in which planned changes are made to one or more system (or process) input factors. The effects of these changes on one or more response variables are then assessed. The DoE aims to plan a minimal list of experiments to be applied in order to get statistically significant answers.
The first step in planning such experiments is to formulate a clear statement of the objectives of the investigation. Then, the next steps are concerned with the choice of *response variables*, and with the identification of *factors* of interest that can potentially affect the response variables. A particular value of a factor is called *level*. A factor is said to be controllable if its level can be set by the experimenter, whereas the levels of an uncontrollable factor cannot be set, but only observed. The identification of response variables, factors, and levels is followed by the definition of a list of experiments, called *treatments*. Each treatment is obtained by assigning a level to each one of the controllable factors.
In order to assess the impact of workload on software aging, we have to plan a set of experiments by varying workload parameters value, and by evaluating the effect of planned changes on aging trends. Thus, the chosen aging indicators are the response variables to be observed, whereas the defined workload parameters are the factors of the experiments. Since experiments, in our case, will be carried out on more than one software application, we also consider an additional factor, i.e., the *software type*. The output of this phase is the list of experiments to perform.
Data Analysis.
The obtained list of treatments is finally executed (step 4 of Figure 1). Collected data are analyzed, in order to determine the presence of trends through statistical hypothesis tests (the most common one is the Mann-Kendall test [20]) for each response variable.
For data analysis (step 5), both absolute and relative (i.e., the normalized) aging indicators must be considered. The former are useful to highlight the aging trends experienced by each application. The latter are used for conducting the classical *Analysis of Variance* (ANOVA), which tell what workload parameters mainly impact on aging trends.
The objective of the analysis is to figure out: *i)* if (and to what extent) the analyzed systems suffer from aging, *ii)* how much the variation of workload parameters from low levels to high levels impacts the *variation* of aging trends, and *iii)* which is the most influential application-independent workload parameters. Since we consider various systems, results allow us also to compare them in terms of aging.

IV. Experiments Execution

In this Section we apply the outlined procedure to plan and execute experiments in three different software applications. Starting from the defined high-level characterization, the *application-dependent* workload characterization, as well as the experimental design, are shown in the following.

A. Application-Dependent Workload Characterization

1) ***Case-studies:*** Applications selected as case-studies belong to three different classes of long-running software systems, which are employed in several contexts ranging from business to mission critical scenarios. They are:
Apache Web Server
Apache is one of the most popular Web HTTP server. We used the version 2.0.48. In the configuration, we enabled SSL, by including the SSL module, and used the MySQL DBMS, version 4.5. The configuration parameters have been tuned as in past studies [14], also to compare results: *StartServers*=150, *MinSpareServers*=150, *MaxClients*=150, *MaxSpareServer*=0, *MaxRequestsPerChild*=0, *MaxKeepAliveRequests*=0. By setting the first three directives to 150 we minimize the waiting time for the management of a request. The directive *MaxClients* lets it serve 150 simultaneous connections, whereas the *MinSpareServers* and *MaxSpareServers* directives allow having 150 child processes waiting for managing connections. *MaxKeepAliveRequests* and *MaxRequestsPerChild* directives allow a process to serve an unlimited number of requests, and a connection to have an unlimited number of requests.
CARDAMOM
CARDAMOM is a CORBA-based middleware tailored for the development of mission critical, near real-time, distributed applications[1]. Its current usage is in the field of Air Traffic Control (ATC) systems. The CARDAMOM architecture is composed of several services, classifiable in basic and pluggable services (i.e., which are included/excluded according to the user needs). In our experiment, we configured the *Naming Service*, *Load Balancer* and *Trace Service*[2], which are widely used in practice. We use the version 3.1.
James
James is a JAVA-based mail server supporting several mailing protocols, such as SMTP, LMTP, POP3, IMAP[3]. It has a modular architecture designed to separate the processing phase from the delivery of a message. The system consists of several components: SMTP server, POP server, NNTP server, DNS server, *FetchMail*, *SpoolManager*. The heart of the system is the spool manager, which implements the processing layer. However we have disabled the components not well tested (e.g., the NNTP server). We used the JVM 1.5_16 with 512MByte of initial allocated memory, extendable up to 1024MByte.

