CARRYING THE CRASH-ONLY SOFTWARE CONCEPT TO THE LEGACY APPLICATION SERVERS

Javier Alonso and Jordi Torres
Technical University of Catalonia
Barcelona Supercomputing Center,
Barcelona,Spain
alonso@ac.upc.edu
torres@ac.upc.edu

Luis Moura Silva
University of Coimbra
CISUC, Portugal
luis@dei.uc.pt

Abstract In the last few years, high-availability on internet services has become a main goal for the academia and industry. We all know how complex and heterogeneous Internet service systems are and how sensitive to suffer from transient failures or even crashes also. Because developing systems that are guaranteed to never crash and never suffer transient or intermittent failures seems an impractical and unfeasible business, there is a need to develop mechanisms that can suffer crashes and transient failures as if they were a clean shutdown. Behind this idea, the creators of the crash-only software concept proposed a new design strategy in order to get crash-safe and fast recovery systems by defining a list of laws which are needed in order to achieve that goal. However, their proposals are focused on new systems design. For this reason, we will discuss how to develop crash-safe and masked fast self-recovery legacy systems following the ideas behind the crash-only software concept. In our work, we have focused on legacy application servers because they are a more sensitive piece of the internet services' big puzzle.

Keywords: Crash-only software, Legacy Software, Self-recovery, Self-healing, Automatic recovery

1. Introduction

High-availability has become one of the main success characteristics for every company engaged in e-business and e-commerce since every company wants their internet services running 24x7 to maximize their revenue. For all internet service companies, high-availability is of paramount importance because unavailability time means potential revenue losses. In recent history, there have been famous internet service outages, resulting in big revenue losses and a bad image for service companies' owners. In general terms, [1] calculates that the cost of downtime per hour can go from 100k for online stores up to 6 million dollars for online brokerage services. One of the most important cases occurred in April 1999, when e-Bay suffered a 24-hours outage. In that time, it was calculated that e-Bay lost around 5 billion dollars in revenue and stock value [4]. This type of failures and similar outages, like the 15 minutes Goggle Outage in 2005 can also affect the customer loyalty and investor confidence [5], resulting in a potential loss of revenue.

Unfortunately, system outages and application failures do occur, provoking crashes.

We can classify the bugs in two big sets of them: permanent failures and transient failures.

The first subset refers to bugs that can be solved in development processes because they're easy to reproduce. However, the second type of failures, also known as Heisenbugs [3] are difficult to fix because they're difficult to reproduce because they depend on the timing of external events and often the best way to skip them is to simply restart the application, the application server or even the whole physical or virtual machine.

We have focused on this second type of bug.

Internet services are complex and dynamic environments where there are lots of components interacting with each other. This type of environments are sensitive to transient failures; however, a simple restart to solve possible failures and crashes may not be enough to solve them and indeed, if we apply blind restarts, internet services can suffer application data inconsistency problems. This happens because while we restart the internet service, the volatile state of user sessions needed for future interactions between the application and the user to achieve a successful communication is lost, also displaying the restart process to end users as a failure or a crash.

As it seems that it is unfeasible to build systems that are guaranteed to never crash or suffer transient failures, G. Candea and A. Fox proposed the crash-only software concept [7]. The crash-only software is based on the idea of designing software systems that handle failures by simply restarting them without attempting any sophisticated recovery. Since, some authors have proposed the idea of developing new Internet services using the concept of crash-only soft-

ware. The crash-only concept can be seen like a generalization of the transaction model taken from the data storage world. The crash-only system failures have similar effects over the data storage systems. However, what happens with the recently designed and deployed internet services, usually made-up by cooperative legacy servers?

In this paper, we discuss the possibility to migrate the crash-only software concept to these legacy servers for internet services. We focus on application servers because traditionally, these servers have the business logical of the applications and are more sensitive to the transient failures or potential crashes.

As an example, we expose the classical online flight ticket store. We have a multi-tier application environment made-up by one web server for static web pages, an application server for business logical and a database server for storing all ticket flight information.

