Memory Leak Analysis of Mission-Critical Middleware[☆]

G. Carrozza[a,c], D. Cotroneo[a], R. Natella[*,a,b], A. Pecchia[a,b], S. Russo[a,b]

[a]*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy*
[b]*Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed.1, Via Cinthia, 80126, Naples, Italy*
[c]*Consorzio SESM SCARL, Via Circumvallazione Esterna di Napoli, 80014, Giugliano in Campania, Naples, Italy*

Abstract

Memory leaks are recognized to be one of the major causes of memory exhaustion problems in complex software systems. This paper proposes a practical approach to detect aging phenomena caused by memory leaks in distributed objects Off-The-Shelf middleware, which are commonly used to develop critical applications. The approach, which is validated on a real-world case study from the Air Traffic Control domain, defines algorithms and ad-hoc support tools to perform data filtering and to find the best trade off between experimentation time and statistical accuracy of aging trend estimates. Experiments show that fixing memory leaks is not always the key to solve memory exhaustion problems.

Key words: software aging, memory leak analysis, mission-critical middleware
2008 MSC: 68M15

1. Introduction

The adoption of Off-The-Shelf (OTS) software items is becoming a rule to develop complex applications and systems. This is mainly due to the benefits they provide in terms of

[☆]This work has been partially supported by the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the framework of the private-public laboratory COSMIC ("Centro di ricerca sui sistemi Open Source per le applicazioni ed i Servizi MIssion Critical", DM23318, `www.cosmiclab.it`) and the project DOTS-LCCI ("Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures", DM1407, `dots-lcci.prin.dis.unina.it`). It has also been supported by the Italian research project "Iniziativa Software", which involves Finmeccanica company and several Italian universities (`www.iniziativasoftware.it`).

[*]Corresponding author (phone: +39-081676770, fax: +39-081676574)

Email addresses: `gcarrozza@sesm.it` (G. Carrozza), `cotroneo@unina.it` (D. Cotroneo), `roberto.natella@unina.it` (R. Natella), `antonio.pecchia@consorzio-cini.it` (A. Pecchia), `stefano.russo@unina.it` (S. Russo) development time and cost reduction, as they come as ready-to-use modules, which can be integrated, possibly customized, and put together to develop modular solutions. In spite of their well-known dependability pitfalls, they are commonly used in critical scenarios like avionics, air traffic control, and monitoring systems, where long-running and complex applications are required to provide stringent reliability, availability, and performance guarantees [1].

Dependability evaluation and improvement for OTS items, as well as for OTS-based complex applications, is still an open research issue. It is well-known that complex and OTS-based software systems may exhibit software failures, which go undetected during pre-operational testing activities [2, 3]. These failures are due to latent software defects, also known as *bugs*, i.e., faults activated by complex and sometimes unpredictable triggering conditions. As OTS modules are generally delivered as executable items (e.g., linkable binary objects for C/C++ programs), without their source code or internal documentation, bugs are difficult to locate and to treat.

Software aging phenomena occur due to the activation of aging-related bugs, which cause, for example, accumulation of round-off errors, memory bloating and leaking, unreleased file locks, data corruption, and storage space fragmentation. These errors accumulate over time and they are likely to degrade system performance and to increase the failure rate [4]. Due to its cumulative property, software aging occurs more intensively in long-running systems, such as web servers or daemon processes. It has been demonstrated that aging-related bugs represent a serious threat to mission-critical software systems, for the following reasons [5]. First, detecting and estimating the effects of these bugs requires complex experimental campaigns with unpredictable durations. It is hard to tune experimental campaigns in order to achieve accurate results. Second, it has been demonstrated that software aging strongly depends on workloads imposed on the system. Thus, each experimental campaign aiming to study aging phenomena must take into account relevant workload parameters to be imposed on the system.

In this paper, focus is on memory management bugs that underlie most of the software aging phenomena in complex systems [6, 7]. Our contribution is the definition of a systematic approach to locate memory leak sources and to detect aging trends due to memory leaks in CORBA OTS-based software systems. By using consolidated and already known methods and techniques, the approach goes beyond the state of the art in that it is practically applicable to a wide class of systems. During the approach definition we took into account fundamental milestones in software aging research studies, as well as some existing techniques for aging trend detection and estimation, such as Design-of-Experiments, Mann-Kendall test and Sen's Procedure. The novelty of the approach lies in the definition of novel algorithms and ad-hoc support tools able to perform data filtering, as well as to find the shortest time needed to obtain accurate results from experimental campaigns. The approach is evaluated on a real-world middleware, named CARDAMOM[1], for developing mission-critical applications in the field of Air Traffic Control. Experiments are conducted

[1]An open-source Community Edition (CE) version of this platform is available at `http://forge.objectweb.org/projects/cardamom`

by using a CARDAMOM version (Development Version, DV) developed in the framework of the COSMIC[2] Italian research project.

The rest of the paper is organized as follows. Section 2 discusses the related research and provides a brief background about CARDAMOM. The proposed analysis approach is described in Section 3, whereas Section 4 shows *where and how* to gather data related to memory usage within a running system. In Section 5 it is shown how the proposed approach, and the implemented support tools, can be used to detect and locate memory leaks. Key findings are discussed in Section 6, which closes the work.

2. Background and Related Work

2.1. Related work

Software aging has been widely addressed by researchers in the field of complex and distributed software systems. It has been shown—mainly by means of measurement-based approaches on operational systems—that aging-related bugs cause performance degradation and, even worse, hang or crash failures [8]. In fact, these approaches are the most effective way to deal with such failures that generally manifest during system execution in the field rather than during the testing phase.

A methodology for aging trend estimation was proposed in [6], aiming to compare aging effects on system resources. Through experiments on operational systems, the authors showed that free memory exhibits the shortest Time to Exhaustion (TTE) if compared to other system resources (e.g., processes or file table size). This was also confirmed by [7] in which resource usage in a web server subject to an artificial workload is studied. Kalyanakrishnam et al. [9] showed that Windows NT systems run out of virtual memory due to memory leaks in most of the cases. Cherem et al. [10] described a reachability problem to detect leaks in C programs, while Java leaks were faced in [11], which propose a container-based tracking technique to detect pending object references.

Application workload effects on aging were investigated by several research studies. In [12, 13], measurement-based models were proposed to estimate resource exhaustion rate as a function of time and system workload state. Workload is characterized by several parameters (e.g., context switches, system calls). A cluster analysis is subsequently used to identify workload states. Aging trends are extracted by analyzing sample data on resource usage (e.g., real free memory, service rate) for each workload state. Trends are finally used to populate models, e.g., reward rates attached to states in semi-Markov processes, and to obtain a more accurate TTE estimation.

A methodology for selecting workload parameters impacting on software aging, and for quantifying their effects in Java Virtual Machine (JVM) subsystems, was presented in [14]. The number and the time of Just In Time (JIT) compilations, the object allocation rate and mean size, and the duration of garbage collection are just a few of the identified system variables. Linear regression analysis on throughput loss and memory depletion shows that

[2]`http://www.cosmiclab.it`

software aging is mainly due to variables related to the operating system abstraction layer and to the JIT compiler. Matias and Filho [15] evaluated the software aging effects on a Web server as well as the effectiveness of rejuvenation techniques. The main contribution of this work was the experimental identification of the factors that contribute to software aging in the Web server. The Design-of-Experiments technique (DOE) is the means to characterize the aging phenomenon and to identify the parameters related to memory consumption (i.e., page type and page size).

As the above discussed research studies share the conclusion that memory management related defects (e.g., memory leaks) represent the most serious cause of aging and that much work still remains to be done, this work focuses on the characterization of memory usage for estimating aging trends and gathering insights for aging prevention.

2.2. CARDAMOM overview

The CARDAMOM middleware, compliant to the OMG standard specifications for distributed objects middleware, has been chosen as the target platform in the context of this work. It is a CORBA-based platform supporting both the object and the component programming models, which are used to develop mission-critical applications in the field of Air Traffic Control.

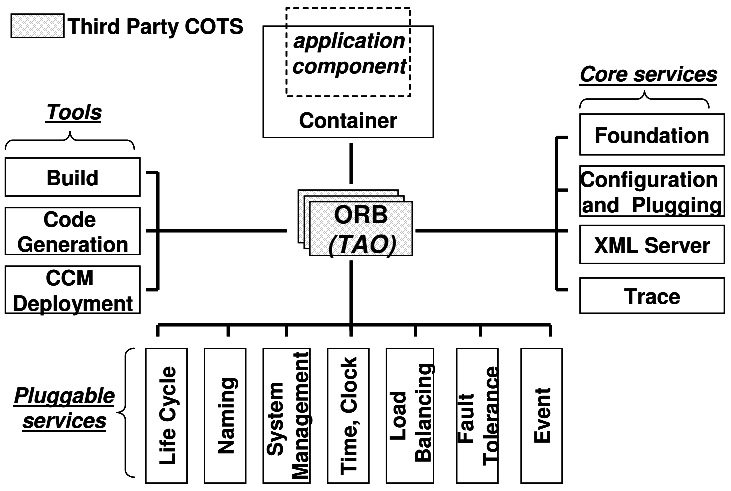


Figure 1: CARDAMOM overview.

