Quantitative evaluation of software rejuvenation of a pool of service replicas

Leonardo Scommegna
Dept. of Information Engineering
University of Florence
Florence, Italy
leonardo.scommegna@unifi.it

Marco Becattini
Dept. of Information Engineering
University of Florence
Florence, Italy
marco.becattini@unifi.it

Giovanni Fontani
Dept. of Information Engineering
University of Florence
Florence, Italy
giovanni.fontani1@unifi.it

Leonardo Paroli
Dept. of Information Engineering
University of Florence
Florence, Italy
leonardo.paroli@unifi.it

Enrico Vicario
Dept. of Information Engineering
University of Florence
Florence, Italy
enrico.vicario@unifi.it

***Abstract*—Cloud-based systems require the management of large volumes of requests while maintaining specific levels of availability and performance. Each service is thus replicated into a pool of identical replicas. This allows for load distribution among the pool of replicas and a greater degree of fault tolerance compared to a single instance of the service that stands as a single point of failure. The high availability and scalability requirements, coupled with the phenomenon of software aging, have made the replica-based approach pervasive in modern online services. In such configurations, the unavailability of a single replica, due to scheduled maintenance or unexpected failures, does not imply the unavailability of the whole system but rather an increase in the load of the remaining replicas. This identifies a performability problem in which the system can tolerate a certain number of offline replicas in the pool. However, once a certain threshold is exceeded, the resulting high workload pending on the online replicas could degrade the performance of the system, potentially leading to a failure in meeting the non-functional requirements.**

In this work, we study the problem of aging in a pool of service replicas. We characterize two inspection-based rejuvenation strategies that could be implemented in this context, which we identify as uncoordinated and coordinated rejuvenation. We represent them through the formalism of Stochastic Time Petri Nets (STPN) and through steady-state analysis, we conduct a performability evaluation of both the models as the frequency of inspections and the pool size vary.

***Index Terms*—Software aging, Software rejuvenation, Cloud-based systems, stochastic time Petri nets.**

I. Introduction

Nowadays, many web applications and online services need to handle a significant number of simultaneous requests and contemporarily provide scalability and highly available services [1]. To manage these requirements, it is considered best practice to use multiple replicas of the same service [2], [3]. As a first advantage, having multiple replicas enhances the service availability [4]. If one replica fails or goes offline for any reason, the other replicas can continue to handle requests, thereby ensuring service continuity. Moreover, the use of multiple replicas enhances the maintainability and flexibility of the system. In fact, a gradual increase of active users can be effectively managed by incrementing the number of replicas. This allows for the development of systems that can smoothly evolve over time, allocating and consuming resources proportionally to the workload expected for the near future, without the need for long-term resource provisioning.

In principle, this replica-based strategy could be implemented manually. This requires manual management of the number of replicas to instantiate, the implementation of an appropriate horizontal scaling strategy, and the implementation of a load balancer capable of identifying the available replicas and adequately distributing the workload. With the advent of the virtualization era and the cloud-native application paradigm, cloud providers like Google Cloud and Amazon Web Services, and orchestration frameworks like Kubernetes and Docker Swarm have emerged. These platforms offer native support for replica-based strategies, which has greatly simplified the process of deploying and managing software systems efficiently. This ease of use, combined with the inherent benefits of replication for availability and maintainability, has led to the widespread adoption of replication-based strategies in the cloud computing landscape. In essence, the combination of virtualization strategies, advanced cloud platforms, and sophisticated orchestration tools has made replication not just a viable, but a prevalent strategy for managing reliable, available, and efficient services in the cloud.

As a downside, long-running systems are prone to software aging [5], [6], and as such, also virtualized and cloud-based systems are subject to this problem [7]–[11]. Software aging is a phenomenon that, due to aging-related bugs (ARBs), affects the system resources, gradually degrading its performance.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on “Telecommunications of the Future” (PE0000001 - program “RESTART”).

As time passes, the state of degradation can increase until it manifests an aging-related failure that can even cause a system crash. Aging is often caused by the accumulation of errors due to mandelbugs, which are ARBs that are difficult to replicate and remove [12]. Consequently, it is a common practice to implement runtime proactive strategies to counteract software aging. Such strategies are commonly defined as software rejuvenation and consist in the periodic preemptive rollback of running applications with the aim to prevent, or at least postpone, aging-related failures [7].

In virtualized and cloud-based systems, the problem of software aging and the strategies of rejuvenation are conditioned by various additional factors. On the one hand, software aging is amplified due to the various middleware components involved, which could contribute to system aging [7], [10]. For instance, Operating Systems, Hypervisors, and Virtual Machine Monitors (VMMs) can be susceptible to software aging [13], [14]. Similarly, orchestrators can also be subject to this phenomenon [15], [16]. On the other hand, the use of replica-based strategies makes the system as a whole more robust to the downtime of a single replica either due to crashes or rejuvenation actions. In this scenario, it becomes crucial to implement software rejuvenation strategies that take into account the current technological context and the replica-based strategies that are now employed pervasively.