[1]http://cardamom.ow2.org/
[2]see http://cardamom.ow2.org/docs/ for documentation
[3]http://james.apache.org/

*2) **Factors and levels specification:*** In this Section, high-level workload parameters are mapped to system-specific ones. Factors and levels are specified for each application.

Apache Web Server

First, the *Types of Request* factor has to be determined. Unlike past studies (e.g., [14], [18]), that considered at most two types of request (i.e., static and dynamic), we consider the following requests: HTTP request of a static page, HTTP request of a dynamic page (created by a CGI script), HTTPS request of a static page, and HTTPS request of a dynamic page. As for levels, this factor ranges from a minimum of 2 requests, to a maximum of 4 requests. For the *Intensity* factor, we take the number of pages requested by clients per second. Levels of this factor will be determined after preliminary tests, reported in the next Section.

The *parameters size* factor is intended as the size of the required page. To determine the values for levels of this factor, we take into account the average page size for the home pages of the most popular web sites, which was found to be 200KB [14], as medium level. The low and up levels are set respectively to 50KB and 450KB. Finally, levels for the *Variation of Request Type* are set to 50%, 30% and 90% (medium, high, and low level, respectively), meaning that the probability of not repeating the same request, $1-p_{i,i}$ in the DTMC representation, is 0.5 in the medium level, 0.3 (i.e., $p_{i,i}=0.7$) in the low level, and 0.9 (i.e., $p_{i,i}=0.1$) in the high level. $p_{i,j}$ values are derived as explained in Section III-A: $p_{i,j}=(1-p_{i,i})/n$, for $j=1,\ldots,n$ with n = 4 (we have four request types).

James

In order to stimulate different parts of the code, the *Types of Request* for James include: sending of emails (SMTP protocol), receiving of emails (POP3 protocol), sending spam messages (testing of the SPAM processor), and sending messages with an attachment (testing of the virus processor). The levels range from a minimum of 2 requests, to a maximum of 4 requests. The *intensity* parameter is measured as number of mails processed per second (its levels are determined after preliminary tests). *Parameters size* refers to the size of the email. The medium level was set to the average size of emails processed by a provider [4], i.e., 70KB ; the high and low level of the parameters size are respectively set to 5KB and 200 KB. As for the *Variation of Request Type* parameter, we have the same probabilities as in Apache, since four request types are considered also in this case.

CARDAMOM

To stimulate the middleware we use a client-server application (developed in the context of the COSMIC project[5]), which exploits CARDAMOM facilities by using the following services: *Naming*, *Load Balance*, and *Trace* Service. Therefore, we tested the following request types: sending of messages, receiving of messages, ping messages, logging (i.e. a request that stresses the *Trace* service). Again, levels are from 2 to 4 requests. The *intensity* of requests is measured in terms of number of concurrent clients processing the requests (each client performing 200 operations); levels for this factor are determined after preliminary tests. *Parameters size* refers in this case to the size of the exchanged message. Such messages correspond to flight data plans, which in the average are of 20 KB[6]. The low and high levels are respectively of 5KB and 200KB. The *Variation of Request Type* parameter assumes again the same parameter as in the previous two applications, i.e., 30%, 50%, and 90%.

[4]http://www.email-marketing-reports.com/

[5]http://cosmicsite.criai.it/

B. Experimental campaign

All the experiments are executed on the same machines, with the same OS, equipped with 2 Xeon Hyper-Threaded 2.8 GHz CPUs, 5 GB of physical memory (both clients and servers). The machines are on LAN connected via a 100 Megabit Ethernet, and each workstation has Red Had 4 OS with kernel version 2.3.19, initialized with minimal system services. Each case study is composed of a client, which has the role of traffic generator, and a server.

In our experiments, we focused on the following aging indicators: *memory depletion* and *throughput loss*.

To measure memory consumption (MC), we use the standard Linux kernel utility, *free*. MC is given by:

$$MC = TM - FM - PC \qquad (1)$$

where TM, FM, and PC are total memory, free memory, and page cache size, respectively. In particular the page cache contains a copy of recently accessed files in kernel memory. Since the page cache can get all the free memory, its memory consumption can be quite large, and could bias the analysis; therefore, it is subtracted from MC. The MC is periodically sampled (each 30 seconds) and stored in a trace.