Figure 1 shows a high-level view of the middleware. It is organized as a collection of core services, plus a set of pluggable services. Core services are needed to run a CARDAMOM system; the adoption of pluggable services depends on user needs. Support tools (e.g., for code generation) are also provided to simplify application development. CARDAMOM is based on OTS modules, from the Linux operating system to a third-party Object Request Broker (ORB) named TAO[3], along with other facilities (e.g., an XML parser, a thread

[3]`http://www.theaceorb.com`

programming library). In particular, aging investigation in this work reveals interesting findings about the TAO ORB.

3. Proposed Approach

The proposed approach aims to characterize software aging and its impact on the performance of CORBA-based mission-critical middleware. These provide services through remote methods invocation, i.e., a client invokes methods implemented by a remote server, which can be time and memory consuming. Performance can be characterized by the method invocation time (i.e., the Round Trip Time, RTT), and by the amount of memory required for executing operations (Memory Consumption, MC). Response time and system resource usage are widely recognized to be relevant metrics for performance analysis in computer systems [16], thus we focus on RTT and MC for performance characterization.

Due to the workload influence on system performance, workload parameters which are likely to affect RTT and MC have to be identified. To this aim, we consider the invocation period, T, as well as the amount of data exchanged during the method invocation, L. We ignore data type, since the Common Data Representation (CDR) format is adopted by the middleware to transfer both primitive and structured data as a vector of byte-size elements.

The ultimate aim of the proposed approach is to understand *whether and how* the selected workload parameters can impact on aging trends. This understanding can be helpful for:

- Designers and maintainers, who want to use estimated trends to populate analytical models for computing the best rejuvenation schedule, as aging trends depend on workload states [12, 13].
- Developers, who want to find software components responsible for aging [14].

A key aspect to take into account when dealing with aging is the experimentation time, which can actually impact on trend estimates. We investigate how to optimize the scheduling of experimental campaigns in order to achieve (a) high quality results in terms of statistical accuracy of trend estimates and, (b) the shortest experimentation time. Obviously, these are conflicting metrics, in that the shorter the observation time the less accurate the memory consumption analysis, and the trend estimate as well. To find a good trade-off between time and accuracy we propose a multi-step approach, sketched in Figure 2.

Step 1 - Experiments setup. For each workload parameter we choose a set of candidate values. Let $T \in O_T = \{t_1, t_2, \ldots, t_n\}$ with $t_1 > t_2 > \ldots > t_n$ and $L \in O_L = \{l_1, l_2, \ldots, l_m\}$ with $l_1 < l_2 < \ldots < l_m$ be these sets. Along with them, let us introduce $T_{\max}$ and th_c. The former is the maximum time allowed for preliminary experiments (see Step 2) and it is upper bounded by the system mission time, T_M. The latter, instead, is the maximum allowed RTT variation. Experiments can be designed once the sets O_T and O_L have been established. In order to discard meaningless combinations, or to reduce the size of the combination set $F = O_T \times O_L$, system analysts can start a selection process as suggested in [17, 15], where the Design-of-Experiments technique has been proposed to

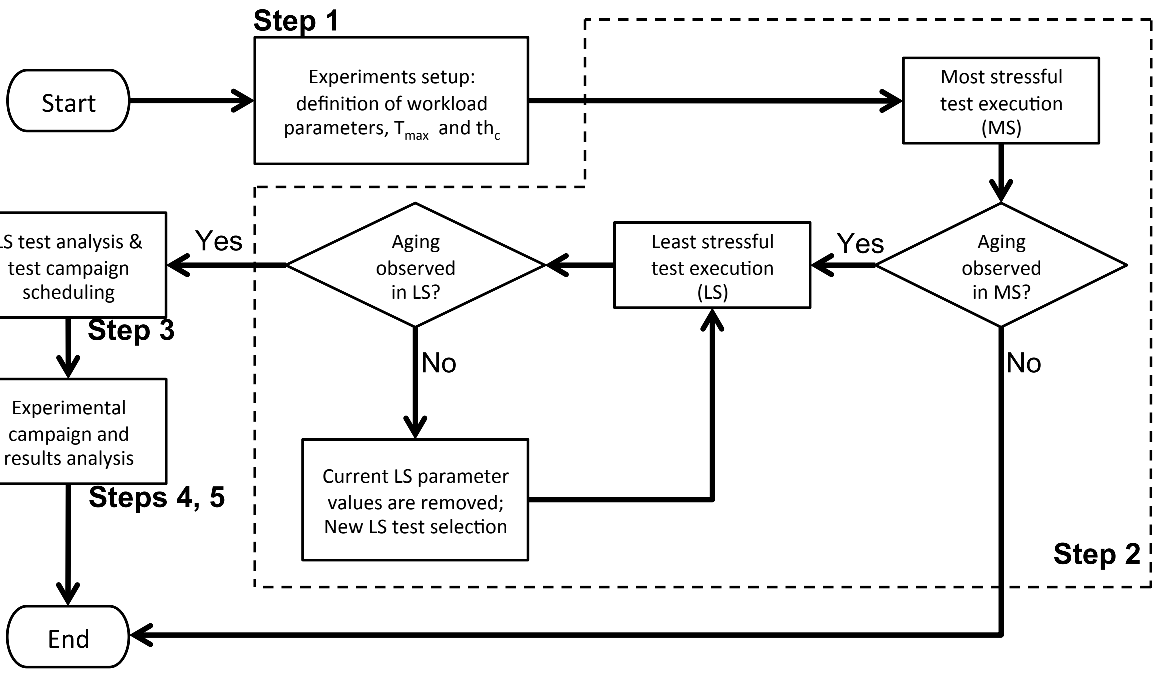

Step 1
Start
Experiments setup: definition of workload parameters, T_{max} and th_c
Most stressful test execution (MS)
Aging observed in MS?
Yes
No
Least stressful test execution (LS)
Aging observed in LS?
Yes
No
Current LS parameter values are removed; New LS test selection
LS test analysis & test campaign scheduling
Step 3
Experimental campaign and results analysis
Steps 4, 5
Step 2
End

Figure 2: Approach overview.

identify the set of combinations expressing significant parameter interactions. Let $S \subseteq F$ be this set.

Step 2 - Preliminary experiments. Stress testing is a widely used practice for aging analysis of complex applications, since software aging is more likely to manifest itself in stressful conditions [7, 15, 14]. In our case, stressful workload means high resource consumption (e.g., CPU usage, memory), hence, a greater probability of leaks. For this reason, we find it reasonable to assume that the less stressing the workload, the lower the trend. This is confirmed by previous results. In [18], memory pressure and aging trends increase with the number of clients, while in [7] the connection rate limit of a web server was deliberately exceeded to accelerate software aging. However, in our approach the analyst has to complete experiments prior to claim that the selected parameters actually have an impact on aging, and at what extent.

Let LS (Least Stressful) and MS (Most Stressful) be the combinations of parameter values that generate the least and the most stressing tests, respectively. They can be expressed as $\overline{p}_{LS} =< t_1, l_1 >$ and $\overline{p}_{MS} =< t_n, l_m >$.

First of all, we have to understand whether the system is actually affected by aging. This is the aim of the MS test that we run at the beginning of the experimental campaign. Among the performance indicators previously mentioned, we focus on RTT, since it is the most relevant metric in critical scenarios with performance requirements, which we consider in this work [1]. During the test we periodically sample RTT and we estimate the average

RTT variations over time. More in detail,

1. RTT average value h_i is computed each H hours;
2. If h_i is th_c times greater than h_1, aging is detected and the test is stopped; otherwise, sub-steps 1 and 2 are repeated;
3. Test is stopped when $t = T_{\max}$.

If sub-step 3 is performed, i.e., aging did not manifest during the test, investigation stops as performance is not compromised during the system mission time. Conversely, the following steps take place if aging was observed within $T_{\max}$.

The LS test is used to establish how long a test should be in order to get the evidence of aging trends. Hence, we look for a test duration D that allows to reveal the presence of aging trends, with a *good accuracy*, for all the tests that are going to be launched. Similar to the MS test, aging detection is based on RTT sampling and analysis. More precisely, LS aims to discard those parameter combinations which do not cause aging trends, and to rearrange O_T and O_L. If $\overline{p}_{LS} =< t_i, l_i >$ does not cause aging, t_i and l_i are removed from O_T and O_L, and the test $< t_{i+1}, l_{i+1} >$ is executed. This procedure is repeated until an aging trend is detected. If $n \neq m$, it may occur that LS experiments $< t_i, l_i >$, $i = 1 \dots q, q = \min\{n, m\}$, do not exhibit aging. In this case, the parameter with the lowest number, namely q, of candidate values most probably has no influence, since there is no aging for test $< t_q, l_q >$. Therefore, subsequent LS experiments are performed by varying only the remaining parameter (for $i = q + 1 \dots \max\{n, m\}$).

Response time and memory consumption measurements are taken during the LS test and will be used in the next step.