Many previous works address the problem of aging in a cloud computing environment [10], [17]–[19] and many others analyze virtualized systems [11], [20]–[22]. Popular rejuvenation techniques, both measurement-based and model-based [7], primarily focus on studying the aging phenomenon limited to a single instance of a service [10], [23], [24]. However, some specific works address rejuvenation in virtual systems with various degrees of granularity considering warm or cold virtual machine replicas or passive or active VM failover [7], [11], [25], [26]. There are also works where the case of a pool of active replicas is considered, both through the concrete implementation of rejuvenation strategies on real systems [27], and through the analysis and quantitative evaluation of models [28]–[30]. However, to the best of our knowledge, no work in the literature proposes a model capable of representing and analyzing an inspection-based rejuvenation strategy.

In this work, we study how software rejuvenation strategies impact a pool of service replicas subject to software aging. Specifically, we quantitatively evaluate reliability and unavailability metrics of the replicas to estimate how specific pool configurations and inspection-based strategies can impact the performability of the entire pool. To do this, we characterize two rejuvenation strategies that can be implemented within a pool of service replicas, utilizing the most widely used orchestration frameworks. The first strategy, which we refer to as uncoordinated rejuvenation, involves an independent inspection and rejuvenation process for each replica. The second strategy instead, termed coordinated rejuvenation, involves a pool-wise orchestrated inspection and rejuvenation process. Finally, the two identified strategies are represented with two Stochastic Time Petri Nets (STPN) [31] underlying a Markov Regenerative Process (MRP) [32]. The numerical solution is performed using the method of stochastic state classes [33], as implemented in the SIRIO library [34] of the ORIS tool [31].

A complete replication package of the study is made publicly available[1] This includes: i) the STPN Oris models, ii) The Java project that utilizes the SIRIO Library, iii) the raw data collected from the experiments, and iv) the corresponding plots.

The rest of the paper is organized as follows. In Section II, we describe the system model considered and we define the technological context in which we place our work. In Section III, we outline two alternative rejuvenation strategies, providing an STPN model for each. In Section IV, we identify the features of interest for replica-based systems and we analyze and compare the results obtained with the two strategies. Finally, in Section V, we draw the conclusions.

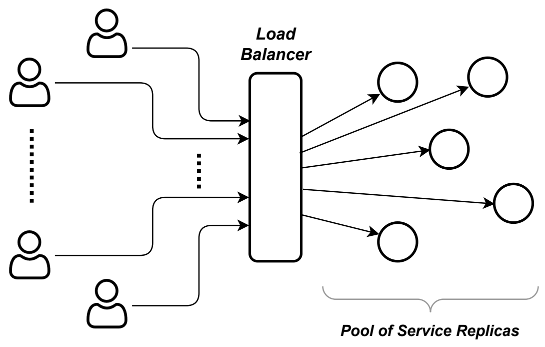


Fig. 1: Graphical representation of the considered system model.

II. System Model

In this section, we identify the system model assumed in this work. As depicted in Figure 1, we consider a pool of service replicas. Service replicas could be hosted within the same machine in the form of virtualized services, such as virtual machines or containerized services. Equivalently they could be hosted on separate machines within the same cluster in the form of non-virtualized services.

We assume the presence of a load balancer [35]. The primary responsibility of this component is to evenly distribute the workload among all the replicas. Furthermore, the load balancer also identifies which replicas are still active and which ones have failed. In case of failed replicas, it activates the repairing process and redirects requests to the available replicas. This type of setup is standard in many enterprise infrastructures and can be replicated either manually, provided natively by cloud providers such as Google Cloud or Amazon Web Services, or by orchestration frameworks like Kubernetes or Docker Swarm.

Software aging is a phenomenon impacting long-lasting software systems, both virtualized and non-virtualized. Furthermore, the workload that the subject handles influences

[1]https://doi.org/10.5281/zenodo.13370985

significantly this process [7], [18]. In our system model, replicas of the same service type, belonging to the same pool, experience analogous loads thanks to the load balancer. Consequently, we can assume that the impact of aging on these replicas occurs concurrently and follows the same aging pattern. Note that this indicates only an average aging trend for each replica and does not imply synchronized degradation or failures. While this assumption appears reasonable, aging may also be influenced by other additional factors, therefore, we regard this assumption as a potential threat to validity.

In order to mitigate the effects of software aging, a variety of software rejuvenation strategies are utilized. One such widely adopted strategy, particularly within virtualized environments, is known as inspection-based rejuvenation. This strategy entails conducting regular inspections of the system replica to determine if rejuvenation should be applied. The aging assessment of each inspection relies on the analysis of specific resource usage metrics, known as aging indicators, that can act as a proxy measure of the aging of a system [7]. Monitoring of the aging indicators helps to estimate the aging state of the system while maintaining low overhead. However, these metrics can occasionally be misleading, resulting in misclassification [7], [24]. Thus, a certain degree of sensitivity and specificity should be considered for each inspection.

The outlined scenario identifies a system designed to maintain high availability even in the presence of failures. However, although redundancy allows the service to remain accessible even in case of multiple failures that occur close together in time, load rebalancing and the consequent drop in performance could lead to a violation of the non-functional requirements. This identifies a system performability problem where the performance of the system and its dependability are studied together. However, even if a failure does not imply the unavailability of the system, it corresponds to the unplanned interruption of a replica. Therefore, a failure identifies a potential interruption of a response that was being processed by the failed replica, causing a tangible disservice to the end user.