To measure throughput loss we use existent or ad-hoc load generators, which are placed in the client machines. They are extensively tested before experiments, in order to assure that they are aging-free. In particular, for Apache we use *httperf*[7] during capacity test, and *JMeter*[8] during experiments execution (which allows better control on experimental factors). These tools can collect the total number of correct reply, and the average response times. For CARDAMOM and James, we developed ad-hoc load generators.

After the experiments, trend detection is performed by the Mann-Kendal test (adopted also in [11], [2], [17]). The trend, if any, is estimated by means of the non-parametric Sen's procedure [20], which is known to be robust, i.e., it does not assume normally distributed measurement errors, and resistant, i.e., it is not sensitive to outliers. For data analysis, a significance level (denoted by α) of 10% is considered.

[6]http://www.swim-suit.aero/swimsuit/projdoc.php

[7]http://www.hpl.hp.com/research/linux/httperf/

[8]http://jakarta.apache.org/jmeter/

*1) **Preliminary Tests**:* The goal of preliminary tests is to measure: *(i)* the maximum limit of the *intensity* factor, (*Capacity tests*); *(ii)* the minimum time duration of experiments necessary to observe aging (*Test Zero*); *(iii)* the aging trend with all the parameters set to their low level (*Test Zero*). The latter acts as comparison term, and may be used to build normalized, i.e., *relative*, aging indicators.

Capacity Test

By increasing the number of requests (i.e., the *intensity* factor), systems eventually reach a knee where the throughput no longer increases. We evaluate several limits, reached by submitting in each capacity test a different type of request. Then, the minimum of such limits is chosen, as explained in Section III-C. Referring to Apache, it has been experienced that increasing the number of requests beyond about 400 requests per second, the server is no longer able to correctly reply. Specifically, the limit has been measured at 382 requests per second. As for CARDAMOM, the load generator increases the number of concurrent clients over time. In this case, the percentage of correctly processed requests gets saturated with more than 10 concurrent clients, i.e., more than 2000 requests. Finally, with James, the test reports that beyond 22 emails per second, the number of correctly processed emails decreases. These values are used to determine the levels of the factor *Intensity*: we set the high level at 90% of the maximum capacity, the medium level at 50% of the limit, and the low level at 30% of the limit. This completes the factor levels determination. Table I summarizes the list of all the factors and their levels.

Table I: Application-dependent Factors and their Levels

	Intensity (req/sec)	*Parameter size* (KB)	*Variation of request type*	*# of requests type*
Apache	Low: 115 Medium: 191 High: 344	Low: 50 Medium: 200 High: 450	Low: 30% Medium: 50% High: 90%	Low: 2 High: 4
CARDAM.	Low: 400 Medium: 1000 High: 1800	Low: 5 Medium: 20 High: 200	Low: 30% Medium: 50% High: 90%	Low: 2 High: 4
James	Low: 7 Medium: 11 High: 20	Low: 5 Medium: 70 High: 200	Low: 30% Medium: 50% High: 90%	Low: 2 High: 4

Test Zero

To evaluate the minimum time duration of each experiment, the algorithm that we proposed in [3] has been adopted. It outputs this time starting from samples of *Test Zero* and from the error value ϵ that can be tolerated (we set $\epsilon = 0.01$). Table II reports the levels of each factor set for the *Test Zero*. According to what described in Section III-C, factors are set to the least stressful configuration that let us observe aging. The lightest type of request, among the types identified for each application, is determined by the experimenter through a quick test with each request (it is necessary to monitor system behavior for a few minutes per type). The factor *Variation of request type* is always set to 0% (i.e., the current request is repeated in 100% of the cases); the values of the factor *parameter size* are set to their minimum level. Finally, as for the *Intensity* factor, the right value are empirically tuned; i.e., if the first experiment does not cause aging, it is repeated by increasing the intensity value until a trend is observed. For all the applications, we

Table II: Levels of each factor for the Test Zero

	Types of request	*Intensity* (req/sec)	*Parameter size* (KB)	*Variation of request type*
Apache	HTTP static	115	50	0%
Cardamom	send message	600	5	0%
James	send mail	7	5	0%

found, by executing this test, a relevant trend in the *memory depletion* indicator; whereas, for throughput loss, a slight trend has been observed only in James, and amounts to $-3.1902 \cdot 10^{-6}$ $operations\ per\ second.$

Giving samples of these tests as inputs to the algorithm, the minimum experimental time is also obtained. Results of *Capacity Test* and *Test Zero* are summarized in Table III, which reports the *Capacity limit*, the *Minimum time duration*, the *Memory depletion trend* and the *Throughput loss trend*, with the lightest workload.