We can define the session in this environment as the process in between the user logs-in and finally, logs-out. During this process the user can search for flights, add some of them to his/her shopping basket and pay for them. If there was a crash during the session process, the more sensitive point of the multi-tier application would be the application server, because the web server only manages static information and database servers have their traditional after crash recovery systems. The application servers have the responsibility to manage the session state. This session state is needed for the session's success because the session state contains information useful to the session's subsequent states. For this reason, if there was a crash in a legacy application server, a simple and blind restart could be dangerous to the consistency of the business logic state.

Due to the crash-only constraints we restrict our proposal to application servers with external session state storage, which meets crash-only laws like SSM [8] or Postgres database system [15].

This paper's main goal is to migrate the ideas proposed by authors of the crash-only concept to the current legacy application servers and put them to practice in order to obtain a "crash-safe" and "fast" recovery system.

The second goal of our proposal is to hide any possible crash from the end-users improving the crash-only software concept which achieves crash-safe and fast recovery systems although it doesn't avoid the occasional unavailability time, as it was shown through the Microreboot technique [6]. The rest of the paper follows as: Section 2 presents the crash-only software properties. Section 3 discusses the viability of crash-only software characteristics on legacy application servers' environments. Section 4 presents our proposal architecture to achieve our two principal goals and section. Section 5 concludes this paper.

2. Crash-only Software

The crash-only concept is based on the idea that all systems can suffer transient failures and crashes thus it'd be useful to develop systems that could overcome a potential crash or transient failure. Furthermore, normally the system crash recovery time is lower than the time needed to apply a clean reboot using the tools provided by the application itself. Table 1 illustrates this reasoning.

System	Clean Reboot	Crash Reboot
RedHat 8 (with ext3fs)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

Table 1. Table obtained from [7]

The reason for this phenomenon is the desire to improve the system performance. The systems usually store potential non-volatile information on volatile devices, like RAM, to be faster and obtain higher performance. For this reason, before a system can be rebooted, all this information has to be saved on non-volatile devices like hard disks to maintain data consistency. However, in the case of a crash reboot all this information is lost and potential inconsistency state may result after reboot.

Based on these potential problems, crash-only software is software where a crash behaves as a clean reboot. Every crash-only system has to be made of crash-only components and every crash-only component has only one way to stop - by crashing the component- and only one way to start the component: applying a recovery process. To obtain crash-only components, authors defined five properties that the component has to include:

(a). *All important non-volatile state is managed by dedicated state stores.* The application becomes a stateless client of the session state stores, which helps and simplifies the recovery process after a crash. Of course, this session state store has to be crash-only, otherwise the problem has just moved down to the other place.

(b). *Components have externally enforced boundaries.* All components have to be isolated from the rest of the components to avoid that one faulty component may provoke another fault on another component. There are different and potential ways to isolate components. One of the most successful ways for isolating components which has become popular in the last few years is virtualization [16].

(c). *All interactions between components have a timeout.* Any communication between components has to have a timeout. If no response is

received after that time, the caller can assume that the callee is failing and the caller can start a crash and recovery process for the component failing.

(d). *All resources are leased.* The resources cannot be coupled up indefinitely thus it is necessary to either guarantee the resources be free after a limited time or the component using the resources crashes.

(e). *Requests are entirely self-describing.* It is needed that all requests are entirely self-describing to make the recovery process easier. After a crash, the system can continue from where the previous instance left off. It is necessary to know the time to live (TTL) of the request and the indempotency property.

Trying to describe in fine detail all philosophy of Crash-only software in this paper would be out of scope. In order to obtain more information, we recommend you visit the ROC project [2] and the website [17] and other authors' papers around this concept.

3. Crash-only and masking failure Architecture for Legacy Application Servers

Our architecture is focused on a determinate set of application servers: application servers with external session state storage. The reason is because we cannot force an application server to manage their internal session objects to external storage without changing the code. We want to propose a solution which avoids modifying the legacy application server code because this can be a titanic work or even impossible if the software is also closed.