Step 3 - Estimation of the minimum experiment duration. In order to minimize the time needed for completing experiments, it is desirable to minimize the duration of tests. We aim to determine the minimum time, D, that allows a statistically accurate aging trend estimate for all the tests and for both RTT and MC. To this aim, we apply Algorithms 1 and 2 (detailed in Section 5) on each data set (samples of performance and memory consumption that were collected during the LS test, respectively). Let us denote by ε the error margin, which is set by system analysts. To be conservative, we set $D = \max(D_{\mathrm{RTT}}, D_{\mathrm{memory}})$ where D_{RTT} and D_{memory} are the estimated durations related to RTT and MC, respectively.

Step 4 - Experimental campaign. The experimental campaign consists of as many stress tests as the number of elements in S (see Step 1). Each test lasts for D time units, hence aging trends can be estimated with a known margin of error for all the tests. During tests, RTT and MC measurements are taken for further analysis.

In order to quantify the actual time reduction we get by using the procedure described above (let us call ρ this reduction), we compare the actual experiments time to the time which would have been required if all the test were executed for $T_{\max}$ time units ($d_{\mathrm{comparison}} = |S| \cdot T_{\max}$). The total duration of the experimental campaign (d_{total}) can be expressed as:

$$d_{\text{total}} = d_{MS} + \sum_{i=1}^{k} d_{LS_i} + (|S_k| - 1) \cdot D \tag{1}$$

in which d_{MS} is the duration of the MS experiment, d_{LS_i} (for $i = 1 \ldots k$) is the duration of an LS experiment, and k represents the number of LS experiments that did not reveal aging trends. Note that MS is performed just once, as opening test. We also exclude all the experiments from S in which the workload do not affect software aging (i.e., all parameter combinations that contain values removed from O_T and O_L during the LS experiments, as explained in Step 2). Let S_k be the set of remaining experiments, with $|S_k| \leq |S|$.

Let us observe that the following statements hold in the worst case:

1. MS and LS experiments last for $T_{\max}$;
2. $|S_k| = |S| - k$, when only the LS experiments are removed from S.

This means that d_{total} is upper bounded by:

$$\begin{aligned} d_{\text{total}} &\leq T_{\max} + k \cdot T_{\max} + (|S| - k - 1) \cdot D && (2)\\ &= (k+1) \cdot T_{\max} + (|S| - (k+1)) \cdot D && (3)\\ &\leq (k+1) \cdot T_{\max} + (|S| - (k+1)) \cdot T_{\max} && (4)\\ &= |S| \cdot T_{\max} = d_{\text{comparison}} && (5) \end{aligned}$$

Hence, $\rho = d_{\text{comparison}} - d_{\text{total}}$ depends on the number $|S| - |S_k|$ of tests in which software aging is negligible, as well as on the difference $T_{\max} - D$. Since we expect $T_{\max} \gg D$ in the presence of software aging, ρ is expected to be significant.

Step 5 - Analysis. Collected samples are processed to perform statistical analyses. As previously mentioned, our main focus is to understand how the aging trend varies with respect to the workload parameters. However, during this phase, we also perform additional investigations. Memory consumption due to memory leaks is studied by means of the MELANY support tool (see next Section), to identify software items responsible for the leaks. Existing statistical techniques for data analysis (e.g., ANOVA) are also used to gain further insights.

4. Characterization of Memory Usage

4.1. MELANY overview

Memory consumption in complex systems is not an easy process to monitor and to understand. To have a complete view of how memory is getting used, data coming from different levels have to be merged, thus complicating the data collection and analysis process. The presence of heterogeneous data, differing in format and information content, induced us to develop a support tool, named MELANY, to perform analysis of memory consumption for complex C++ applications in Linux environments. The tool is able to manage data coming from (i) the Operating System (OS) and, (ii) Valgrind, i.e., an open source binary instrumentation tool[4]. Data coming from Valgrind, concerning runtime memory leaks, are hard to manage and difficult to read due to the unfriendly text report format. More important, as they only concern memory leaks, they do not allow characterizing memory consumption exhaustively. To overcome these limitations, MELANY integrates Valgrind reports with OS logs and provides a user friendly GUI for data management.

Memory consumption is not only due to memory leaks. Indeed, it also arises from those memory regions that are intentionally and properly used by running programs (e.g., memory regions referenced by valid C/C++ pointers). Hence, we say that Total Memory Consumption (TMC) is the sum of Memory Definitely Lost (MDL), i.e., unreferenced regions due to memory leaks, plus Intentional Memory Consumption (IMC). The OS provides information about TMC, while MDL data come from Valgrind. The main contribution of the tool is to isolate different contributions to TMC and to provide an aggregated value at the same time.

The MELANY data processing approach is summarized in Figure 3. MELANY processes data collected during experiments (see Section 3). These are stored into the MELANY database, and then used to produce statistics and reports about application memory usage.

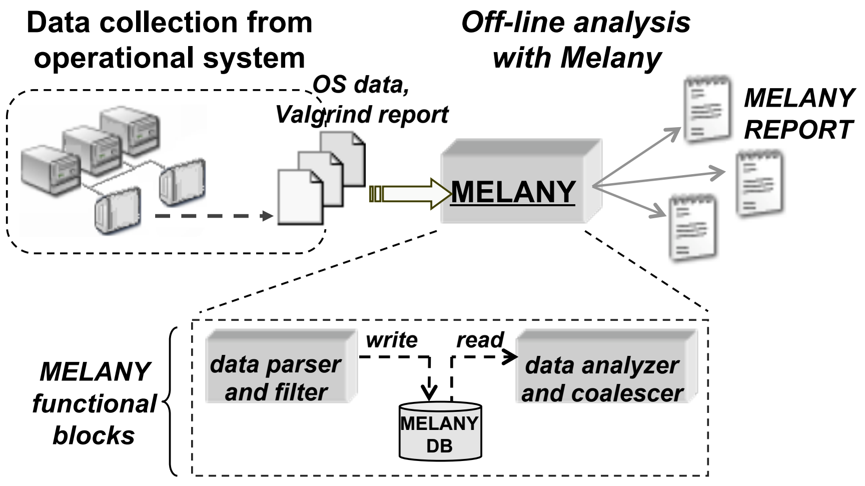


Figure 3: Data collection and analysis.

4.2. Information sources

Modern OSs use advanced memory management mechanisms, which complicate the analysis of memory consumption data; in particular:

1. *Demand paging*: in demand paging systems, the OS actually reserves physical memory when a page is referenced by processes. Hence, memory allocated by a process and physical memory actually used by it can differ in size.

[4]http://valgrind.org

2. *Shared memory*: several processes can share physical memory regions. Sharing can be either explicit (e.g., through Inter Process Communication mechanisms such as UNIX shared memory), or implicit (e.g., the OS automatically configures page tables of processes to share program text area and shared libraries).
3. *Virtual memory*: the memory at disposal of processes is the physical memory plus swap space.

The actual memory footprint of a process is the sum of physically allocated memory (Resident Set Size, RSS) plus swapped out memory. This equals RSS if there is no swap space. Additionally, the presence of shared memory regions makes the overall memory occupation less than the sum of processes memory footprints.

For this reason, we calculate memory consumption by considering free memory instead—which is easy to determine (e.g., by means of *free* UNIX utility)—as the reference metric: the lower the available memory, the greater the memory consumption. Note that OSs use free physical memory for filesystem caching purposes. However, it can be considered as available memory since it is a resource that processes can preempt.

The size of a virtual address space, which depends on the CPU and on the running OS, also impacts on user-space process allocation. In a standard Linux OS running on 32-bit x86 architecture[5] no more than 3 GBs are available for each process[6]. Memory that can be allocated by a process is, at most:

$$M = \min(\textit{address space size}, \textit{virtual memory available}) \tag{6}$$

Memory exhaustion occurs when either address space size is exceeded or there is no more virtual memory for the running process, which is then killed (assuming that the OS does not kill any other process to free virtual memory).

Valgrind output data are formatted as XML reports. We enable memory-usage monitoring (by using the *memcheck* tool) in order to find errors in *syscall* parameters, e.g., a not addressable memory buffer, as well as memory leaks. These can be further classified in:

- *definitely lost bytes*: the application is causing a leak and it has to be fixed;
- *possibly lost bytes*: it is very likely that the application is going to cause a leak since it is making complex use of pointers.

4.3. MELANY output

Once data have been collected, MELANY performs the following tasks (Figure 3):

1. data parsing;
2. data filtering, to discard redundant or meaningless data;
3. data storing on the MELANY database:

[5]The case addressed in this work.

[6]The fourth gigabyte is reserved by the Linux OS for kernel-level virtual memory mappings [19].

- Valgrind logs provide the error type, corrupted memory and error source, which are stored for each memory error.
- OS logs give information about TMC and free memory.