In this context, it is crucial to study the behavior of the system as the number of replicas in the pool (the pool size) and the frequency of inspections vary. The metrics of interest that we identify for this work are *reliability*, *performability*, and *unavailability due to rejuvenation*. Specifically, by reliability, we refer to the probability of a replica experiencing a failure. Performability involves the probability of having a certain number n of unavailable replicas, either due to failure or because they are in rejuvenation. Where n represents the threshold within which the QoS requirements are no longer guaranteed. Finally, we are interested to the unavailability due to rejuvenation, since it allows us to evaluate the impact of the rejuvenation practice on performability.

III. Inspection-Based Rejuvenation in Pools of Service Replicas

In this section, we characterize two inspection-based rejuvenation strategies and their corresponding models in the form of STPNs. In Section III-A, we report a strategy that we define as uncoordinated rejuvenation, in which each replica is analyzed independently of the others. In Section III-B, instead, we present a coordinated strategy that consists of periodically identifying the most aged replicas within the pool and rejuvenating them sequentially.

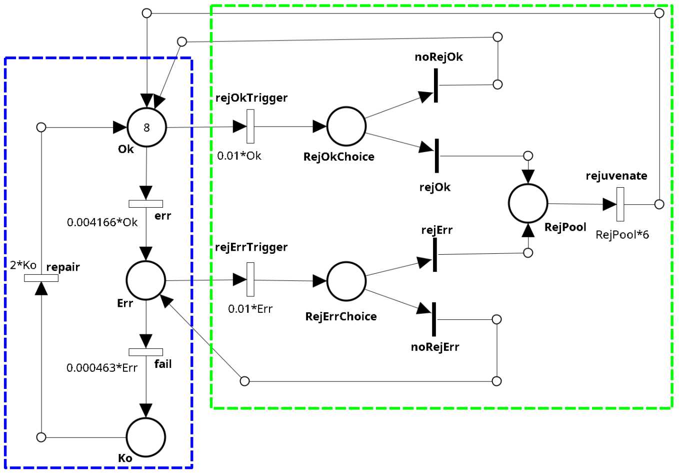


Fig. 2: Model of uncoordinated rejuvenation of a pool of service replicas.

A. Uncoordinated Replicas Rejuvenation

The most direct and natural way to implement inspection-based rejuvenation in practice is to implement an independent rejuvenation strategy for each replica. Each replica is thus periodically analyzed and rejuvenated if deemed necessary, without considering the overall state of the pool to which it belongs. We classify this type of rejuvenation strategy as uncoordinated rejuvenation.

Orchestration frameworks natively support system monitoring practices through probing. For instance, the Liveliness Probe[2] in Kubernetes enables periodic inspection of the internal state of a pod, which is equivalent to a service replica. Through criteria defined programmatically by the developer, the liveliness probe can automatically trigger the reboot of the pod. Thus, Kubernetes allows for the effortless implementation of inspection-based rejuvenation strategies and in particular offers a built-in mechanism for uncoordinated rejuvenation.

As in the well-known work of Garg et al. [36], we consider a two-step failure process to represent the aging of a single replica. Based on this, we extend the model in the form of an STPN, to describe the concurrent progress of multiple replicas in an uncoordinated rejuvenation configuration. The resulting model is represented in Figure 2. On the left side of the network (blue section), similar to Garg et al., we describe the system aging, failure, and repair process. Conversely, on the right side (green section), we model the mechanism of inspection-based rejuvenation. Each token in the left-hand

[2]https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/ Accessed July 27, 2024

section represents a service replica of the pool. Initially, all replicas are in a safe state, marked as `Ok`. In the specific representation of Figure 2 then, we are assuming a pool size equal to 8. As ARBs accumulate over time, replicas will progressively shift into a compromised state, denoted as `Err`. If the process persists, they will ultimately reach a failure state, indicated as `Ko`. After a failure occurs, the replica undergoes an unplanned repair process, which restores it back to a safe state.

Concurrently, each replica undergoes periodic inspection, regardless of whether it is in a safe state or in a deteriorated state. We represent the inspection from a safe space with the `RejOkChoice` place and conversely, we denote the inspection from a deteriorated state with the `RejErrChoice`. During the inspection, the replica could be classified as aged and subsequently rejuvenated, transitioning to the `RejPool` place, or it could be classified as safe and maintained available until the next inspection. This mechanism inherently allows for the potential misclassification of the aging state of a replica, leading to false positives (represented by the `rejOk` transition) and false negatives (represented by the `noRejErr` transition).

The values of the exponential distributions have been assigned so that the aging, failure, repair, and rejuvenation times of a single replica fit the expected values identified by Garg et al. [36]. For convenience, these values are reported in Table I. A peculiar feature of the scenario considered is that, unlike in the original model, each replica experiences an independent aging and monitoring process. Thus, the exponential distribution rate is here defined as proportional to the number of tokens (replicas) present in the place that acts as a precondition for the associated transition. In the STPN formalism, this is implemented by assigning to the transitions the rate assumed for the single replica multiplied by the name of the precondition place (e.g., `0.004166*Ok` for the `err` transition). Moreover, the original model by Garg et al., outlined a time-triggered strategy where the system is periodically rejuvenated. In contrast, this model uses `rejOkTrigger` and `rejErrTrigger` to define the frequency at which each replica is inspected.