The architecture is made-up by three main components: the Requests Handler (RH), the Storage Management Proxy (SMP) and the Recovery Manager (RM) as shown in figure 1.

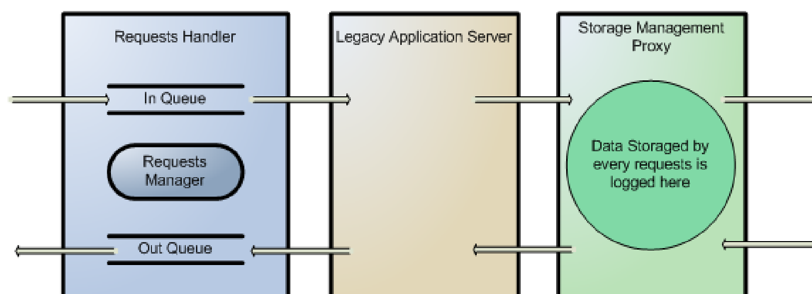

Figure 1. Basic Architecture diagram

The Requests Handler has the responsibility to capture all HTTP requests that are sent to the application server from a potential end-user or a web server. Two queues and one requests manager form the RH. There is one requests queue, a responses queue, and a requests manager to manage and synchronize them.

When a new request is sent to the application server, RH handles the request and copies the request to the requests queue and then the request is redirected to the application server without any modification. When the application server sends a response to the end-user or web server, the Requests Handler captures the response and copies the response to the response queue and redirects the response without changes to the end-user.

To achieve this behavior the Requests Handler has to work in the same network domain as the application server and has to be configured to work in a promiscuous mode, and by using the application server IP it can capture all the application server requests and discard all not relevant requests (e.g. non-HTTP requests). Capturing the response is quite more complicated because, the Request Manager has to save all IP sources from all requests without response in the requests queue and any packet sent by the application server to one of the IP's from the requests queues will be captured. When the response is captured, the request associated to the response is removed from the queue. We are based on the idea that the requests and responses can be joined if we understand that the requests from one end-user are sequential and the responses too, for this reason, using the source IP of the request and the destination IP of the response can be enough to join one response with its request.

The Requests Handler has more important tasks other than only preserving a copy to know what in-flight requests are there in the application server. It has also the responsibility of detecting potential failures (transient failures and crashes).

Based on the idea proposed by [7] and [11], every time that one request is sent to the application server from the Requests Handler, a timeout is activated. If the request timeout finishes without response, the Requests Handler tries to ping over the application server and if there isn't an answer, the Requests Handler assumes that the application server has crashed and it starts a recovery action. Furthermore, to detect fine-grain potential failures like application failures, the Requests Handler reads the every response content to try to detect potential transient or intermittent failures (e.g. any HTTP 400 error) and notifies this fact to the developers or administrators and applies a recovery action discarding the response message failure to avoid the potential concerns to the end-user. The Storage Management Proxy (SMP) is based on the idea proposed in [14]. [14]proposes a new ODBC for communications with data base servers which understands all SQL statements as a transaction and mod-

ifies the statements syntax to achieve a successful behavior. This ODBC is integrated in Phoenix APP [10], to achieve a system with crash-safe processes.

We have simplified the idea proposed in [14]. Our system only stores information of the communications between the application server and any data storage device (database servers or session state storage). The information obtained from these communications will be used only in the recovery process and during a correct behavior of the application server. We can say that SMP is only a logging system that saves all requests and responses' information. If a crash happens during a transaction process in the database server, the transaction will rollback and the SMP will write this event only to keep track of what happened during this interaction between the application server and the database server. Finally, the Recovery manager has the responsibility to coordinate the recovery process. To define the recovery process, we have designed an architecture and a process to avoid the potential problems of the recovery process described in [13]: exactly-once execution, (where the latter means, no output to the user is duplicated to avoid confusion), the user provides input only once (to avoid user irritation) and the user attempt is carried out exactly once. If the Requests Handler detects a failure a signal is triggered to the Recovery manager to start a recovery process. First, the RM notifies this situation to the RH and SMP. When the RH receives the notification, it stops to redirect requests and redirect responses to and from the application server and it only receives requests from end-users. At the same time, the SMP avoids all communications between the application server and the data storages. After these both processes are concluded, the RM recovers (crash reboot) the application server (e.g. kill -9 pid-process) and restarts the application server waiting for a new application server instance. When the new instance is running, the RM notifies the fact to the RH and SMP. The RH redirects all requests without response (in-flight requests) to the new instance of the application server again, so the user doesn't need to provide the input again. If at any time, the RH receives a response without request associated to it, the response is discarded to avoid potential duplicates to the user in order to avoid confusion.

