Towards Resilient Java Computational Programs

Queen L. Nguyen, Arun Sood

Towards Resilient Java Computational Programs

Quyen L. Nguyen
Computer Science, George Mason Univ., Fairfax, USA
U.S. Census Bureau, Suitland, MD, USA
qlnguyen@yahoo.com, quyen.nguyen@census.gov

Arun Sood
Computer Science, George Mason Univ., Fairfax, USA
SCIT Labs, Clifton, VA. USA
asood@cs.gmu.edu, asood@scitlabs.com

Abstract— **With the proliferation of sensors and web logs, big data are generated in large volume and high rate. The analytical processing of these data usually requires long duration computational transaction. To ensure resiliency of such computational services, both in terms of data integrity and security, we propose to apply the SCIT (Self Cleansing Intrution Tolerance) approach, that consists of frequently restoring systems to their pristine state. This paper describes JSCIT, a combination of SCIT and Java Coroutine. JSCIT can be deployed on compute nodes running Java-based computational programs to migrate the in-progress computation between virtual machines or containers for the cleansing process.**

Keywords—SCIT; Resilient; Coroutine

I. Introduction

With the proliferation of sensors and web logs, big data are generated in large volume and high rate. Due to the big volume, data analytics is compute intensive, and consists of long duration computational transaction. In Census business domain, adaptive survey design, which allows dynamic operation changes while the survey is in progress, relies on the analytics of large body of data and paradata to optimize data collection rules and cost efficiency. The challenge is to ensure the resiliency of computational services against both system and malicious faults in order protect the security and integrity of data and analytical results. While current Information Technology systems are *static*, meaning they operate in a relatively static configuration and primarily focus on intrusion avoidance, SCIT approach is proactive by frequently restoring the system to a pristine state [2]. In [5], it was proved via the analysis of stochastic process that this continuous restoration can improve service's resilience, expressed by metrics such as MTTSF (Mean Time to Security Failure), by tuning the exposure window. Moreover, for long computational programs, the proactive restoration must also ensure the resumption of the program while performing the cleansing and migration.

In our research, we propose JSCIT as an approach to automate the periodic cleansing of long running Java programs in order to improve their resiliency. At the high level, the concept is similar to software rejuvenation proposed in [3]. But, in addition to ensure resiliency from system faults, JSCIT aims at protecting Java programs against malicious intrusions. JSCIT mechanism makes use of Coroutine to perform program checkpoints during the cleansing phase of Java programs, which can run on platforms other than Unix. In Computer Science, Coroutine is more general than Routine [4]. With *yield* and *resume* operations, Coroutine allows multiple points of suspension and resumption; thus, a Coroutine can resume execution from its previous suspension point, while with its local data can be persisted through successive invocations. While some programming languages have built-in Continuation constructs, for others such as Java, Coroutine and Continuation are add-on features. The implementation of Coroutine is usually based on Continuation, whose structure contains state information. The contribution of our research is to: a) extend the SCIT approach to improve resiliency of long running computations; b) utilize Coroutine to migrate and restore Java-based programs. Section II will give an overview of JSCIT approach; Section III describes our proof of concept experiment; and the conclusion is in Section IV.

II. JSCIT Approach

A. Overview

SCIT is a time-based restoration and recovery intrusion tolerance scheme [5]. The goal of SCIT is to make applications and services resilient in the face of malicious attacks. SCIT's automated periodic cleansing of nodes running on a virtualization layer facilitates the instantiation of multiple compute nodes with possibly different guest operating systems for diversity. Moreover, the utilization of virtual machines enables rapid reloading and reactivation of the servers. The main functionality of SCIT is node cleansing, which consists of bringing a node back to a pristine state based on a virtual image maintained, patched, and enhanced in a safe offline storage. It is the SCIT Controller that manages the rotative cleansing of the nodes.

As an extension of the SCIT approach targeted for long running Java computational programs, JSCIT is founded on the concept of data segmentation and partial cleansing. Indeed, we consider runtime data and code as separate segments. Unlike *full* cleansing mode, where the compute nodes undergo complete erasure to make place for the clean image*,* JSCIT performs cleansing of the code segment while carrying over the runtime data necessary for the analytic computation during the migration. Full cleansing is mostly used for services with short transactional operations, which can be implemented as stateless RESTFul web services. Since the service operations are stateless, there is no need to capture in-flight information before performing the cleansing. On the contrary, *partial* cleansing is used for services with long running operations. Scientific computation, and big data analytics fall into this category. Given that state information exists when a node enters the cleansing phase, using full cleansing mode may unintentionally cause loss of process state information, and data being processed. Hence, the on-going computation may be corrupted. At the cleansing instant all but the runtime data are restored to their pristine state. Thus malicious code is deleted, and application processing continues. There are two phases for JSCIT, Preparation and Runtime described below.