Transition	**Expected Value** (hours)
`rejPool/rej`	0.1666
`err`	240
`fail`	2160
`repair`	0.5

TABLE I: Expected values of timers used in the MRGP under enabling restriction of [36]. Stochastic parameters of timers in Figure 2 and Figure 3 were selected to obtain the same expected values.

We assume that each inspection may misclassify the state of a replica with a certain sensitivity and specificity. In particular, the model proposed allows the characterization of specificity for each inspection through a probabilistic switch represented by the `RejErrChoice` place and the concurrent transitions `rejErr` and `noRejErr`. These transitions encode true positive and false negative classifications, respectively. Similarly, the sensitivity is represented by the probabilistic switch composed by the `RejOkChoice` place and the concurrent transitions `rejOk` and `noRejOk`. These other two transitions encode false positive and true negative classifications, respectively. For our model, we have assumed a probabilistic weight of 0.1 for the transitions `rejOk` and `noRejErr`, and a probabilistic weight of 0.9 for `rejErr` and `noRejOk` resulting in a sensitivity and specificity of 0.9 for each inspection.

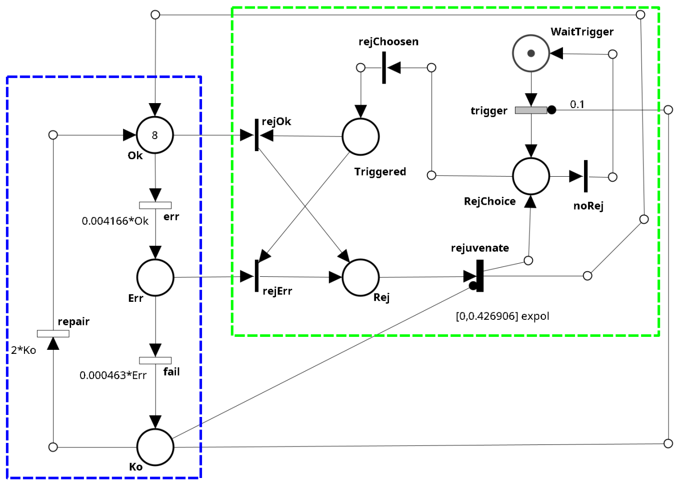


Fig. 3: Model of coordinated rejuvenation of a pool of service replicas.

B. *Coordinated Replicas Rejuvenation*

The probing and monitoring technologies offered by the orchestration frameworks mentioned in Section III-A could be used to implement more complex rejuvenation mechanisms. With uncoordinated rejuvenation (Section III-A), each replica is subject to independent inspection mechanisms and rejuvenation actions. However, this strategy could cause simultaneous rejuvenation of multiple replicas with consequent drops in system performance. In addition, as already mentioned, inspections aimed at identifying software aging in a replica are prone to classification errors.

We therefore propose a centralized inspection strategy that is aware of the overall state of the pool. Knowing the state of the other replicas allows for the implementation of mechanisms that prevent the phenomenon of simultaneous rejuvenation. Furthermore, it enables the comparison of aging metrics among the various replicas, suggesting a rejuvenation scheduling of the replicas and reducing the possibility of misclassification. In contrast with the uncoordinated policy outlined in Section III-A, we classify this type of rejuvenation strategy as coordinated rejuvenation.

Figure 3 shows the model for coordinated rejuvenation in the form of STPN. To represent the strategy of coordinated rejuvenation on a pool of replicas, we maintain the same representation of the aging, failure, and repair processes used for uncoordinated rejuvenation (the blue section), while we completely change the part for the rejuvenation mechanism (the green section).

Unlike uncoordinated rejuvenation, this strategy implies a periodic inspection performed pool-wise. This behavior is then represented in the model through the `WaitTrigger` place and the deterministic transition `trigger`. Periodically, with a period identified by `trigger`, the inspection is performed. The choice whether to start rejuvenation actions or not depends not only on the state of a single replica but rather on the state of all the active replicas of the pool. In practice, this consists of evaluating the aging state of each individual replica and consequently choosing whether to rejuvenate one or more replicas or postpone to the next inspection. We represent this choice with the probabilistic switch in the place `RejChoice` with the immediate transitions `rejChosen` and `noRej`.

When the rejuvenation action is chosen, the `rejChoice` transition fires, placing a token in the `Triggered` place. Similar to the uncoordinated policy, this model also consider possible misclassification of replicas. In this model, the sensitivity and specificity is represented by the probabilistic switch with the `rejOk` and `rejErr` transitions in combination with the probabilistic switch of `RejChoice`.

Once the replica to rejuvenate is extracted, the corresponding token is moved to the `Rej` place and the rejuvenation time is modeled by the rejuvenate transition. Upon completion of rejuvenation, the replica is reinserted into the pool in the `Ok` place. With coordinated rejuvenation however, it is not mandatory to wait for a new rejuvenation period to perform another rejuvenation. In fact, another token is placed in `RejChoice` where the decision on whether to perform an additional rejuvenation or not is repeated. This last behavior encodes the possibility of performing an indefinite number of rejuvenations at each rejuvenation period, albeit forced to be sequential. Therefore, it can be said that coordinated rejuvenation implicitly models periodical burst rejuvenations with a sequential schedule.