Table III: Results of *Capacity tests* and *Test Zero*.

	Apache	*Cardamom*	*James*
Capacity limit (req/sec)	382	2000	22
Minimum exp. time (h)	24	4	24
Memory depletion (KB/sec)	0.163	78.2	38.3
Throughput loss (Op./sec)	no trend	3.190210^{-6}	no trend

*2) **Experimental plan and execution**:* In previous Sections, the workload to apply has been characterized in terms of parameters and their levels; then aging indicators to observe have been selected, and, finally, the limits of each application have been determined. According to the outlined procedure, we are in this phase provided with all the necessary information to design experiments. To generate the experimental plan according to the DoE approach (step 4 of Figure 1), we adopt the tool *JMP*®[9]. *JMP* allows to create a plan that adheres to the principle of DoE (e.g., randomization, orthogonality), generating a minimal list of treatments necessary to get statistically significant responses. In this plan, factors are workload parameters (Table I), response variables are the aging indicators. The list of treatments is reported in Table IV. Columns report the involved factors, set with a given level value. In addition, we also consider the software type as factor, whose levels are the chosen applications. Experiments are executed for the time duration determined by the *Test Zero*, amounting to a total of 210 hours of experimental time.

[9]http://www.jmp.com/

Table IV: Experimental plan. M=Medium, L=low, H=High

Exp ID	*Case study*	*Types of Request*	*Intensity* (req/sec)	*Parameter size* (KB)	*Variation of request type*
1	Apache	L (2)	M (191)	H (450)	M (50%)
2	Cardamom	H (4)	L (400)	H (250)	H (90%)
3	James	L (2)	M (11)	M (70)	H (90%)
4	Apache	L (2)	L (115)	L (5)	L (30%)
5	Cardamom	H (4)	M (1000)	M (20)	L (30%)
6	Cardamom	L (2)	M (1000)	L (5)	M (50%)
7	Apache	H (4)	M (1000)	L (50)	H (90%)
8	Apache	L (2)	L (115)	M (200)	H (90%)
9	James	L (2)	H (20)	H (200)	L (30%)
10	Apache	H (4)	H (344)	M (200)	M (50%)
11	Cardamom	L (2)	H (1800)	L (5)	H (90%)
12	James	H (4)	L (7)	L (5)	M (50%)

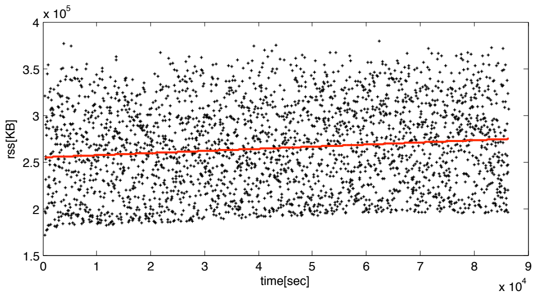
V. Results and Data Analysis

Aging indicators' samples, collected each 30 seconds, are first analyzed by means of statistical tests, in order to estimate aging trends, if present. Table V lists the results that we obtained for each treatment. It reports trends, estimated by the Mann-Kendal test, for both memory depletion (MDT) and throughput loss (TLT), and the computed indexes, named, respectively, MDI (memory depletion index) and TLI (throughput loss index). These *relative* indexes are computed by normalizing the MDT and the TLT by means of the min-max formula ($index = (RawTrend - Min)/(Max - Min)$) in order to have aging indexes between 0 and 1 (Min is the minimum between the *Test Zero* and trends observed in these treatments).