MELANY uses data coalescence to build aggregated results from individual memory leak entries. For instance, leaks are grouped by source library, in order to evaluate its contribution with respect to the total leaked memory.

```
Type: Leak Possibly Lost
Description: 20 bytes in 1 blocks are possibly lost in loss
        record 465 of 3,280
Cause: operator new(unsigned)
Sources:
        1) Method: main; Library: client.cpp; Line 85
        2) Method: Cdmw::OrbSupport::OrbSupport::ORB_init
        3) Method: Cdmw::OrbSupport::OrbTaoImpl::ORB_init
        4) Method: Cdmw::OrbSupport::AceTaoLogger
        5) Method: std::string::string
        6) Method: unknown; Library: /usr/lib/libstdc++.so
        7) Method: std::string::_Rep::_S_create
```

(a) Leak stack trace.

```
Library: /tools/exec/TAO141_2_06_0913/src/TAO/tao/libTAO.so.1.4.1
Error occurrences: 12
        Root Source 'client.cpp':               91%
        Root Source '/lib/tls/libc-2.3.4.so'     8%
```

(b) Source report entry.

```
Process name:        demo_client
Process PID:         5859
Initial sample:      1
Final sample:        36
Period:              5 min
Initial TMC:         68,184 KB
Final   TMC:         229,268 KB
Initial MDL:         0,000 KB
Final   MDL:         72,633 KB
Initial IMC:         68,184 KB
Final   IMC:         156,635 KB
```

(c) Integrated report.

Figure 4: Examples of MELANY reports.

As for reports, MELANY provides the following types of report:

1. BASE REPORT, providing basic process information (e.g., the total amount of allocated memory), and the complete error list;
2. DETAILED REPORT, providing additional information, such as the invocation stack for each reported error. An example is shown in Figure 4a.
3. SOURCE REPORT, providing the number of errors, the root causes of errors, and the percentage of each root cause on the errors amount, for each library that caused

a leak. Figure 4b shows a single entry of this report; it shows the case of an error located in *libTAO.so.1.4.1*, which occurred 12 times.

4. INTEGRATED REPORT, to isolate the IMC and MDL contributions to TMC (see Figure 4c).

Reports provide information at different levels of detail. For example, for each occurred memory leak, invocation stack and the responsible source object are reported. Again, reports can be generated to identify how many leaks have been caused by the same object or function. Reports bring benefits to both developers and system analysts. The former ones are supported in detecting the locations of memory management bugs in the source code. The latter ones are helped in estimating TTE, which is crucial for an easier and more effective rejuvenation planning. This is especially helpful when leaks are due to bugs in OTS items, e.g., software libraries or other runtime supports, for which source code fixing is precluded, making rejuvenation strategies the only way to prevent aging.

5. Case Study

We apply the described approach to a CARDAMOM case study application. First we describe the application along with its components, and then we present the results of each step of the approach.

The test application, running on Linux OS, uses both basic and pluggable services (see Figure 1), since we aim to detect memory leaks introduced by the middleware. It has a client-server structure, and it makes extended use of CARDAMOM services in the form of shared libraries (see Section 4 for details about process monitoring). The client forwards requests to two load-balanced server processes; requests are processed according to a round-robin policy (Figure 5). Requests consist of remote method invocations to the server. In order to easily control L during the campaign, we make the server replies to be returned as strings. Both the client and the servers are deployed on the same machine, in order to minimize the effects of network fluctuations on application performance, and they interact only through standard BSD sockets. Furthermore, application code was accurately analyzed to make it free from memory management bugs.

Experiments have been executed on testing machines equipped with 2 Xeon Hyper-Threaded 2.8 GHz CPUs, 5 GB of physical memory, and without swap space[7]. The Linux kernel version 2.6.25 has been used.

5.1. Experiments setup (step 1)

We assume as ranges for the two workload parameters $T \in O_T = \{300, 400, 600, 800, 1000\}$ ms and $L \in O_L = \{1, 10, 100, 1000\}$ bytes, respectively. The parameters of preliminary experiments are $T_{\max} = 100$ *hours* and $\text{th}_c = 1\%$. These values take into account real world traces of the Air Traffic Control domain where CARDAMOM is used as support middleware[8].

[7]This is in order to simplify memory footprint measurements.

[8]Traces have been provided by industrial partners involved in the COSMIC project.

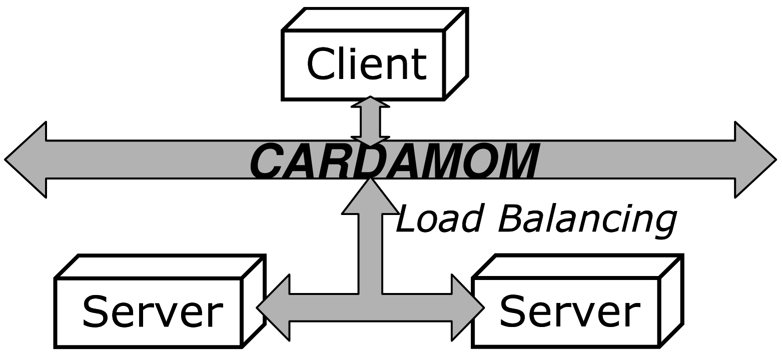


Figure 5: Testing application.

5.2. Preliminary experiments (step 2)

Parameters driving preliminary experiments are shown in Table 1. The MS experiment highlights (i) performance decrease and, (ii) the crash of the Client process after 65 hours. A post-mortem analysis of application and OS logs revealed that the process used up the all available virtual memory, and that it was killed by the OS. Further details about process killing are provided in Sections 4 and 5.5.2. Since the application code is very simple and it was accurately analyzed to make it fault-free, memory consumption within the process should be due to memory management bugs in external libraries. This finding allows us to run the LS experiment.

Table 1: Parameters used in the case study.

$\overline{p}_{LS}$	$\overline{p}_{MS}$	H
< 1000 ms, 1 byte $>$	< 300 ms, 1000 bytes $>$	15 hours

Figure 6 shows how the estimated RTT varies during the LS experiment. It reveals the presence of a software aging trend after 75 hours of execution. We used a windowing approach for computational and storage purposes. In particular, RTT samples within a 5 minutes time window are coalesced to their mean RTT value. This time window has been used by several studies in this field, such as [7, 6]. Additionally, we experience that choosing a time window of 5 minutes does not affect the accuracy of the results.

Trend detection has been performed by means of the Mann-Kendall test [20]. The Mann-Kendall test computes the value of a numerical index W, namely a *test statistic*, over a set of samples. The conditional probability that the test statistic takes values equal to or greater than the computed value, given that the null hypothesis of no trend is correct, is compared to a type I error level provided by the analyst. If the probability is lower than this type I error level, then the null hypothesis is considered unlikely and it is rejected (i.e., a trend has been detected).

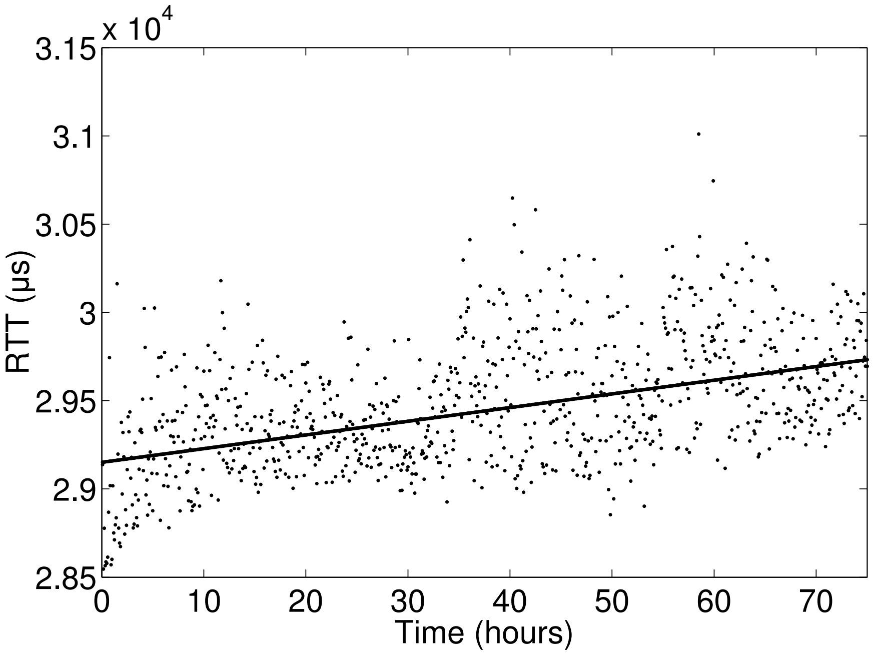


Figure 6: RTT samples and Sen's regression.

Once the trend has been detected with high confidence (we will assume 95% confidence in the following), it is estimated by means of the non-parametric Sen's procedure [21], which computes the median slope of all pairs of data points. We use this procedure because it is *robust*, i.e., it does not assume normally distributed measurement errors, and *resistant*, i.e., it is not sensitive to outliers. We achieved a slope equal to 0.1292 μs/min in the interval of values $[0.1136, 0.1449]$ μs/min with a 95% confidence interval.

During the LS experiment we also investigated how both the client and the servers managed memory, i.e., we monitored their RSS during the execution (see Section 4.2). We periodically sampled RSS every 5 minutes. About 900 samples have been collected for each process. Figure 7 shows the RSS with respect to the execution time. Total increase in memory consumption amounts to $\Delta = 501,092 = 569,276 - 68,184$ KB, even if the application did not allocate memory explicitly. Since memory consumption grows almost linearly, we hypothesize the existence of memory management defects that constantly increase memory consumption at each method invocation, and that should be pinpointed by means of precise memory usage analysis. We found the slope of the memory consumption trend to be 105.417 KB/min, within the $[105.416, 105.418]$ KB/min range, at client side (with a 95% confidence limit). A similar finding has been achieved at server side.

