An experiment in memory leak analysis with a mission-critical middleware for Air Traffic Control

G. Carrozza*‡, D. Cotroneo*, R. Natella*†, A. Pecchia*† and S. Russo*†
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
Email: {ga.carrozza, cotroneo, stefano.russo}@unina.it
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed.1,
Via Cinthia, 80126, Naples, Italy
Email: {roberto.natella, antonio.pecchia}@consorzio-cini.it
‡Consorzio SESM-SCARL, Via Circumvallazione Esterna di Napoli, 80014, Giugliano in Campania, Naples, Italy

***Abstract*—This paper reports a practical experience with memory analysis on a real world complex middleware platform, being developed in the context of an academic-industrial collaboration. The reported experience suggests a practical method that can help practitioners to analyze memory leaks and to adopt proper actions to mitigate these bugs, especially in the context of complex Off-The-Shelf based software systems, and in some cases it highlights issues still open. Indeed, we experience that fixing a memory leak, when possible, might be not enough to solve the memory exhaustion problem.**

I. Introduction

Current software systems are getting more and more complex both in the services they provide and in their architectures. The use of Off-The-Shelf (OTS) software items helps developers to keep down the development efforts and the time-to-market but, on the other hand, represents a threat for the system dependability level. This is more significant in case of safety and mission critical systems where components delivered by third-party manufactures or communities (e.g., libraries, compilers, virtual machines, up to middleware platforms and operating systems) are adopted. The main problem is that they do not provide dependability guarantees when they are integrated, because of complex interactions within a system [1]. Furthermore, they typically come as executable items (e.g., binary objects to be linked in the case of C/C++ programs) to be included within the application under development. This makes difficult to discover, to locate, and to treat latent software defects (also named bugs).

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 frameworks of the Centro di ricerca sui sistemi Open Source per le applicazioni ed i Servizi MIssion Critical (COSMIC) Project (www.cosmiclab.it). It is also in the context the "Iniziativa Software" Project, an Italian Research project which involves Finmeccanica company and several Italian universities (www.iniziativasoftware.it).

This paper addresses aging-related bugs [2]. They represent software defects that lead to the progressive system state corruption (this issue is also known as software aging). These bugs typically cause memory leaks, data corruption, unreleased file handles or locks, unclosed sockets, round-off errors buildup and so on. As a general rule, aging sources should be located and fixed during the application development. Unfortunately, it is quite difficult locating these bugs during system testing since the execution trigger that potentially causes the system failure may not be easily reproduced. Even when the bug has been detected, it has to be located and potentially fixed: this is trivial if it is located in the source code, but it is not always possible to do that, since many defects are located in OTS software items. In particular, we focus on memory leaks because they represent the most considerable source of software aging, as shown by several works.

This paper reports a practical experience with memory analysis on a real world industrial middleware platform, namely CARDAMOM, and, in particular, it describes a memory leaks analysis technique used to mitigate memory leaks in the considered case study. An open-source Community Edition (CE) version of this platform is available on the Web[1], and it is currently under study within the framework of the COSMIC project, which aims to achieve a dependable version of the middleware (in the following Development Version, DV). Moreover, our experiment shows that fixing a memory leak might be not enough to solve the memory exhaustion problem. In these cases, it is very useful to estimate the Time To Exhaustion (TTE) of the involved system resources in order to plan proper rejuvenation strategies.

The paper is organized as follows. Section II discusses related works that justify our approach, and introduces our reference middleware platform. Section III describes the proposed analysis technique and the related tool, Melany, designed to automatically process data related to memory leaks. In section

[1]http://forge.objectweb.org/projects/cardamom

IV we use the proposed strategy and the developed tool to locate and fix several memory leaks. In section V we conclude the work, by discussing the key findings of our experience with this industrial system.

II. Background

Software aging is a well known issue both for mass-scale and safety-critical applications [3]. Several works report practical evaluation of operational systems showing that aging-related bugs cause performance degradation or, at least, hang or crash failures. These works, which are reported below, also point out that memory management software related defects (e.g., memory leaks) represent the most considerable aging source. This statement justifies our study which focuses on the characterization of memory usage.

A methodology for estimation of aging is presented in [4]. A comparative analysis of aging effects on different system resources shows that estimated TTE for real free memory is lower than the other system resources (i.e. the process or the file table size). A similar finding comes out from the analysis of software aging in a web server provided in [5]. As stated in [6], a significant problem with a software component is the machine running out of virtual memory, often caused by a memory leak in the same component. Finally, we can cite several works that directly focus on memory leak issues, even in case of Java software [7], [8], [9].