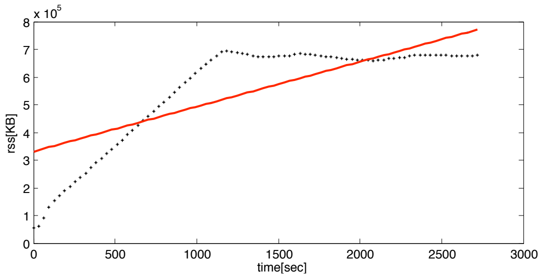
Table V: Results of experiments

Id	*MDT* KB/sec	*MDI*	*TTE* hours	*TLT* ($*10^{-2}$) op/sec	*TLI*
1	0.185	0.4557	7357.4	0	0.00
2	110.4	0.3852	12.3	3.43	1.00
3	153.4	1.0000	8.9	6.85	1.00
4	0.149	0.0000	9135.0	0	0.00
5	116.7	0.4605	11.7	2.33	0.68
6	84.39	0.0740	16.1	0	0.00
7	0.203	0.6835	6705.0	0	0.00
8	0.176	0.3418	7733.6	0	0.00
9	122.9	0.7350	11.1	0	0.00
10	0.228	1.0000	5969.8	0	0.00
11	161.8	1.0000	8.4	2.07	0.13
12	64.1	0.2242	21.2	1.94	0.28

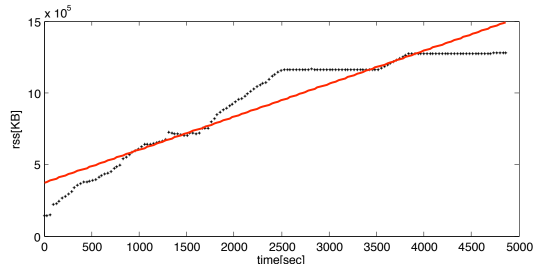
From results, it is clear that memory depletion is the most significant aging indicator, since we have experienced a trend in all the treatments; contrarily, a slight throughput loss trend has been observed only in five treatments. Hence, since throughput loss exhibits much slower dynamics, we conduct the subsequent analyses only considering the memory depletion trend. Figure 2 shows snapshots of the most relevant (i.e., with the highest slopes) memory depletion trends, observed for each of the considered applications.



(a) Treatment 10, MDT Apache.



(b) Treatment 11, MDT Cardamom.



(c) Treatment 3, MDT James.

Figure 2: Most relevant experiments

Considering the absolute values (cf. column 2 and 4 of Table V), the treatments that revealed by far the lowest aging trends are those carried out on Apache (that are: #1, #4, #7, #8, #10), which exhibited trends two orders of magnitude lower than CARDAMOM and James. Observed values for Apache are also consistent with those observed in past studies [21]. Apache experienced an average trend of $0.1856\ KB/sec$, whereas in CARDAMOM and James it is of $110.298\ KB/sec$ and $94.75\ KB/sec$, respectively. This behavior leads CARDAMOM and James to very short times to failure due to resource exhaustion (the expected *Time To Exhaustion*, i.e., the TTE, is reported in Table V, column 4.). For CARDAMOM, consequences of this result could be even more dangerous, since its foreseen employment is in mission critical systems for Air Traffic Control (ATC). Note that results are comparable to each other, since they have been obtained under comparable workload conditions, i.e., referring to the same high-level workload parameters, set with values that are relative to the limits of each application.

To complete the analysis, we conduct the ANOVA in order to assess the effect of factors on response variables. The analysis consists in determining which of the principal factors (i.e., the workload parameters: *Intensity*, *Paramaters size*, *Types of request*, and *Variation of request type*), impact on aging indicators. Since aging dynamics turned out to be noticeably different depending on the application, we evaluate the influence of workload parameters on aging by considering the *relative* aging indicator, i.e., the *MDI*.
ANOVA is conducted by testing the hypothesis, H_0, that the factor F_i does not statistically affect the response variable. Table VI reports results of one-way ANOVA for each considered factor, with a significance level for acceptance/rejection of α = 0.10 (i.e., confidence of 90%). The analysis shows