Memory consumption is so significant that heap increases and fragments. This requires more frequent and slower operations by memory allocation procedures, resulting in a performance decrease of application processes. The relationship between the amount of allocated memory and performance has been demonstrated by past works [22, 23]. We exclude other aging sources (e.g., file tables, unreleased sockets and locks) since we observed that their usage is not significant in our case study. As a result, it is reasonable to claim that *the experienced performance loss is caused by the memory depletion trend.*

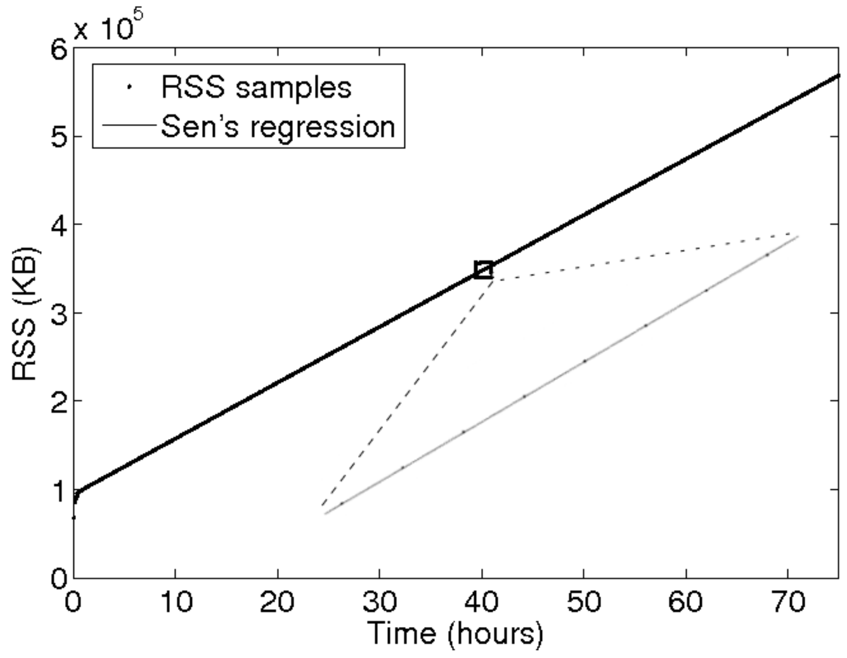


Figure 7: RSS samples and Sen's regression on client side.

The above-described experiment lasted more than three days, and then we were able to point out the existence of a trend for both RTT and RSS. However, such a long time is unacceptable when performing a large number of tests. For this reason, we improve the effectiveness of the experimental campaign by estimating the bare minimum execution time needed to provide an accurate trend estimate.

5.3. Estimation of the minimum experiment duration (step 3)

Collected data reveal that during the starting phase of each experiment the estimated trend is greater than expected. This has also been observed by some previous works, showing that software processes can exhibit high memory consumption at the application start-up, which is prone to be misinterpreted as software aging [15, 24]. For example, a complex long-running application could pre-allocate memory for key data structures that will be used during the execution (e.g., pools of connections and threads, shared libraries, caching and data prefetching from filesystems). This phenomenon is a transitory phase that could bias trend estimate if taken into account. Hence, it is crucial to discard samples collected during this phase. This is especially true when the execution time is short, i.e., the steady state does not last long enough to absorb transient effects. Since we want to reduce experimentation time, transient samples (i.e., samples collected during the transitory phase) have to be filtered out before estimating the bare minimum time for a single test. To this aim, we formalize a two-step procedure:

- ***Step 3.1:*** Estimation of the transitory phase duration, and removal of transient samples;

- ***Step 3.2:*** Estimation of the bare minimum time needed to achieve an accurate trend estimate, once the transitory phase has ended.

Step 3.1. In the literature, there are several algorithms for transient detection, developed in the context of queueing systems simulation. For instance, algorithms discussed in [25, 26] aim to estimate *steady-state mean values*; in [27], the same algorithms are used for estimating *steady-state quantiles*. To the best of our knowledge, transient-detection algorithms have never been studied for estimation of *trends* in data series. In fact, existing algorithms cannot be used for transient detection when estimating a trend, because they assume a stationary probability distribution of samples (which is not the case for aging measurements). To this aim, we implemented Algorithm 1 for transient detection. We take advantage of existing trend estimation algorithms (e.g., Sen's procedure) for transient detection in aging measurements, which has never been made in past works.

Let samples($x_1 \dots x_{\text{last}}$) be the collected samples (in the following, we will refer to the index of the ith sampled value as x_i, and to the sampled value as samples(x_i)). For each x_i in $\{x_1 \dots x_{\text{last}}\}$ the algorithm splits collected samples in 2 sets, samples($x_1 \dots x_{\text{i}}$) and samples($x_{\text{i}} \dots x_{\text{last}}$), respectively (these expressions represent sets of samples between two indexes). Trend estimation is performed for each set. If a statistically significant trend exists for both sets, trends are compared. The x_i where the trends difference is the highest is assumed to be the end of the transitory phase. Since all the samples before x_i exhibit a trend significantly different from subsequent ones, they are affected by transient effects and should be filtered out. Figure 8 depicts such a circumstance. The arrow indicates the *breakpoint*, i.e., the point where the greatest difference between line slopes has been measured. The Mann-Kendall test is used to check if a trend actually exists before and after x_t. If the algorithm does not find any suitable x_t (i.e., $\neg\mathcal{S}_1$ or $\neg\mathcal{S}_2$), then we assume that there were no significant aging effects during the experiment.

Algorithm 1 works also in the case that trends do not increase monotonically, or that the most significant difference exists in several points. If trends do not increase monotonically, but fluctuate over time (e.g., the trend decreases for a short period, then it increases again), the algorithm will stop only when the most significant difference between trends is found, thus neglecting trend fluctuations. If the most significant difference exists in several points, the first point is considered as the end of the transient period; further trend fluctuations may not be due to transient phenomena, therefore they are not considered to be part of the transient phase.

Step 3.2. Algorithm 2 has been designed to estimate the shortest interval $[x_{t+1} \dots x_d]$, for which (i) there exists a significant trend, and (ii) the estimated slope is *close enough* to the reference slope, i.e., the one estimated over the whole LS experiment (75 hours in our case study). Algorithm 2 can be applied with any choice of the period of the reference slope. In the algorithm, the ε parameter represents the estimation error tolerated by the user. There is a trade-off between test duration and estimation accuracy. The algorithm stops when the slope does not fall within the range $T_{\text{ref}} \pm \varepsilon \cdot T_{\text{ref}}$. Algorithm 2 detects if

```
**Data**: samples($x_1 \dots x_{\text{last}}$)
**Result**: Transient phase end $x_t$
$\mathcal{S}_1$: There exists a statistically significant trend between $x_1$ and $x_i$
$\mathcal{S}_2$: There exists a statistically significant trend between $x_i$ and $x_{\text{last}}$
max_diff $\leftarrow 0$
$x_t \leftarrow 0$
**for** $x_i \in \{x_1 \dots x_{\text{last}}\}$ **do**
    **if** $\mathcal{S}_1 \wedge \mathcal{S}_2$ **then**
        $T_1 \leftarrow$ TrendEstimation(samples($x_1 \dots x_i$))
        $T_2 \leftarrow$ TrendEstimation(samples($x_i \dots x_{\text{last}}$))
        **if** $|T_1 - T_2| >$ max_diff **then**
            max_diff $\leftarrow |T_1 - T_2|$
            $x_t \leftarrow x_i$
        **end**
    **end**
**end**
```

Algorithm 1: Algorithm for estimation of the initial transient period (step 3.1).

