Component-Dependency based Micro-Rejuvenation Scheduling

Vinaitheerthan Sundaram, Matthew Tan Creti, Rajesh K. Panta, Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University
{vsundar,mtancret,rpanta,sbagchi}@purdue.edu

1. Introduction

With the growth of Internet, "always on" services are becoming increasingly important. Software rejuvenation is a well-known proactive technique to prevent failures due to software aging and extend the lifetime of long-running software such as Internet Servers, billing systems, telecommunication switches [1]. However, doing a machine reboot to rejuvenate takes time in the order of minutes and can be expensive for large-scale Internet systems, even when clusters are employed[2]. To reduce the downtime caused by machine reboot, a fine-grained reboot technique called micro-reboot, which reboots components, was proposed in [2]. Micro-reboot has been shown to be an order of magnitude faster than machine reboot and systems can be rejuvenated by parts without ever doing a full reboot. Micro-rejuvenation [2] is a proactive technique that uses micro-reboot to prevent ageing failures. Since micro-rejuvenation involves rebooting components, it is necessary to rejuvenate a component's transitive closure of dependents [2].

A complete rejuvenation schedule specifies the rejuvenation time instants so that the cost of rejuvenation is minimized. The complete rejuvenation can occur due to periodic or random timer trigger, transaction load trigger, or failure prediction trigger [1,3,4]. Similarly, micro-rejuvenation can occur due to all three of these triggers. However, a micro-rejuvenation schedule should specify both the rejuvenation time instants as well as the component(s) to be rejuvenated. The micro-rejuvenation schedules are less well-studied in the literature than complete rejuvenation.

One approach to micro-rejuvenation scheduling proposed in [2] rejuvenates if utilization of system resources, such as memory, is above a certain threshold. The components are rejuvenated until the system resources reach a normal level. The components rejuvenated are selected in the order of amount of resources released in earlier rejuvenations. Another approach can be to adapt the complete rejuvenation schedule to micro-rejuvenation schedule. For example, if the complete-rejuvenation schedule is load-triggered, that is, the system is rejuvenated when the load is greater than a threshold; the adapted micro-rejuvenation schedule will use component load-triggered, that is, the component is rejuvenated if the component load is greater than a threshold. We refer these two approaches as simple scheduling, because both approaches take the simplified approach that all components are independent and will have the same cost in rejuvenating. Both the simple scheduling approaches are easy to implement, however, they do not take into account the dependency of the components and therefore, can yield non-optimum schedules. When a component is rejuvenated, all its dependent components are rejuvenated and therefore, any optimum schedule must take into account the fact that dependent components will be rejuvenated twice or more.

In this paper, we propose a dependency-aware micro-rejuvenation scheduling policy that uses load-based trigger and rejuvenate independent components only. We develop a SAN model that closely reflects the real system and used it for evaluating our scheduling policy.

2. Dependency-Aware Scheduling

We consider transaction based software systems such as Internet servers, billing systems in which a transaction on a component will cause transactions on all of its dependencies. In such a system if component C_1 is dependent on C_2 then the number of transaction on C_2 will always be larger than the number of transactions on C_1 and thus, C_2 will always have a larger age than C_1. Moreover, when C_2 is rejuvenated, C_1 is also rejuvenated because of dependencies. Furthermore, the dependency graph cannot have cycles as otherwise, the transactions executing in a cycle will lead to live-lock. Therefore, it is enough to rejuvenate independent components. Thus, our scheduling policy, dependency-aware scheduling, looks only at the components which have no dependencies and decides when to rejuvenate those components based on the load on the components..

3. SAN Model and Simulation Results

To demonstrate that component dependency based rejuvenation can increase availability, we implemented the SAN model [4]. We present a simple SAN model where we model a system with only two components C_1 and C_2 and the component C_1 depends on C_2. The SAN model is shown in Figure 1 and can be broken down into three parts, namely, component simulation (middle left), rejuvenation simulation (top and bottom right) and failure simulation (middle right).