Table VI: Results of the Anova

Factor	*p-value*	Outcome
Intensity	0.0235	$H0$ can be rejected with $\alpha < 0.1$
Parameter size	0.4837	$H0$ cannot be rejected
Types of requests	0.55	H0 cannot be rejected
Variation of req. type	0.13	H0 cannot be rejected

that the most influential factor is the *Intensity* factor, with a *p-value* lower than 0.05. Other factors do not significantly affect the response variable. More in detail, Table VII reports data aggregated by factors. It is evident that requests with high intensity exhibit, in the average, much higher trends. Differences can be noticed also in the other parameters, whose variation from low to high level causes an increase in the experienced aging. However, only for the intensity parameter, we found a statistical evidence. Hypothesis tests report that the difference between experiments with high and low level of the factor:

- *intensity* (i.e., 0.6738) is significant at $p - value = 0.00153$, i.e., with a confidence greater than 90% (approximately of 99%);
- *parameter size* (i.e., 0.1289) is not significant ($p - value = 0.3235$);
- *number of request type* (i.e., 0.0354) is not significant ($p - value = 0.4367$);
- *variation of request type* (i.e., 0.2836) is not significant ($p - value = 0.1466$).

Other than the *intensity* parameter, it should be noted that the second greatest difference is in the *variation of request type*, i.e., of how often the type of request is varied. The difference

Table VII: Results of MDI. Average MDI grouped by factors

Factors	*High*	*Medium*	*Low*
Grouped by Intensity	0.9117	0.5348	0.2378
Grouped by Parameters Size	0.5253	0.7006	0.3963
Grouped by Types of Request	0.5507	-	0.5152
Grouped by Variation of Request Type	0.6821	0.4385	0.3985

between experiments with 90% and with 30% of chances to not repeat the same request, is significant with a confidence of about 85%. It seems that more varying workload causes greater aging.
Finally, regarding the additional factor *software type*, the differences between CARDAMOM and Apache, and between James and Apache turns out to be significant with $p - value << \alpha = 0.1$, (CARDAMOM-Apache, $p - value = 2 * 10^{-4}$, James-Apache $p - value = 4 * 10^{-4}$), unlike the difference between CARDAMOM and James.

VI. Conclusions

In this work, the phenomenon of software aging and its relation with workload have been investigated. To overcome limitations of past studies, which typically analyze the phenomenon on single software applications, we defined a procedure to design experiments for revealing aging, and able of producing comparable results as well. The procedure has been applied through an experiment on three large software applications. The first result regards the identification of aging trends, in terms of *memory depletion*, in all the experimented applications. As secondary result, due to the adoption of the outlined procedure, we have been able to compare systems from the aging point of view: the comparison highlighted that CARDAMOM and James showed higher aging dynamics than Apache (the average differences are 118.13 KB/sec and 113.27 KB/sec for CARDAMOM-Apache and James-Apache, respectively). The average TTE in hours, for our configuration, are: 7672, 12.1, and 13.7 for Apache, CARDAMOM and James, respectively. Finally, the third result is about the possibility to infer aging-workload relationships using results obtained from different applications; from this point of view, the analysis, even though preliminary, showed that more stressful workloads, in terms of *Intensity* (number of requests per unit time), greatly influence the manifestation of software aging, and is statistically the most influential factor. As for the *Parameter size*, *Types of requests*, and *Variation of Request Type* factors, although some of them seem to affect aging (e.g., *Variation of Request Type*, at 85%), there is no statistical evidence at a confidence of 90%. However, regarding the latter point, it is no possible to provide a definitive answer about their influence on aging, since the obtained results are affected by the limited number of samples. As any in empirical study, this represents a threat to results validity. On the other hand, this limitation is what the proposed procedure intends to overcome: if future analyses are performed by adopting the outlined steps, their results can be considered as "samples" of the same experiment, increasing the confidence on the observed relationships between workload and aging.
This knowledge can also be used for predicting purposes, e.g., to improve model-based solutions, or develop tools for aging estimation. Along this direction, next steps include: *i)* the execution of additional experimental campaigns on

different systems, in order to enrich the knowledge of aging-workload relationship, and to promote the comparability among systems; *ii)* the implementation of a tool for runtime TTE prediction, which, based on the defined workload parameters, monitors the runtime workload and evaluates aging trends by means of workload-dependent models; *iii)* the definition of benchmarking approach in which aging is one of the dependability aspects to consider.

References

[1] M. Grottke and K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. Computer Communications, 40(2):107–109, 2007.