As in the previous model, the values of the distributions were chosen in order to fit the expected values of Table I. However, some noteworthy differences deserve to be mentioned. The decision on whether to initiate a rejuvenation or not should depend on how many replicas are in a safe or in an aging state. This is concretely expressed through the assignment of probabilistic weights on the `rejChosen` and `noRej` transitions. In particular, the following formula $\epsilon_1 + \lambda * \texttt{Err}$ is used for `rejChosen` and $\epsilon_2 + \texttt{Ok}$ for `noRej`.

With these weights, the greater the number of aged replicas (number of tokens in `Err`), the higher the probability with which the system will choose to perform rejuvenation actions. Furthermore, there is a residual possibility ϵ_1 of performing rejuvenation even without any replica in aging state, thus implementing the possibility of committing false positives for the inspection method. Similarly, ϵ_2, encodes the residual possibility that no rejuvenation is performed even when all the replicas are in aging state, thus committing a false negative classification. For the experiments, we have chosen to set $\epsilon_1 = \epsilon_2 = 0.3$.

One of the key features of coordinated rejuvenation is the ability to choose a replica to rejuvenate by comparing it with the other active replicas. This process is implemented through the definition of weights in the probabilistic switch formed by the `rejOk` and `rejErr` transitions. We identify with p the probability that in a single comparison between an aged replica and a healthy one, the wrong replica is erroneously chosen for rejuvenation. Then, the probability of making a mistake in identifying an aged replica through the comparison of all the replicas in the pool is equal to $k * p^m$, where k is the number of aged replicas and m is the number of healthy replicas. In fact, to select a healthy replica for rejuvenation the system should make m mistakes in the comparison with the healthy replicas for each of the k replicas in aging state. Therefore, the probabilistic weight of `rejOk` is equal to $\texttt{Err} * p^{\texttt{Ok}}$, while for `rejErr` it will be $1 - \texttt{Err} * p^{\texttt{Ok}}$. For the experiments, we have assumed $p = 0.1$.

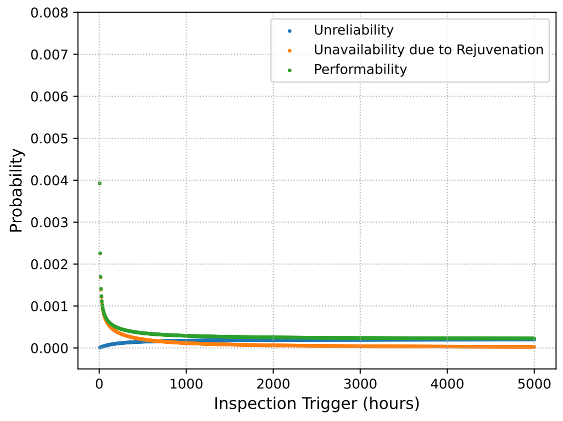
As a final remark, the `rejuvenate` timer is modeled with a truncated exponential distribution with density $f(x) = 4.28 \exp(-3.17x)$ over $[0, 0.43]$ keeping the fit with the expected value in the Table I.

IV. Analysis and Results

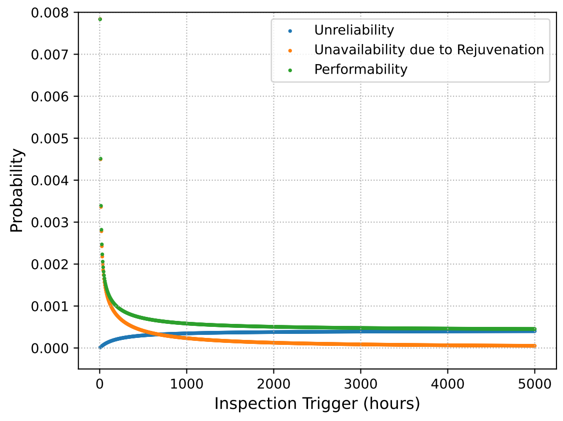
To evaluate the two strategies identified in Section III, we analyze the associated STPN models. In particular, we define three rewards, one for each metric of interest identified in Section II, *i)* unreliability, *ii)* unavailability due to rejuvenation, and *iii)* performability. Once identified the rewards, we evaluated how the inspection frequency and the number of replicas affect the metrics of interest at the steady state. The models were developed and validated using the Oris Tool. The execution of the experiments, in which the steady-state rewards were evaluated by varying the parameters of interest, was performed using the SIRIO Library.

Starting from the model of uncoordinated rejuvenation of Figure 2, we denote the unreliability with the reward `Ko` and the unavailability due to rejuvenation as `rejPool`. In terms of performability, we assume that the system fails to meet its QoS requirements when n or more replicas are offline. Thus, we define the performability metric as `Ko+RejPool≥n`. Regarding the coordinated rejuvenation strategy of Figure 3, equivalently to the scenario of uncoordinated rejuvenation, reliability is characterized by the reward `Ko`. Instead, the unavailability due to rejuvenation is denoted by the reward `Rej`, and as a result, the performability is expressed as `Ko+Rej≥n`. Note that although the rewards chosen to measure unreliability also imply unavailability due to non-rejuvenation causes, the main purpose of the rewards is to measure the replica failures occurring in the system.