At the same time the SMP is monitoring the communications between the application server and avoiding potential duplicated database or session object modifications. When SMP detects a duplicated communication, the packet is not redirect to the storage if the performance of this communication was successful, otherwise the communication is redirected. On the other hand, if the communication wasn't successful, we use the SMP response saved to build a new (old) response and send it to the application server as if it'd been stored. Thanks to this control of the duplicated communications of the SMP we avoid potential problems presented in [13].

The idea of this coordination between RH and SMP using the RM is to avoid the potential crash hazards presented in [11–12]. In these papers, the authors

present an architecture based on interact contracts. These contracts are thought to apply safe-recovery mechanisms after a crash, replaying all requests without response before the crash and avoid uncomfortable behaviors of the application servers. However, authors present a solution that has an important constraint for all components: even internal components have to keep the contracts to maintain the coherence of the architecture. We have used the idea presented in these papers to present the architecture to achieve crash-only software legacy application server and mask the failure to the user.

4. How our architecture achieves the goals?

In this paper, we propose a new architecture based on two proposals to achieve a crash-safe and fast recovery. Our goals proposed at the beginning of this paper were to migrate the crash-only software concept to achieve a legacy application server with the same characteristics as a crash-only designed system. Moreover, we proposed the improvement with regards to results presented in [6] where when using the crash-only concept, the system had an unavailability time of 78 missed requests during a microreboot process based on crash-only software. We want to design an environment to avoid these missed requests, reducing to zero downtime if it is possible. We understand a legacy application server as an indivisible component to make possible the crash-only concept migration, because the crash-only software is made-up by crash-only subcomponents. Based on this premise, it is easy to understand the reasoning of how our architecture preserves the properties of the crash-only software. As we have mentioned, we have restricted our study to the "stateless" application servers: The session state is preserved in external session state storages, accomplishing the first property (a). The architecture alone cannot achieve the second property (b), though we can use virtual machines (VM) to run the legacy application server and the rest of our proposal's components: one VM for each component to guarantee the isolation between components. The third property (c) is guaranteed by the RH, the SMP components and the behavior of the application servers. All communications have a timeout configured at least at OSI 4-layer (e.g. TCP/IP protocols). The fourth property is more difficult to accomplish. Working with legacy application servers, this property has to be delegated to the Operating system, which guarantees that all resources are leased. Finally, the HTTP requests, the traditional type of message for application servers, which are completely self-described. The secondary goal is preserved thanks to the queues inside the Requests Handler. During the crash and the recovery process, the application server is unavailable for the end-users or third applications. However, our Requests Handler continues capturing requests for the application server like a proxy and when the recovery process finishes, these requests waiting on the queue will be redirected to the appli-

cation server like nothing happened. This process can mask the crash for the end-users in most cases like a previous work [9] where the solution masked potential service degradation and reboot process. Our proposal also avoids error messages if it is possible, because the RH parses the requests to try to detect fine-grain application failures or the application server failures (e.g. 400 and 500 HTTP errors) avoiding that end-users may observe neither these transient failures nor temporary system unavailability. In our proposal we get crash-safe self-recovery legacy application servers and even our solution offers a "fast" recovery process. We can confirm that the Requests Handler simulates a non-stop service which is the maximum speed of the recovery, and though the end-users will suffer response delays during the recovery process, we think that that penalty delay is proportional to the advantages of the proposal.