[2] M. Grottke, L. Lie, K. Vaidyanathan, and K. S. Trivedi, Analysis of software aging in a web server, IEEE Trans. Reliability, vol. 55, no. 3, pp. 411–420, 2006.

[3] G. Carrozza, D. Cotroneo , R. Natella, A. Pecchia, S. Russo, Memory Leak Analysis of Mission-Critical Middleware. Journal of Systems and Software vol. 83, no. 9, 2010.

[4] E. Marshall Fatal Error: How Patriot Overlooked Scud. Science, p. 1347, Mar.1992

[5] A.Tai, S.Chau, L. Alkalaj and H.Hecht. On-board Preventive Maintenance: Analysis of Effectiveness an Optimal Duty Period. Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems, 1997

[6] D. Cotroneo, S. Orlando, S. Russo, Characterizing Aging Phenomena of the Java Virtual Machine. Proc. of the 26th Symp. on Reliable Distributed Systems, 2007, pp. 127–136.

[7] S. Garg, A. Puliato, K. Trivedi, Analysis of Software Rejuvenation using Markov Regenerative Stochastic Petri Nets. Proc. of 6th Intl. Symp. on Software Reliability Engineering, 1995.

[8] Y. Bao, X. Sun, and K. Trivedi, A Workload-Based Analysis of Software Aging, and Rejuvenation, IEEE Transactions on Reliability, vol. 54, no. 3, p. 541, 2005.

[9] K.J. Cassidy, K.C. Gross, A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. Proc. of the Intl. Confer. on Dependable Systems and Networks, 2002, 478–482.

[10] K. Vaidyanathan and K. Trivedi, A Comprehensive Model for Software Rejuvenation, IEEE Transactions on Dependable and Secure Computing, vol. 2, no. 2, pp. 124137, 2005.

[11] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, A Methodology for Detection and Estimation of Software Aging, in Proc. of the 9th Intl. Symp. on Software Reliability Engineering, 1998.

[12] K. Vaidyanathan and K.S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. Proc. of 10th International Symposium on Software Reliability Engineering, 1999, 84–93.

[13] Wang, Dazhi and Xie, Wei and Trivedi, Kishor S.. Performability analysis of clustered systems with rejuvenation under varying workload. Performance Evaluation, 247–265, 2007.

[14] R. Matias Jr and P. Freitas, An Experimental Study on Software Aging and Rejuvenation in Web Servers. Proc. of the 30th Intl. Computer Software and Applications Conference (COMPSAC), vol. 01, 2006, pp. 189196.

[15] L. Silva, H. Madeira, and J.G. Silva. Software aging and rejuvenation in a soap-based server. Proc. of 5th Intl. Symposium on Network Computing and Applications, 56 65, 2006.

[16] G. A. Hoffmann, K. S. Trivedi, and M. Malek. A best practice guide to resources forecasting for the apache webserver. In 12th IEEE Pacic Rim International Symposium on Dependable Computing, 183–193, 2006.

[17] D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, Software Aging Analysis of the Linux Operating System. Proc. of 21st IEEE Intl. Symp. on Software Reliability Engineering, 2010.

[18] R. Matias, Jr., Pedro A. Barbetta, K. S. Trivedi, P. J. Freitas Filho, Accelerated Degradation Tests Applied to Software Aging Experiments, IEEE Trans. on Reliability, 59 (1), 2010.

[19] Douglas C. Montgomery, Design and Analysis of Experiments. 5th edition.

[20] Pranab, Kumar, Sen, Estimates of the Regression Coefficient Based on Kendalls Tau, Journal of the American Statistical Association, Vol. 63, No. 324 (Dec., 1968), pp. 1379- 1389

[21] Yun-Fei Jia, Xiu-E Chen, Lei Zhao and Kai-Yuan Cai. On the Relationship between Software Aging and Related Parameters. Proc. of the 8th Intl. Conference on Quality Software, 2008

[22] A.Avritzer, A. Bondi, M. Grottke, K. Trivedi,. E.J. Weyuker. Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms, Proc. of the International Conference on Dependable Systems and Networks 2006

[23] M. Grottke, R. Matias Jr., K. S. Trivedi, The Fundamentals of Software Aging. Proc. of the 1st International Workshop on Software Aging and Rejuvenation/19th IEEE International Symposium on Software Reliability Engineering, 2008.