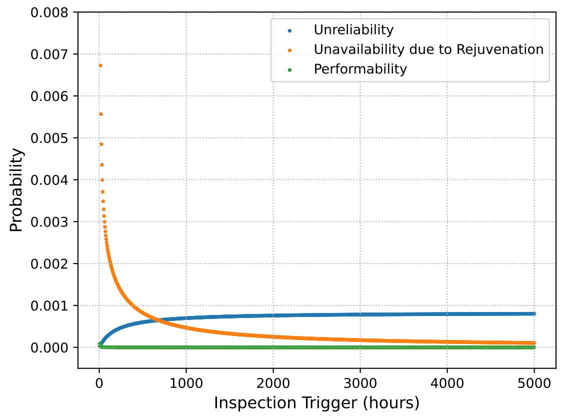
For our experiments, we evaluated the rewards defined by varying the inspection period of the strategy and the pool size. Regarding the inspection period, we evaluate the behavior of the two strategies by ranging from 5 to 5000



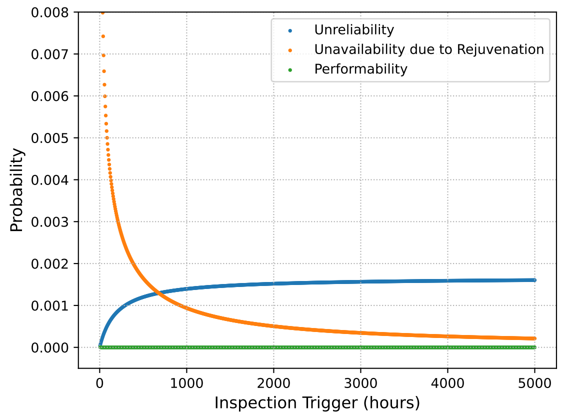
(a) *Pool size=1 and n=0*



(b) *Pool size=2 and n=1*



(c) *Pool size=4 and n=2*



(d) *Pool size=8 and n=3*

Fig. 4: Steady-state unreliability, unavailability due to rejuvenation and performability metric (`Ko+RejPool≥n`) as a function of the inspection period for the model in Figure 2 (uncoordinated rejuvenation).

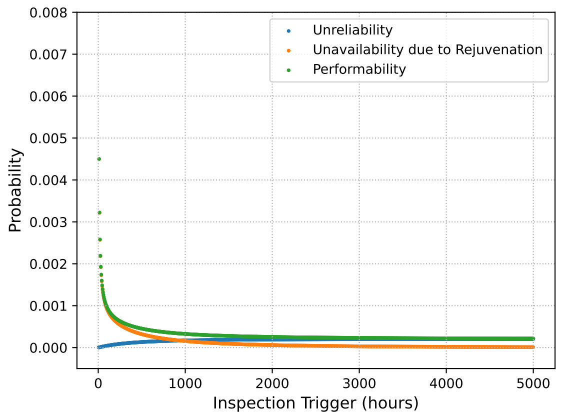
hours, incrementing by 5 hours. To ensure the comparability of the analysis results, for uncoordinated rejuvenation, we vary the average inspection time for each individual replica. For coordinated rejuvenation, instead, we vary the pool-wise inspection period i.e., the value of the deterministic transition `trigger`.

In terms of the pool size, we analyze the two strategies varying the number of replicas with the following values 8, 4, 2, 1. Note that the case with the single replica degenerates into a system without redundancy. For each of these pool sizes, we assume different performability threshold n. In particular, for a pool size of 8, we assume a maximum number of unavailable replicas n of 3; for a pool size of 4 we assume n=2; for a pool size of 2, n=1; and finally for a pool size of 1, we assume n=0.

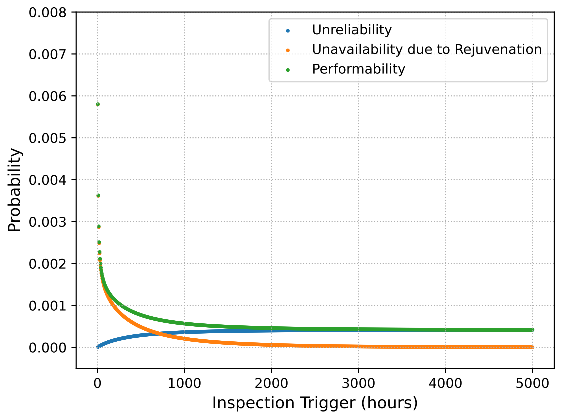
After analyzing all combinations of pool size and inspection period, we report the results, which are depicted in Figure 4 regarding the STPN model of uncoordinated rejuvenation, and in Figure 5 regarding the model of coordinated rejuvenation. Note that, the aim of this experimentation is to provide insights into the behavior of the system as the rejuvenation strategy and parameters vary. Nonetheless, it also assists system experts in consciously configuring the rejuvenation strategy, taking both reliability and performability into account.