As mentioned, the practical experience described by this work concerns the CARDAMOM middleware. It is an open source CORBA-based platform supporting both the object and the component programming models and it is used in the development of safety and mission critical systems such as the ATC domain.

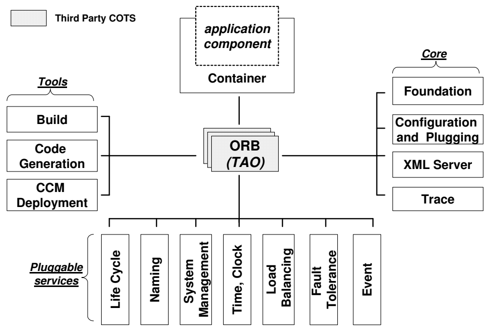

Fig. 1: CARDAMOM overview.

Fig. 1 gives an overall view of CARDAMOM. It provides both tools to simplify application development (e.g., automatic code generation) and two sets of services intended to be used by critical applications. The core services are mandatory for a CARDAMOM-based application. Furthermore, one or more pluggable services can be adopted by the same application. The described platform makes use of several OTS components. Among them, the most relevant are the Linux operating system and a third-party ORB (specifically TAO[2]). The presence of this OTS item is particularly relevant for the issues we are going to discuss.

III. Characterization of memory usage

In order to cope with the analysis of resources usage by complex long-running applications, we adopted an approach based on stress tests. The target application is executed under stressful environmental conditions (e.g., by requesting a high number of operations) for a long period, in order to point out resource leaks, that otherwise would not be identified during the testing phase of the system (e.g., functional and performance tests), and could lead to unpredictable failures during the operational phase. Moreover, stress tests are repeated under different environmental conditions, in order to highlight resource leaks dependencies on workload parameters (which, in turn, can be used for long-term predictions about resource consumption). During the program execution field data are collected in order to be processed by external analysis tools.

We have developed a specific tool, MELANY (MEmory Leak ANalYzer), to locate and to analyze memory-leaks sources within complex C++ applications. Its goals are to i) find source-code memory management errors, and ii) estimate the TTE of the main memory.

As a matter of fact, locating the code line that causes a memory leak can help developers to correct and to leverage the software. On the other hand, many leaks come from libraries and other runtime supports. In this case, it is not always possible to fix the error, although the TTE estimation turns out to be useful to plan rejuvenation strategies. Fig. 2 gives an overall view about the adopted information processing approach.

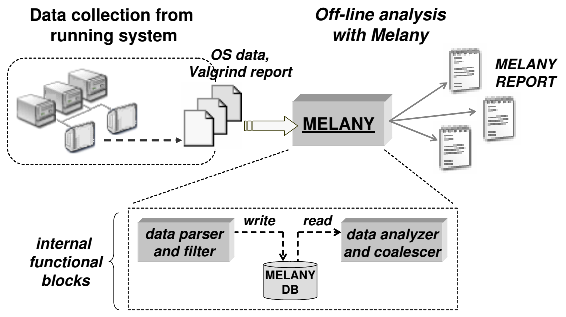
Data collection from running system
Off-line analysis with Melany
OS data, Valgrind report
MELANY
MELANY REPORT
internal functional blocks
data parser and filter
write
read
data analyzer and coalescer
MELANY DB

Fig. 2: Analysis approach.

First of all, there is a preliminary data collection process from the target system. Collected data are imported within the MELANY environment by loading them into an internal database. Using MELANY, the analyst can generate several reports at different levels of detail. As a matter of fact, the reports contain information as: the occurred memory leaks

[2]http://www.theaceorb.com

with the whole invocation stack that lead to the error; the source objects causing one or more leaks, their number, and the functions where they are located; statistics concerning the memory usage of the application during the test.

A. Information sources

MELANY infers the needed information from two sources: the operating system and Valgrind[3]. We obtain the operating system (OS) information as the output of the command-line util `ps`. In particular, for the target process we take into account the amount of allocated virtual memory, PID, priority, etc.

The other information source is represented by XML reports coming from Valgrind. In particular, using the memcheck tool, we enable memory-usage monitoring in order to discover the following errors: i) syscall parameters (the application tries to use a not addressable memory buffer), ii) memory leaks which are grouped in:

- *definitely lost bytes*: the application is certainly causing a leak and it has to be fixed;
- *possibly lost bytes*: the application is probably causing a leak, unless it is having a very complex pointers usage.