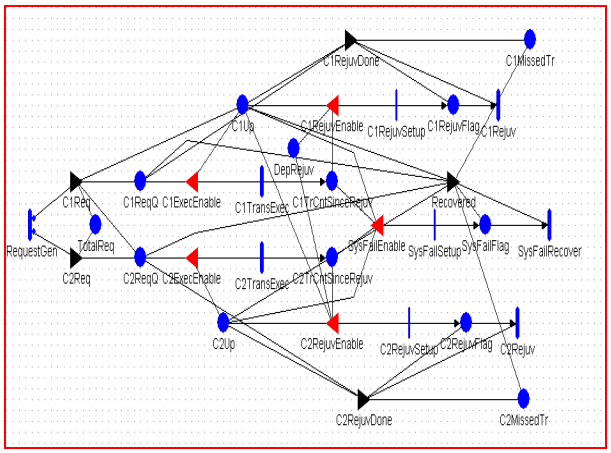

Figure 1: SAN Model for two dependent components

We assume that the transaction requests arrival and service rates are exponentially distributed with rates TRANS_REQ_RATE and TRANS_SERVICE_RATE. A transaction request is serviced by component C1 or C2 with equal probability. The rejuvenation for both components are simulated as timed activity with exponential distribution with rates C1_REJUV_RATE or C2_REJUV_RATE. Likewise, system failure is simulated as timed activity with exponential distribution with rate FAIL_RECOVER_RATE. All transactions entering the system are counted by the places C1ReqQ or C2ReqQ. The transactions entering during rejuvenation or failure are not executed and are separately counted by the places C1MissedTr and C2MissedTr.

The transactions execution of each component is simulated. Despite rejuvenation, failures can occur in the system. When the component is rejuvenated or failed, the model captures the number of transactions missed. We assume that all transactions cause equal aging to any component that is involved with it. We have used load-based rejuvenation scheme, in which each component is rejuvenated when it has executed some threshold amount of transactions. To evaluate our simulation, we inject failures into our system as follows. The age of a component increases its likelihood of failure. The detailed simulation results are discussed next.

Table 1. Parameter values used in our simulation

Parameter	Value
TRANS_REQ_RATE	0.1 (/seconds)
TRANS_SERVICE_RATE	1 (/seconds)
C1_REJUV_RATE	0.05 (/seconds)
C2_REJUV_RATE	0.05 (/seconds)
FAIL_RECOVER_RATE	0.005 (/seconds)
C1_REJUV_THRES	2000 (Transactions)
C2_REJUV_THRES	2000 (Transactions)
SYS_FAIL_THRES	500 (Transactions)

Since rejuvenation affects availability of the system, we plot availability of the system against aging. The availability of the system is defined as the ratio of the number of transactions executed to the number of transactions entered the system. We ran the SAN simulation using Mobius [4]. The parameters used in the experiment are shown in Table 1. We recorded the availability for different loads or transaction rates.

The results are shown in Figure 2. As the load increases, the number of missed transactions increases and therefore, the availability decreases in all cases. In the case of no rejuvenation, the system takes longer time to recover, which can be seen from the faster decrease of availability. Comparing the simple and dependency-aware scheduling policies, we see that dependency-aware scheduling policy improves the availability up to 3.38% over the simple scheduling policy. This is because in the latter case, the component C_1 gets rejuvenated independently as well as when component C_2 was rejuvenated. However, in the former, the component C1 is rejuvenated only when the component C_2 is rejuvenated.

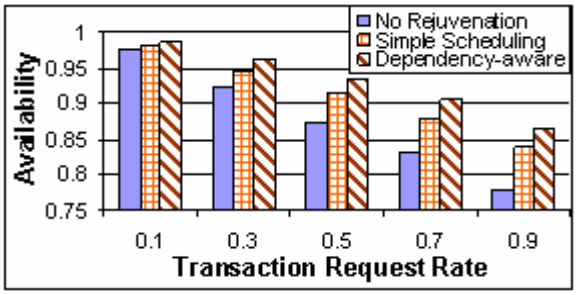

Figure 2: Comparison of availability for different rejuvenation scheduling policies.

4. Conclusions and Future Work

In this paper, we presented a new dependency-aware micro-rejuvenation scheduling policy. We created an SAN model to evaluate our policy and showed that it can improve availability in transaction-based systems. In future, we would like extend our SAN model to several components and complex interactions. We are also implementing the proposed scheduling policy in J2EE servers such as JBOSS and GlassFish.

5. References

1. Y. Huang, C. Kintala, N. Kolettis and N. Fulton. Software Rejuvenation : Analysis, Module and Applications. In IEEE Intl. Symposium on Fault Tolerant Computing, FTCS 25.
2. George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In OSDI 2004.
3. S. Garg, A. Puliafito, M. Telek and K. S. Trivedi. Analysis of Preventive Maintenance in Transactions Based Software Systems. In IEEE Transactions on Computers, Jan 1998.
4. A Performability-Oriented Software Rejuvenation Framework for Distributed Applications. A. T. Tai, K. S. Tso, W. H. Sanders, and S. N. Chau. A In IEEE DSN-2005