The results show that certain patterns are common across both the rejuvenation strategies. High inspection frequencies (low values of inspection trigger) are associated with lower unreliability levels but also correspond to higher levels of unavailability due to rejuvenation. This, in fact, is consistent with the underlying logic of both rejuvenation strategies. Frequent inspections over time reduce the likelihood of the aging of a replica going undiagnosed before the failure, thereby enhancing resilience against false negatives. Conversely, this also increases the chance of erroneously identifying a healthy replica as aged, thereby simultaneously increasing the propensity for false positives.

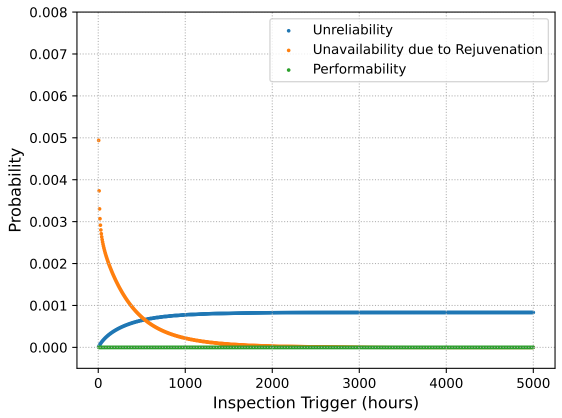
Maintaining the same inspection period, a larger pool size results in higher values of unreliability and unavailability due



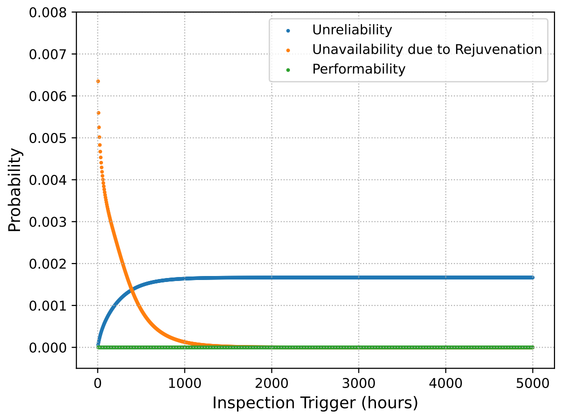
(a) *Pool size=1 and n=0*



(b) *Pool size=2 and n=1*



(c) *Pool size=4 and n=2*



(d) *Pool size=8 and n=3*

Fig. 5: Steady-state unreliability, unavailability due to rejuvenation and performability metric (`Ko+RejPool≥n`) as a function of the inspection period for the model in Figure 3 (coordinated rejuvenation).

to rejuvenation (except for the single replica case). This result is not surprising as the more replicas there are, the greater the chance that a replica will need to be rejuvenated or fail. On the contrary, the performability metric remains very low for pool sizes above 2, where we consider the maximum acceptable number of offline replicas to be n=2 for a pool size of 4, and n=3 for a pool size of 8. In light of this, the results suggest that as the pool size increases, it is advisable to increase the frequency of inspection. This will lead to an increase in the unavailability due to rejuvenation, but it will preserve the required QoS.

Regarding the differences between uncoordinated and coordinated rejuvenation strategies, with equal pool size and inspection period, the coordinated strategy generally offers better performance, which becomes even more pronounced as the pool size increases. These results are due to the intrinsic mechanisms of the coordinated rejuvenation strategy. Firstly, a replica selection policy for rejuvenation based on comparison with other replicas improves the detection mechanism. In fact, the more replicas are in an aged state, the more likely it is that one of these replicas will be chosen, thus reducing false positives and false negatives. Furthermore, the coordinated rejuvenation model prevents, by design, the simultaneous rejuvenation of multiple replicas, contributing to the performance of the system.

V. Conclusions

We addressed the problem of software aging in a pool of service replicas by characterizing two realistic inspection-based software rejuvenation strategies, named uncoordinated and coordinated rejuvenation respectively. Using Oris Tool and Sirio Library, we outlined a process of software aging in the pool of service replicas from which we defined two STPN models capable of representing the operation of both rejuvenation strategies. On top of this, we outlined how the unavailability of a single replica does not imply the complete unavailability of the service provided by the pool but rather underlies a performability problem. We then defined three metrics to evaluate the models: performability, reliability, and unavailability due to rejuvenation. Exploiting the Sirio Library, we conducted a quantitative evaluation of the models to study how the system behaves at steady state as the inspection period and pool size vary. The obtained results are consistent with the mechanism of software aging and rejuvenation outlined and offer interesting insights into the operation of both strategies, also highlighting better performance for the coordinated rejuvenation strategy.

Validation of the proposed models is currently under development. Once validated, this work will pave the way for other possible extensions. For instance, it could be interesting to study how these strategies behave with more unstable systems than the one presented by Garg et al. [36], with significantly shorter aging and failure times. Alternative rejuvenation policies could be explored, such as allowing the parallel rejuvenation of a specified maximum number of replicas.

References

[1] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications*, vol. 60, pp. 54–67, 2016.

[2] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, "Reliability and high availability in cloud computing environments: a reference roadmap," *Human-centric Computing and Information Sciences*, vol. 8, pp. 1–31, 2018.