Both the operating system and Valgrind output is saved on files during the data collection phase. These files are intended to be processed off-line by MELANY.

B. MELANY output

The collected data contain heterogeneous information. This is the reason why we developed a tailored tool able to merge both operating system logs and Valgrind report. As shown in Fig. 2, MELANY parses these data and performs filtering operations to store in the internal database (DB) only the relevant part for the subsequent analysis phase. The DB contains, for each memory error of the target process, the error type, the amount of memory involved in the error, and the source location of the error. This information is inferred from the Valgrind report, and DB entries are populated. We enriched the DB also with the information on the amount of allocated memory extracted from the OS logs, since memory consumption is not due only by memory leaks. From this point on, statistical analysis concerning memory errors and usage can be performed. MELANY performs coalescing operations (by the means of SQL queries) to build aggregated results from the single memory leak entries. For instance, leaks are grouped by source library in order to evaluate its contribution with respect to the total leaked memory.

MELANY provides two memory error report formats (namely, *base* and *detailed*) and a summarizing report describing memory leak sources (namely *source*). The base-report gives overall information about the under-test process, the total amount of errors and allocated memory and, finally, the complete errors list. On the other hand, the detailed-report adds, for each error, the whole invocation stack. In Fig. 3 (A)

[3]http://valgrind.org

```
Type: Leak Possibly Lost
Description: 20 bytes in 1 blocks are possibly lost in loss record
  475 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)

```
Library:
/tools/exec/TAO141_2_06_0913/src/TAO/tao/libTAO.so.1.4.1 Error
  occurencies: 12