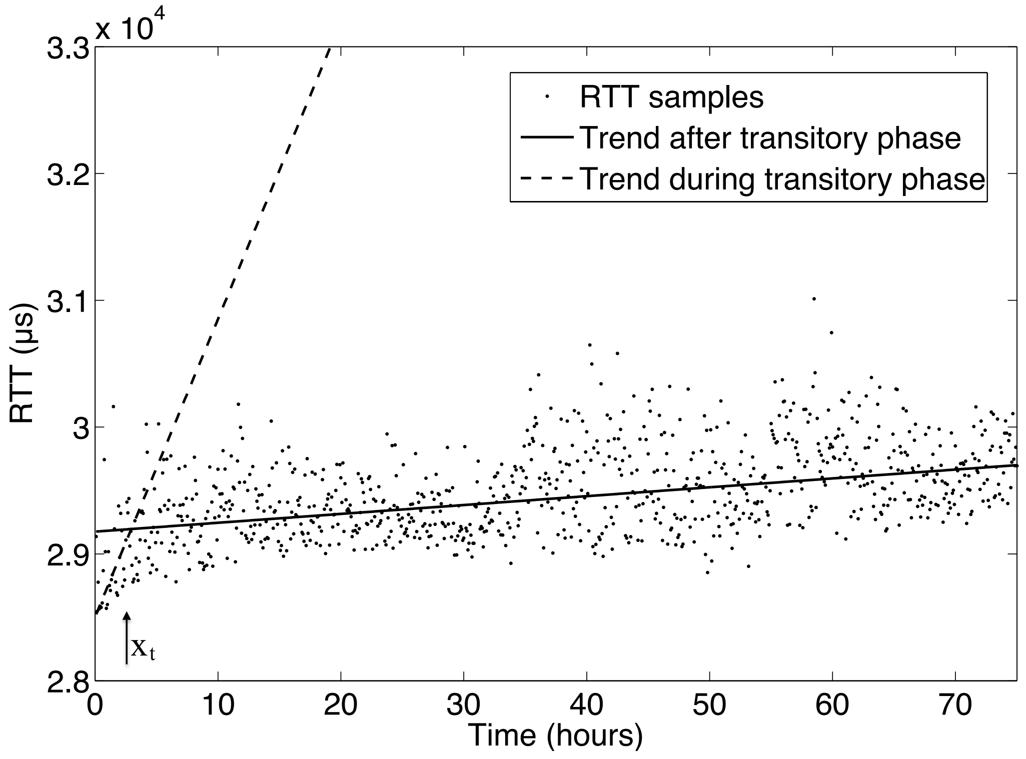


Figure 8: Estimated slopes for RTT during and after the transient phase.

```
Data: samples($x_1 \dots x_{\text{last}}$); $x_t$
Result: Test duration $x_d$
$\mathcal{S}_1$: There exists a statistically significant trend between $x_t$ and $x_i$
$T_{\text{ref}} \leftarrow$ TrendEstimation(samples($x_t \dots x_{\text{last}}$))
$T \leftarrow 0$
$x_i \leftarrow x_{\text{last}}$
if $\neg\mathcal{S}_1$ then
    Notify: Input data is not enough to obtain an accurate trend
    $x_d \leftarrow$ NIL
else
    repeat
        $T \leftarrow$ TrendEstimation(samples($x_t \dots x_i$))
        $x_i \leftarrow x_{i-1}$
    until $x_i = x_t \vee \neg\mathcal{S}_1 \vee |T - T_{\text{ref}}| > \varepsilon \cdot |T_{\text{ref}}|$ ;
    $x_d \leftarrow x_i$
end
```

Algorithm 2: Algorithm for estimation of stress test duration (step 3.2).

collected samples are enough to obtain an accurate trend estimate; the initial test in the Algorithm also assures that $T_{\text{ref}} \neq 0$.

Algorithms 1 and 2 do not ensure that a detected trend is a performance degradation or resource depletion trend. However, in the proposed case study the detected trends were actually due to the aging phenomena, as discussed in Section 5.2.

Table 2 summarizes results achieved for both RTT and RSS under low stress condition. The experienced durations can be assumed to be upper bounds, since, in our case study, the more intensive the workload, the faster the trend detection.

Table 2: Transient phase and minimum test duration estimates obtained from algorithms 1 and 2.

	Transient phase duration	**Minimum test duration**	**Margin of error (ε)**
RSS	0.41 hours	2.5 hours	0.01
RTT	1.58 hours	25.41 hours	0.15

The influence of transient samples on the estimation accuracy can be appreciated in Tables 3 and 4. These Tables show trends obtained with the transient phase included in the samples in leftmost column. In the first row, there is the aging trend estimated over the maximum test duration (i.e., 75 hours); in the second row, there is the aging trend obtained by analyzing samples within the minimum test duration (e.g., 2.5 hours for RSS). The last row shows the percentage difference between trends obtained after maximum and minimum test durations. In the rightmost column, the same comparison between maximum and minimum duration is made, in case that the transient phase is excluded from the samples. The difference between trends is greater when the transient phase is not filtered; this is especially true in the case of RTT, where the difference drops from 63.12% to 9.97% when transient phase is filtered. This demonstrates that, when test duration is reduced to its minimum, filtering transient samples is crucial to keep the estimation error low.

Table 3: RSS trend estimates (KB/min) with 95% confidence interval for different test durations and transient hypotheses.

	With transient phase	**Without transient phase**
Maximum duration (about 75 h)	$1.05417 \cdot 10^2$ $[1.05416, 1.05418] \cdot 10^2$	$1.05417 \cdot 10^2$ $[1.05416, 1.05418] \cdot 10^2$
Minimum duration (about 2.5 h)	$1.0970 \cdot 10^2$ $[1.0686, 1.1619] \cdot 10^2$	$1.0640 \cdot 10^2$ $[1.0533, 1.0834] \cdot 10^2$
Difference	+4.06%	+0.93%

Table 4: RTT trend estimates (μs/min) with 95% confidence interval for different test durations and transient hypotheses.

	With transient phase	**Without transient phase**
Maximum duration (about 75 h)	$1.2918 \cdot 10^{-1}$ $[1.1365, 1.4488] \cdot 10^{-1}$	$1.2028 \cdot 10^{-1}$ $[1.0485, 1.3610] \cdot 10^{-1}$
Minimum duration (about 25 h)	$2.1074 \cdot 10^{-1}$ $[1.3083, 2.8499] \cdot 10^{-1}$	$1.3316 \cdot 10^{-1}$ $[0.5627, 2.1350] \cdot 10^{-1}$
Difference	+63.12%	+9.97%

5.4. Experimental campaign (step 4)

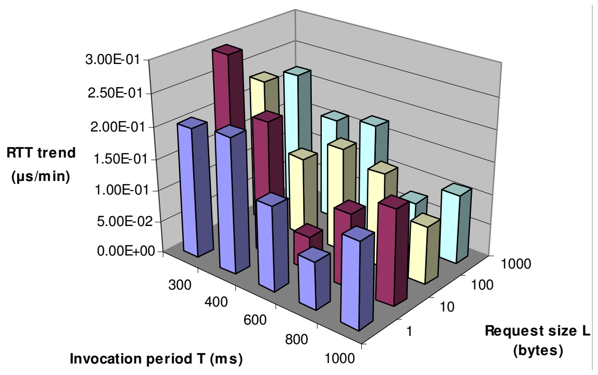
We executed the experimental campaign for all combinations of workload parameters. We applied a full factorial design of experiments ($S = F$) [17] since the number of combinations is low ($|F| = 20$). Each experiment lasted $D = 25.41$ hours, as computed in the previous step. This value is the maximum of the test durations for the two performance indicators (i.e., RSS and RTT). D is lower than $T_{\max}$; the total experimentation time is $d_{\text{MS}} + d_{\text{LS}} + D \cdot (|F| - 2) = 597.38$ hours, which is lower than $T_{\max} \cdot |F| = 2000$ hours.

5.5. Analysis (step 5)

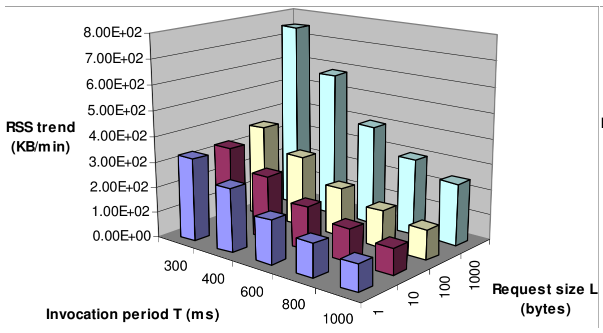
5.5.1. Aging trends analysis

We analyze how memory consumption varies with respect to the workload parameters, T and L. In Figure 9a, RTT trends exhibit a greater variability than RSS (Figure 9b and

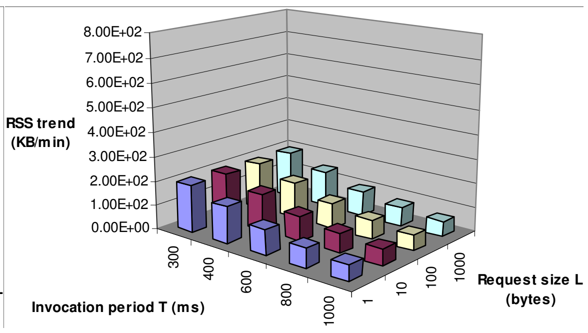
9c), due to the greater uncertainty associated to RTT estimates. In fact, for RTT we chose a greater margin of error than for RSS ($\varepsilon = 15\%$, Table 2), in order to keep the test duration low.



(a) RTT trend.



(b) RSS trend for the Client process.



(c) RSS trend for the Server process.

Figure 9: Aging trends, with respect to the request size (L) and invocation period (T).

We use the ANalysis Of VAriance (ANOVA) [17] to detect relationships between workload parameters and aging trends, if any, with a given confidence. ANOVA makes it possible to quantitatively analyze the impact of a workload parameter on the aging trend. It compares the variability of all runs to the sum of the variability within each sample and the variability between samples (in our case study, a sample is a group of runs with the same value of the workload parameter under evaluation), that is:

$$variability_{\text{total}} = variability_{\text{withinsamples}} + variability_{\text{betweensamples}} \tag{7}$$

in which the variability of a set of values is estimated by the mean square error (namely, the average sum of squared deviations from the mean value of the set). Equation 7 is used to compute a test statistic for the hypothesis test. Under the null hypothesis that the workload parameter is not influential, the total variability should be similar to the variability within samples. To test the null hypothesis, we evaluate the p-value, i.e., the conditional probability that the test statistic is equal to or greater than the actual value of the test statistic, given that the null hypothesis holds. If the p-value is lower than the preset type I error level (i.e., one minus the preset confidence level), then the null hypothesis can be rejected. In other words, the lower this value, the greater the evidence that the parameter does indeed influence the trend (e.g., if *p-value* < 0.01, the parameter had a significant effect with 99% confidence level).