5. Conclusions

We have presented an architecture to migrate the advantages of the crash-only software concept to the legacy application servers. Furthermore, we have to improve the crash-only designs potential by introducing the idea behind the interact contracts presented in the Phoenix project [10] in order to achieve a successful and useful self-recovery process without modifying neither the application server code nor the application code. Nevertheless, our proposal has to have all application' and all application server' information to modify the behavior of the Requests Handler and the Storage Manager Proxy to correctly capture the requests and responses and use the correct network protocol (e.g. TCP/IP, HTTP, SOAP or others).

Our solution introduces a potential time-to-service delay during the recovery process. We could reduce this time if we introduce a hot-standby application server waiting to substitute the failing server. This idea is proposed in [9] for different environments with promising results.

Acknowledgments

This research work is supported by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01. We thank Ceila Hidalgo Sánchez for her contribution during the review process.

References

[1] J.Hennessy, D.Patterson. *Computer Architecture: A Quantitative Approach*, Morgan & Kaufmann Publishers, 2002.

[2] D. Patterson, et. al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies.*, Technical Report UCB CSD-02-1175, U.C. Berkeley, March

2002.

[3] K. Vaidyanathan and K.S. Trivedi. *Extended Classification of Software Faults based on Aging*, In Fast Abstracts, Proc. of the IEEE IntŠl Symp. on Software Reliability Engineering, Hong Kong, November 2001.

[4] D. Scott. *Operation Zero Downtime* A Gartner Group report, Donna Scott, 2000

[5] Chet Dembeck. *Yahoo cashes in on Ebay's outage*, E-commerce Times, June 18, 1999. [web] http://www.ecommercetimes.com/perl/story/545.html

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. *A Microreboot - A Technique for Cheap Recovery* Proc. 6th Symp. on Operating Systems Design and Implementation (OSDI), Dec. 2004.

[7] G.Candea, A.Fox. *Crash-only Software*, Proc. 9th Workshop on Hot Topics in Operating Systems, Germany, 2001

[8] B. Ling and A. Fox. *A self-tuning, self-protecting, selfhealing session state management layer*, In Proc. 5th Int. Workshop on Active Middleware Services, Seattle, WA, 2003.

[9] Luis Silva, Javier Alonso, Paulo Silva, Jordi Torres and Artur Andrzejak. *Using Virtualization to Improve Software Rejuvenation* The 6th IEEE International Symposium on Network Computing and Applications (IEEE NCA07), 12 - 14 July 2007,Cambridge, MA USA

[10] Roger S. Barga. *Phoenix Application Recovery Project* IEEE Data Engineering Bulletin, 2002.

[11] Barga, R., Lomet, D., Paparizos, S., Yu, H., and Chandrasekaran, S. *Persistent applications via automatic recovery*, In Proceedings of the 17th International Database Engineering and Applications Symposium, Hong Kong, China, July 2003.

[12] R. Barga, D. Lomet, G. Shegalov, G. Weikum. *Recovery Guarantees for Internet Applications*, ACM Transactions on Internet Technology (TOIT), vol. 4, no. 3, pp. 289-328, 2004.

[13] R. Barga, D. Lomet, G. Shegalov, G. Weikum. *Recovery Guarantees for General Multi-tier Applications*, Proc. of the 18th Int. Conf. on Data Engineering, p. 543, Feb. 26-March, 2002.

[14] Roger S. Barga , David B. Lomet. *Measuring and Optimizing a System for Persistent Database Sessions*, Proc, of the 17th Int. Conf. on Data Engineering, p.21-30, April 02-06, 2001.

[15] M. Stonebraker. *The design of the Postgres storage system*, Proc. 13th Conf. on Very Large Databases, Brighton, England, 1987.

[16] R. Figueiredo, P. Dinda, J. Fortes. *Resource Virtualization Renaissance*. IEEE Computer, 38(5), pp. 28-69, May 2005

[17] *[website] http://roc.cs.berkeley.edu/*