  Root Source 'client.cpp':                            91%
  Root Source '/lib/tls/libc-2.3.4.so'                 8%
```

(B)

Fig. 3: Leak stack trace (A) - Source-report entry (B).

we show a simplified leak stack trace provided by the detailed report. For each library causing a leak, the source-report shows the number of errors, the error root-causes, and the percentage of each root on the errors amount. Fig. 3 (B) shows a single entry of this report. In the example figure, the leak located in *libTAO.so.1.4.1* occurs 12 times.

IV. Real World Case Study

We evaluate the memory consumption of the CARDAMOM middleware, through a testing application which uses both core and pluggable services provided by the middleware (see Fig. 1); in particular, the Trace service (which will turn out to be the main source of memory leaks) is responsible for collecting application level messages. The application is composed by two clients and two servers on different nodes, in which clients' requests are balanced among the servers using a round-robin policy (Fig. 4). The remote method invoked by the clients returns a fixed-size string. The application is simple with intent, because we want to highlight memory leaks only introduced by the middleware. The workload can be varyied with respect to i) the time between consecutive requests (T) (in order to evaluate the memory leak trend in function of time), and ii) the string size (L) (to find out if the amount of processed data may affect memory management). We study the memory consumption of CARDAMOM in function of these workload parameters, in order to discover in what extent memory leaks are introduced by the middleware when the load increases. One of the client processes (which make the most extended use of middleware services, in the form of shared libraries) was analyzed as described in section III. The hardware configuration of testing machines is made up of 2 Xeon Hyper-Threaded 2.8GHz CPUs, 3.6GB of physical memory, and a Gigabit Ethernet interface; machines are interconnected by a 56 Gbps switch.

First of all, we conduct a preliminary experiment to analyze the memory consumption trend of the client process in function of time, for a particular choice of workload parameters ($T=300$ ms, $L=1$ byte), in CARDAMOM CE. After two hours, the

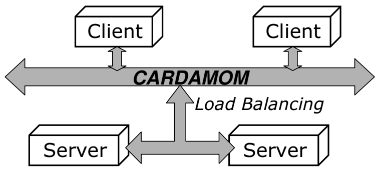

Fig. 4: Testing application using CARDAMOM services.

memory consumption increases by $\Delta = 4808$ KB (from 42572 KB to 47380 KB), despite that it should remain constant (no explicit memory allocation is made by the application; therefore, unnecessary memory consumption is due to CARDAMOM misbehavior). Memory consumption is periodically sampled every second, and a large set of data samples was collected. Linear regression shows that the memory consumption increases linearly with time throughout the test (with a correlation coefficient $r > 0.99$), and TTE amounts to 1484 hours in our testbed. The test actually causes the crash of the application. This result demonstrates that the middleware is affected by software aging, and it can eventually lead to memory exhaustion. Because the memory consumption grows linearly, we hypothesize that there are predominant sources of memory leaks (within the middleware code involved in the remote method invocation) which constantly adds a memory leak delta at each invocation, and which should be pinpointed by the means of precise memory usage analysis.

Next, we evaluate the memory consumption after a fixed execution time (two hours) in function of workload parameters *T* and *L*. As Fig. 5 shows, the memory consumption increases with the number of processed requests (i.e., when *T* decreases). Moreover, the greater the size of the requested string, the greater the memory consumption. Therefore, the amount of wasted memory by the faulty code is not fixed, but varies with the volume and contents of transferred data (e.g., memory leaks are hidden within code that handles CORBA data structures).

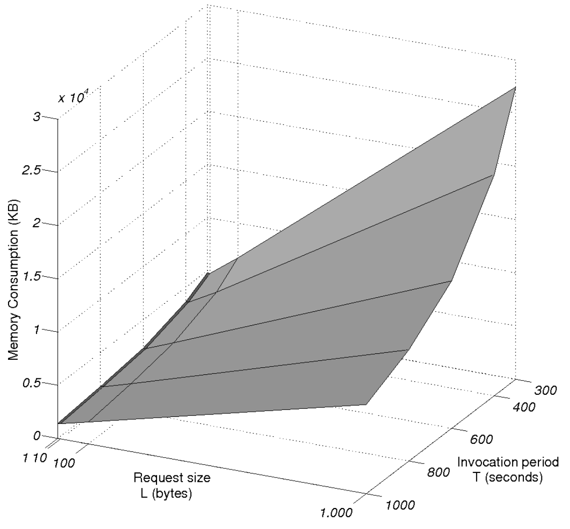

Fig. 5: Memory consumption after 2 hours, with respect to the request size (L) and invocation period (T).

The memory consumption for CARDAMOM CE is better detailed in Fig. 6. The Total Memory Consumption (TMC) is split into Intentional Memory Consumption (IMC) and Memory Definitely Lost (MDL). The former represents the amount of memory that can be referenced by the application (e.g., there exists a valid pointer to that memory region); the latter represents memory leaks (therefore, TMC = IMC + MDL). It can be observed that the greatest part of memory consumption was due to memory leaks; the IMC growth with time is negligible.