B. Preparation Phase

Let A be the computation application implemented by the Java code P_1. In order to improve the resiliency of A by applying JSCIT, the following preparation steps need to be performed upon P_1 to produce the associated instrumented program bytecode P_3:

1. Wrap program P_1 within a Java Coroutine runner to ;
2. Determine a point or points in the program to suspend;
3. Instantiate a Continuation object in P_2;
4. Compile the program P_2 into Java bytecode.
5. Instrument the bytecode P_2 to generate the to-be used bytecode P_3 in order to enable Coroutine functionalities.

During the runtime phase, the SCIT Controller will utilize the bytecode P_3 thus obtained to perform periodic migration and cleansing.

C. Runtime Phase

In what follows, a Container contains the operating system and the Coroutine-instrumented bytecode P_3. The SCIT's algorithm that is based on the exposure time and described in detail in [2] remains the same in JSCIT. What changes is the step to make a current Online Container offline, and replace it by a new and pristine Container. Let C_1 and C_2 be the currently online and replacement containers respectively.

During the SCIT rotation, the Controller will pre-load C_1 and C_2 with the instrumented bytecode P_3 of application A. We design the new scheme to replace C_1 by C_2 as follows:

1. Suspend the execution of A in C_1;
2. Serialize the Coroutine containing runtime data of the in-progress application A in C_1;
3. Write this serialization to a file F_S in a secure storage;
4. Make C_1 offline;
5. Move the file F_S to C_2;
6. Make C_2 online;
7. Resume the execution of A in C_2.

Note that the code loaded in C_2 is the pristine code that has been obtained via the Preparation Phase. What has been retained from C_1 is the runtime data portion captured using the Continuation mechanism, in order to continue the execution of the computational program on a pristine container.

III. Proof of Concept

For the proof of concept, we use the Java Toolkit library for Coroutines and bytecode instrumentation [1]. Specifically, we use the .jar files: user-1.1.1.jar and instrumenter-1.1.1.jar. For using this Coroutine toolkit, we have to implement the Java interface *com.offbynull.coroutines.user.Coroutine*, with the public method *run(Continuation c)*. In the experiment, The JVM platform was Java JDK 1.8.0_51. The container C_1 was run on a Dell Inspiron N411Z laptop with the following configuration: Intel Core i5-2450M CPU, 2.50 GHz, 6 GB RAM, and 64-bit Windows 7 Home Premium. The container C_2 was run on a virtual desktop on Amazon Cloud Web Services. The application A is a simple program that does matrix multiplication. We chose the use case of transition matrix of ergodic Markov chains, which will converge to steady-state values. Both containers are pre-loaded with the instrumented bytecode of application A; only the serialized Coroutine data are migrated. From the experiments, the observation that iterative multiplications of Markov transition matrices converge demonstrates that the migration from C_1 to C_2 and vice-versa does not corrupt, but yield correct results. In the context of this research, the common thread is the size N of the matrix; we can thus vary N to make the computation duration longer or shorter. The rotation time is comprised of data (de)serialization time, and transfer time, which depends on the network performance. Table 1 shows some numerical examples.

Table 1. Matrix Dimension N, Data Size S, and (De)serialization Time D**.**

N	10	20	50	100	200	500	1000
S (in KB)	3	11	61	238	945	5875	23468
D (in ms)	126	141	156	156	234	342	889

IV. Conclusion

The significance of the approach described in this paper is two-fold: a) JSCIT can be used to protect a long running computational Java program; and b) there is a potential that SCIT can be extended to the "micro-level", i.e. it is possible to perform rotational cleansing at the object level, instead of at the Virtual Machine level as in [2]. For future work, we plan to run experiments with other well-known Linear Algebra problems, such as Gaussian Elimination, LU Decomposition, and Cholesky Decomposition. Linear Programming problems will also be considered. We also would like to investigate the application of the JSCIT approach to an application fulfilled by a COTS (Commercial Off -the-Shelf Software). Finally, we will research the possibility to incorporate Popcorn Linux [6] into the migration of programs between kernel instances, which may eliminate the cost of transferring the data segment by utilizing its Global Accessible Memory.

Acknowledgment

This research was partially supported by Office of Naval Research contract N000141310017. The views expressed in this paper are solely those of the authors, and not of their affiliated organizations.

References

[1] Coroutines Project. https://github.com/offbynull/coroutines.

[2] Y. Huang, D. Arsenault and A. Sood, "Incorruptible system self-cleansing for intrusion tolerance," *2006 IEEE International Performance Computing and Communications Conference*, Phoenix, AZ, 2006.

[3] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "Software rejuvenation: analysis, module and applications," *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, Pasadena, CA, USA, 1995, pp. 381-390.

[4] D.E.Knuth. "The Art of Computer Programming, Volume 1: Fundamental Algorithms".Addison-Wesley Pub. Co., MA, 1973.

[5] Q. Nguyen. "Quantitative Framework to Design Services with Intrusion Tolerant QoS". Ph. D. Thesis, 2014. http://digilib.gmu.edu/xmlui/bitstream/handle/1920/8925/Nguyen_gmu_0883E_10622.pdf?sequence=1&isAllowed=y

[6] Popcorn Linux. http://www.popcornlinux.org/.