We use the non-parametric Friedman's test [28] for ANOVA, which only assumes that observations are mutually independent. Mutual independence is guaranteed by the careful experimental setup (e.g., machines are rebooted before each experiment; experiments are executed in random order). Non-parametric tests do not rely on the assumption that data come from normal distributions. However, Friedman's test does not allow to analyze interactions between factors. This limitation is not a concern in our case study, since we are interested in the influence of individual parameters on software aging, in order to map them to aging sources. Since interactions cannot easily be mapped to aging sources, we do not consider them.

Table 5 shows ANOVA results for the case study. Experiments were executed once for each experimental condition, i.e., each combination of workload parameters in F. A single execution was enough to obtain reliable results, since experiments were conducted in a supervised environment (e.g., machines were rebooted after each experiment, and they only executed the test application). ANOVA reveals a non-negligible dependence of RTT trend on T. Hence, we expect the application slowdown to be more evident under most stressful conditions. On the contrary, the amount of transferred data has low influence on RTT. Figure 9b and 9c show the estimated aging trends for RSS, with respect to the client and to the servers, respectively. A significant influence of L on memory consumption can be observed for both the client and the servers. The influence is more pronounced for the client due to the use of the CARDAMOM Trace service to log the whole content of messages. As detailed in the following, Trace service is a significant source of memory leaks in CARDAMOM. The estimated RSS trend exhibits a strong dependence on the invocation period T, probably due to the fact that a memory leak is caused at each invocation.

Table 5: ANOVA on RTT and RSS trends with respect to workload parameters. SS, DF, and MS values refer to the considered factors (T and L).

	RTT		**RSS (client)**		**RSS (server)**	
	T	L	T	L	T	L
Sum of Squares (SS)	23.5	1.8	40	25	40	20.2
Degrees of Freedom (DF)	4	3	4	3	4	3
Mean Squares (MS)	5.875	0.6	10	8.3333	10	6.7333
χ^2	9.4	1.08	16	15	16	12.12
p-value	0.0518	0.7819	0.003	0.0018	0.003	0.007

Aging trend estimates can be leveraged to plan software rejuvenation strategies. Aging trends can be used to predict the time when the application response time or the available free memory will be lower than a predefined Quality of Service (QoS) threshold, and to determine optimal times for triggering a proactive application restart. Hence, it is an effective mean to control and to prevent application performance losses. Of course, the more accurate the trend estimate, the more accurate rejuvenation actions (that will be performed only if required), thus preserving system availability and performance.

5.5.2. TTE estimation

We aim to assess the accuracy of the TTE predictions calculated based on the estimated trends. To this aim we compare actual free memory and the estimated one. As the application is deployed on a single host, predicted free memory for all processes at time t is given by:

$$free\_memory(t) = initial\_memory - \sum_{j} trend_j(T, L) \cdot t \qquad (8)$$

in which $trend_j(T, L)$ is the RSS aging trend for process j, and $initial\_memory$ is the amount of free memory right after the initial transient period (which is estimated by means of Algorithm 1). The trend depends on the workload (we assume *T=300* ms and *L=1000* bytes in this experiment). Observed free memory depletion trend is depicted in Figure 10. It shows that free memory decreases almost linearly (for about 64.5 hours), before stopping at $149,932$ KB. This is due to the OS memory management policy. In Linux environments, the *Out-Of-Memory killer* (OOM) selects processes to be killed in order to guarantee that free memory within the system does not fall below a given threshold. Low priority processes are killed first; then privileges, memory consumption, and execution time are considered to select the victim process [19].

Such a behavior can be observed in Figure 10, where the zoomed region corresponds to the three-hours interval in which the OOM keeps killing low priority processes (the amount of available memory fluctuates around the mean value $149,549$ KB). When the OOM kills the case study processes, almost all physical memory is freed; we hypothesize that the OS delays their termination because they are actively executing.

Assuming $149,932$ KB as the minimum amount of free memory before the OOM starts killing processes, equation 8 estimates the time until which the application can still allocate memory, before other processes get killed. By using the RSS trends in figures 9b and 9c, the estimated TTE amounts to 65.1752 hours. The difference between the actual and the estimated trends amounts to the 1.05% of the actual trend, which is close to the chosen margin of error (see Table 2). Hence, aging trends are able to provide an accurate TTE estimates with respect to the available free memory.

5.5.3. Detailed memory consumption analysis

The most significant aging phenomena emerged when *T=300* ms and *L=1000* bytes. We went further into the investigation of aging sources in this case. We aim to pinpoint

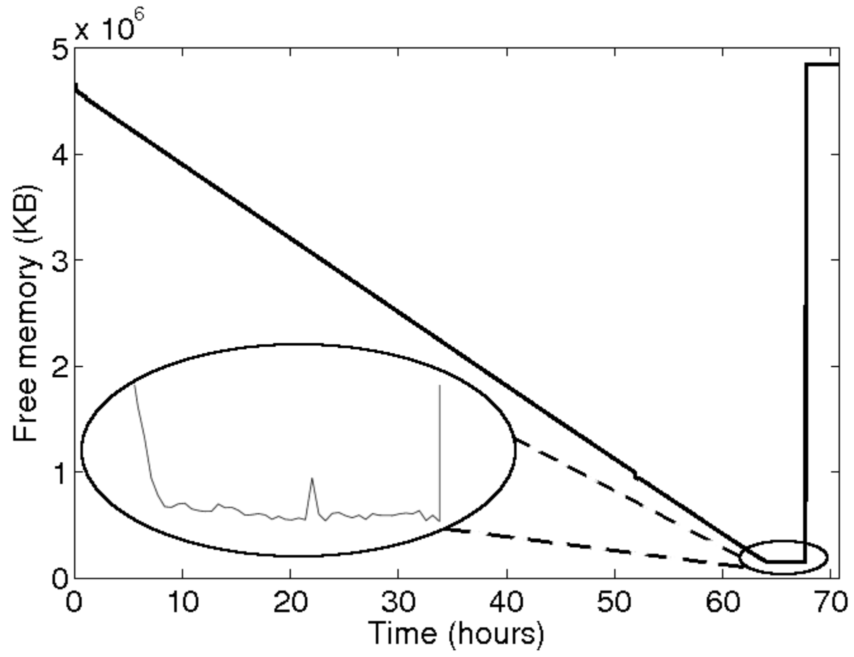


Figure 10: Free memory during most intensive workload conditions, until application crash.

the software items that are responsible for the memory consumption trend. This is done by means of MELANY reports, which have been analyzed in depth.

Processes memory consumption is detailed in Table 6, when using CARDAMOM CE. It points out that the most of memory consumption is due to memory leaks for the client process. On the other hand, the MDL of the server process grows more slowly than the one of the client. This is because the client makes a more extensive usage of CARDAMOM libraries. Additionally, a greater number of memory leaks at the client side are due to the TAO ORB. *These results are useful hints for developers* (e.g., they could concentrate their efforts on the improvement of those services used by the client).

Table 7 provides an in-depth analysis on the sources of memory leaks for CARDAMOM CE. As evidenced, most of them are due to the Trace service and the TAO ORB. Additionally, we found out that memory leaks in TAO are greatly affected by the string size L, i.e., the volume of transferred data (we omit these data for the sake of brevity). This makes it reasonable to suppose that leaks are hidden in the request serialization handling code.

Memory usage reports provided by MELANY also helped us to pinpoint the defects in CARDAMOM source code that caused memory leaks. These data have been sent as feedback to the CARDAMOM development team in order to get these bugs fixed.

Table 8 summarizes memory consumption for both the client and the server, when using CARDAMOM DV. A significant reduction of MDL has been achieved in this version.

Table 9 indicates that the main source of memory leaks in CARDAMOM CE (i.e., the shared library code of the Trace service) has been removed. Nevertheless, the TMC in CARDAMOM DV is even greater. This is because almost all the wasted memory is referenced

Table 6: Memory consumption (KB) for CARDAMOM CE processes

	Client				**Server**		
	1 hour	**2 hours**	**3 hours**		**1 hour**	**2 hours**	**3 hours**
TMC	140,120	184,832	229,268	**TMC**	116,280	128,052	139,740
IMC	115,755	136,337	156,635	**IMC**	113,897	123,334	132,694
MDL	24,365	48,495	72,633	**MDL**	2,383	4,718	7,046

Table 7: Memory leaks for CARDAMOM CE.

Source Object	**Leaked Bytes**	**Errors / Functions**
libACE.so.5.4.1	31,826,000	1 / 1
libTAO.so.1.4.1	31,826,262	5 / 2
libcdmwcommon.so	29	1 / 1
libcdmwlbcommon.so	276	3 / 2
libcdmwlogging.so	10,723,932	52 / 1
libxerces-c1_3.so	24	1 / 1

(a) Client process.