In order to discover software items responsible of this memory consumption trend, reports provided by MELANY were closely investigated. TABLE I details the sources of memory leaks for CARDAMOM CE: most of them are due to the Trace service and the TAO ORB underlying CARDAMOM (highlighted in TABLE I). Memory usage reports were useful to spot the defects liable of memory leaks within the source code of CARDAMOM. This data was feed back to CARDAMOM developers, which tried to tackle these defects in order to reduce the amount of leaked memory.

TABLE I
MEMORY LEAKS IN CARDAMOM CE.

Source Object	**Leaked Bytes**	**Errors / Functions**
Client	88	1 / 1
libACE.so.5.4.1	1799990 (1.46%)	2 / 1
libTAO.so.1.4.1	905844 (0.73%)	12 / 4
libTAO_CosNaming.so.1.4.1	176	2 / 1
libcdmwcommon.so	177	2 / 1
libcdmwlbcommon.so	276	3 / 2
libcdmwlbinit.so	123	1 / 1
libcdmwlogging.so	120953872 (97.81%)	57 / 1
libcdmworbsupport.so	48	1 / 1
libcdmwrepositoryidl.so	88	1 / 1
libxerces-c1_3.so	24	1 / 1

Once a new CARDAMOM version has been released, we tested it in the same way as the previous version. Fig. 7 shows the memory consumption of the corrected CARDAMOM version (DV), which has to be compared to CARDAMOM CE in Fig. 6. In CARDAMOM DV, the MDL dramatically decreases. TABLE II highlights that the main source of memory leaks (the shared library code of the Trace service) was removed. Nevertheless, the TMC in CARDAMOM DV is even greater than the one in CARDAMOM CE. Therefore, although the amount of non-referenced memory (i.e., the MDL) is reduced, the TTE does not benefit from the corrections (TTE varies from 128 hours in CE to 100 hours in DV) because, as shown in Fig. 7, the wasted memory is still referenced for the whole duration of the experiments. We hypothesize that CARDAMOM developers prevent memory leaks by continuously storing references to the allocated memory, delaying the correct memory deallocation; nevertheless, this approach leads

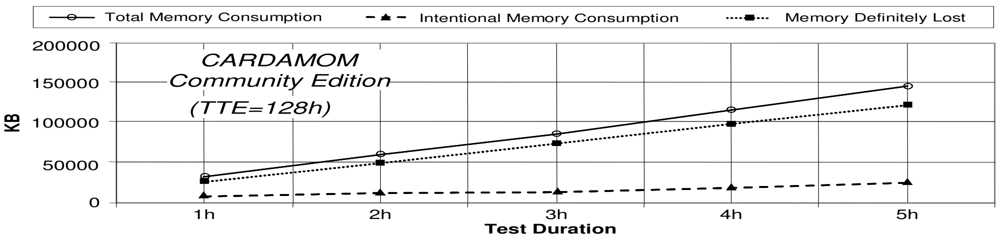

Fig. 6: Contributions to memory consumption by CARDAMOM CE.

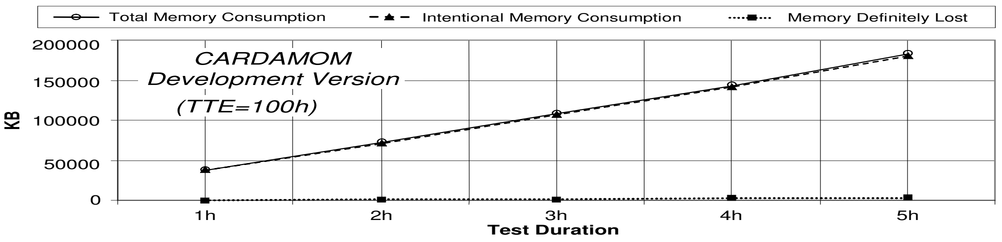

Fig. 7: Contributions to memory consumption by CARDAMOM DV.

to memory exhaustion.

TABLE II
MEMORY LEAKS IN CARDAMOM DV.

Source Object	Leaked Bytes	Errors / Functions
Client	76	1 / 1
libACE.so.5.5.1	1799970 (66.49%)	2 / 1
libTAO.so.1.5.1	905172 (33.44%)	9 / 3
libTAO_CosNaming.so.1.5.1	1181	2 / 1
libcdmwcommon.so	148	1 / 1
libcdmwlbcommon.so	276	3 / 2
libcdmwlbinit.so	111	1 / 1
libcdmworbsupport.so	8	1 / 1
libcdmwrepositoryidl.so	76	1 / 1

Finally, it should be noted that sources of memory leaks still exist in CARDAMOM DV. TABLE II shows that almost all memory leaks are located in TAO's library code; CARDAMOM developers were not able to correct these defects because they were located in third party OTS components.

V. LESSONS LEARNED AND OPEN ISSUES

This paper showed that several problems can arise when coping with memory leaks in complex applications. The practical experience with the case study let us to obtain the following lessons, and leaved some questions open, which may deserve further research efforts in the near future:

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 spot by MELANY, they are difficult to deal with, because system integrators have little or no knowledge about OTS items internals (e.g., they 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 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 remains unsolved, because the memory consumption trend is still present in CARDAMOM DV. This result may be due to the lack of adequate 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.

ACKNOWLEDGMENTS

Authors would like to thank Mr. Guido Balzano and Mr. Antonio Strano for the precious efforts they put in the completion of this work, as well as for they constant technical support and cooperation.

REFERENCES

[1] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental risk assessment and comparison using software fault injection. In *Proc. of the 37th Intl. Conf. on Dependable Systems and Networks (DSN)*, 2007.

[2] K. S. Trivedi K. Vaidyanathan. Extended Classification of Software Faults Based on Aging. In *Proc. of the 12th Intl. Symp. on Software Reliability Engineering (ISSRE)*, 2001.

[3] Y. Huang, C. Kintala, N. Kolettis, and N. Dudley Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Fault Tolerant Computing Symp. (FTCS)*, 1995.

[4] S. Garg, A. Van 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 (ISSRE)*, 1998.

[5] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of software aging in a web server. In *IEEE Trans. Reliability, 2006*.

[6] M. Kalyanakrishnam, Z. Kalbarczyk, and 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)*, 1999.

[7] S. Cherem, L. Princehouse, and 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)*, 2007.

[8] G. Xu and A. Rountev. Precise Memory Leak Detection for Java Software using Container Profiling. In *Proc. of the 30th Intl. Conf. on Software Engineering (ICSE)*, 2008.

[9] D. Cotroneo, S. Orlando, and S. Russo. Characterizing aging phenomena of the java virtual machine. In *Proc. of the 26th IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2007.