Source Object	**Leaked Bytes**	**Errors / Functions**
libTAO.so.1.4.1	3,202	1 / 1
libcdmwcommon.so	29	1 / 1
libcdmwlbcommon.so	176	2 / 1
libcdmwlogging.so	7,212,140	16 / 1
libxerces-c1_3.so	24	1 / 1

(b) Server process.

for the whole experiment duration. We hypothesize that CARDAMOM developers prevent memory leaks by continuously storing references to the allocated memory (e.g., by means of the C++ "smart pointers" feature or abstract data types such as STL containers), in order to delay deallocation of unnecessary memory. This approach can inadvertently lead to memory exhaustion if deallocation does not occur on time (e.g., elements within an STL container are never deleted), as in the case of CARDAMOM DV.

It should also be noted that sources of memory leaks still exists in CARDAMOM DV. Table 9 shows that almost all memory leaks are located in the code of the TAO library; CARDAMOM developers did not correct these defects because they were located in third party OTS components.

6. Conclusions

This paper proposed a practical approach to detect aging trends due to memory leaks in complex software systems based on a CORBA-compliant OTS middleware. A real-world middleware for developing mission-critical applications for Air Traffic Control has been studied. More specifically, the proposed approach aimed (i) to achieve a definite quality of results, in terms of statistical accuracy of aging trends, and (ii) to keep down experiments duration, by means of algorithms for data filtering and careful scheduling of the test campaign. The

Table 8: Memory consumption (KB) for CARDAMOM DV processes

	Client				Server		
	1 hour	2 hours	3 hours		1 hour	2 hours	3 hours
TMC	153,108	202,768	252,604	**TMC**	140,336	174,704	210,240
IMC	152,930	202,414	252,074	**IMC**	140,332.8	174,700.8	210,236.8
MDL	178	354	530	**MDL**	3.2	3.2	3.2

Table 9: Memory leaks for CARDAMOM DV.

Source Object	Leaked Bytes	Errors / Functions
libACE.so.5.5.1	362,082	2 / 1
libTAO.so.1.5.1	181,276	5 / 2
libcdmwlbcommon.so	276	3 / 2

(a) Client process.

Source Object	Leaked Bytes	Errors / Functions
libcdmwlbcommon.so	3,321	2 / 1

(b) Server process.

approach allowed to estimate the aging trends with a duration of experiments much shorter than other approaches (given the margin of error), with a time reduction factor greater than 3. Moreover, we experienced that several problems can arise when coping with memory leaks in complex applications. The experiments with the real-world case study revealed the following two interesting lessons.

OTS items are a significant source of memory leaks. In general, the adoption of OTS software poses serious dependability issues on complex systems, because their interactions within complex systems may trigger latent faults. Even when defects within OTS items are known, as in the case of memory leaks spotted by MELANY, they are difficult to deal with, because system integrators have little or no knowledge about internals of OTS items (e.g., these items are often closed source, and they have been made by other developers). Therefore, it can be expected that OTS items will be the most significant source of memory leaks when software becomes mature. Currently, the only way to approach them is software rejuvenation, although an approach for correcting them in advance, or at least to prevent memory consumption, would be beneficial for system availability and performance; several efforts are devoted to this problem by the research community in this field, but software aging in OTS items still represents an open issue.

Developers may not correctly fix known memory leaks. Even after CARDAMOM developers tackled the memory leaks pointed out by our analysis, the problem of memory exhaustion remained unresolved; the memory consumption trend is still present in CARDAMOM DV. This result may be due to the lack of adequate techniques and skills to cope with memory management defects during the application lifecycle; simpler and more effective development techniques in this field, or the adoption of long-term memory management strategies, would be helpful to solve these defects.

References

[1] B. Natarajan, A. Gokhale, D. Schmidt, C. Gill, J. Cross, C. Andrews, S. Fernandez, Towards Dependable Real-time and Embedded CORBA Systems, in: IEEE Workshop on Dependable Middleware-Based Systems, 2002.

[2] E. Weyuker, Testing Component-Based Software: A Cautionary Tale, IEEE Software 15 (5) (1998) 54–59.

[3] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, H. Madeira, Experimental Risk Assessment and Comparison Using Software Fault Injection, in: Proc. of the 37th Intl. Conf. on Dependable Systems and Networks (DSN), IEEE Computer Society, 2007, pp. 512–521.

[4] M. Grottke, K. Trivedi, Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate, IEEE Computer 40 (2) (2007) 107–109.

[5] M. Grottke, K. Trivedi, Software Faults, Software Aging and Software Rejuvenation, Journal of the Reliability Engineering Association of Japan 27 (7) (2005) 425–438.

[6] S. Garg, A. V. Moorsel, K. Vaidyanathan, K. S. Trivedi, A Methodology for Detection and Estimation of Software Aging, in: Proc. of the 9th Intl. Symp. on Software Reliability Engineering (ISSRE), IEEE Computer Society, Washington, DC, USA, 1998, p. 283.

[7] M. Grottke, L. Li, K. Vaidyanathan, K. S. Trivedi, Analysis of Software Aging in a Web Server, IEEE Trans. Reliability (2006) 480–491.

[8] Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, Software Rejuvenation: Analysis, Module and Applications, in: Proc. of IEEE Fault Tolerant Computing Symp. (FTCS), 1995, pp. 381–390.

[9] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, Failure Data Analysis of a LAN of Windows NT Based Computers, in: Proc. of the 18th IEEE Symp. on Reliable Distributed Systems (SRDS), IEEE Computer Society, Washington, DC, USA, 1999, p. 178.

[10] S. Cherem, L. Princehouse, R. Rugina, Practical Memory Leak Detection using Guarded Value-Flow Analysis, in: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), ACM Press New York, NY, USA, 2007, pp. 480–491.

[11] G. Xu, A. Rountev, Precise Memory Leak Detection for Java Software using Container Profiling, in: Proc. of the 30th Intl. Conf. on Software Engineering (ICSE), ACM, New York, NY, USA, 2008, pp. 151–160.

[12] K. Vaidyanathan, K. Trivedi, A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems, in: Proc. of the 10th Intl. Symp. on Software Reliability Engineering (ISSRE), 1999, pp. 84–93.

[13] D. Wang, W. Xie, K. S. Trivedi, Performability Analysis of Clustered Systems with Rejuvenation under Varying Workload, Performance Evaluation 64 (3) (2007) 247–265.

[14] D. Cotroneo, S. Orlando, S. Russo, Characterizing Aging Phenomena of the Java Virtual Machine, in: Proc. of the 26th IEEE Symp. on Reliable Distributed Systems (SRDS), IEEE Computer Society, 2007, pp. 127–136.

[15] R. Matias Jr, P. Filho, An Experimental Study on Software Aging and Rejuvenation in Web Servers, in: Proc. of the 30th Intl. Computer Software and Applications Conference (COMPSAC), IEEE Computer Society Washington, DC, USA, 2006, pp. 189–196.

[16] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons New York, 1991.

[17] D. Montgomery, Design and Analysis of Experiments, John Wiley New York, 1991.

[18] Y. Jia, X. Chen, L. Zhao, K. Cai, On the Relationship between Software Aging and Related Parameters, in: Proc. of the 8th Intl. Conf. on Quality Software (QSIC), 2008, pp. 241–246.

[19] D. Bovet, M. Cesati, Understanding the Linux Kernel, O'Reilly, 2005.

[20] H. Mann, Nonparametric Tests Against Trend, Econometrica 13 (3) (1945) 245–259.

[21] P. K. Sen, Estimates of the Regression Coefficient based on Kendall's tau, Journal of the American Statistical Association (1986) 63:1379–1389.

[22] E. D. Berger, B. G. Zorn, K. S. McKinley, Reconsidering custom memory allocation, in: Proc. of the 17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002, pp. 1–12.

[23] S. Mamagkakis, C. Baloukas, D. Atienza, F. Catthoor, D. Soudris, J. Mendias, A. Thanailakis, Reducing Memory Fragmentation with Performance-Optimized Dynamic Memory Allocators in Network Applications, in: Proc. of the 3rd Intl. Conf. on Wired/Wireless Internet Communications (WWIC), 2005, pp. 354–364.

[24] V. Castelli, R. Harper, P. Heidelberger, S. Hunter, K. Trivedi, K. Vaidyanathan, W. Zeggert, Proactive Management of Software Aging, IBM Journal of Research and Development 45 (2) (2001) 311–332.

[25] L. Schruben, Detecting Initialization Bias in Simulation Output, Operations Research 30 (3) (1982) 569–590.

[26] D. Goldsman, L. Schruben, J. Swain, Tests for Transient Means in Simulated Time Series, Naval Research Logistics 41 (2).

[27] J. Lee, D. McNickle, K. Pawlikowski, N. Christchurch, Initial Transient Period Detection for Steady-State Quantile Estimation, in: Summer Computer Simulation Conference, 2000, pp. 169–174.

[28] M. Hollander, D. Wolfe, Nonparametric statistical methods, John Wiley New York, 1973.