[3] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th international conference on software quality, reliability and security (QRS)*. IEEE, 2019, pp. 176–185.

[4] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," in *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 2010, pp. 32–39.

[5] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)*. Ieee, 2008, pp. 1–6.

[6] J. Araujo, R. Matos, V. Alves, P. Maciel, F. V. d. Souza, R. M. Jr, and K. S. Trivedi, "Software aging in the eucalyptus cloud computing infrastructure: characterization and rejuvenation," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, pp. 1–22, 2014.

[7] T. Dohi, K. S. Trivedi, and A. Avritzer, *Handbook of software aging and rejuvenation: fundamentals, methods, applications, and future directions*. World scientific, 2020.

[8] L. Vinícius, L. Rodrigues, M. Torquato, and F. A. Silva, "Docker platform aging: a systematic performance evaluation and prediction of resource consumption," *The Journal of Supercomputing*, vol. 78, no. 10, pp. 12 898–12 928, 2022.

[9] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A study on the aging and fault tolerance of microservices in kubernetes," *IEEE Access*, vol. 10, pp. 132 786–132 799, 2022.

[10] R. Pietrantuono and S. Russo, "A survey on software aging and rejuvenation in the cloud," *Software Quality Journal*, vol. 28, no. 1, pp. 7–38, 2020.

[11] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and analysis of software rejuvenation in a server virtualized system with live vm migration," *Performance Evaluation*, vol. 70, no. 3, pp. 212–230, 2013.

[12] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[13] M. Torquato, P. Maciel, J. Araujo, and I. Umesh, "An approach to investigate aging symptoms and rejuvenation effectiveness on software systems," in *2017 12th iberian conference on Information systems and technologies (CISTI)*. IEEE, 2017, pp. 1–6.

[14] K. Kourai and S. Chiba, "Fast software rejuvenation of virtual machine monitors," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 839–851, 2010.

[15] J. Costa, R. Matos, J. Araujo, J. Li, E. Choi, T. A. Nguyen, J.-W. Lee, and D. Min, "Software aging effects on kubernetes in container orchestration systems for digital twin cloud infrastructures of urban air mobility," *Drones*, vol. 7, no. 1, p. 35, 2023.

[16] M. Torquato and M. Vieira, "An experimental study of software aging and rejuvenation in dockerd," in *2019 15th European Dependable Computing Conference (EDCC)*. IEEE, 2019, pp. 1–6.

[17] E. Andrade, R. Pietrantuono, F. Machida, and D. Cotroneo, "A comparative analysis of software aging in image classifiers on cloud and edge," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 563–573, 2021.

[18] D. Bruneo, S. Distefano, F. Longo, A. Puliafito, and M. Scarpa, "Workload-based software rejuvenation in cloud systems," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1072–1085, 2013.

[19] J. Liu, J. Zhou, and R. Buyya, "Software rejuvenation based fault tolerance scheme for cloud applications," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 1115–1118.

[20] L. Cui, B. Li, J. Li, J. Hardy, and L. Liu, "Software aging in virtualized environments: detection and prediction," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 718–719.

[21] J. Alonso, Í. Goiri, J. Guitart, R. Gavalda, and J. Torres, "Optimal resource allocation in a virtualized software aging platform with software rejuvenation," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 250–259.

[22] S. K. Mondal, J. K. Muppala, F. Machida, and K. S. Trivedi, "Computing defects per million in cloud caused by virtual machine failures with replication," in *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2014, pp. 161–168.

[23] J. Bai, X. Chang, G. Ning, Z. Zhang, and K. S. Trivedi, "Service availability analysis in a virtualized system: A markov regenerative model approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 2118–2130, 2020.

[24] L. Carnevali, M. Paolieri, R. Reali, L. Scommegna, and E. Vicario, "A markov regenerative model of software rejuvenation beyond the enabling restriction," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2022, pp. 138–145.

[25] L. M. Silva, J. Alonso, and J. Torres, "Using virtualization to improve software rejuvenation," *IEEE Transactions on Computers*, vol. 58, no. 11, pp. 1525–1538, 2009.

[26] M. Torquato, P. Maciel, and M. Vieira, "Model-based performability and dependability evaluation of a system with vm migration as rejuvenation in the presence of bursty workloads," *Journal of Network and Systems Management*, vol. 30, no. 1, p. 3, 2022.

[27] I. Polinsky, K. Martin, W. Enck, and M. K. Reiter, "nm-variant systems: Adversarial-resistant software rejuvenation for cloud-based web applications," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 235–246.

[28] R. Tirtea and G. Deconinck, "Software rejuvenation and replicated rejuvenated services," in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 2. IEEE, 2007, pp. 767–772.

[29] T. Thein, S.-D. Chi, and J. S. Park, "Availability modeling and analysis on virtualized clustering with rejuvenation," *International Journal of Computer Science and Network Security*, vol. 8, no. 9, pp. 72–80, 2008.

[30] D. Wang, W. Xie, and K. S. Trivedi, "Performability analysis of clustered systems with rejuvenation under varying workload," *Performance Evaluation*, vol. 64, no. 3, pp. 247–265, 2007.

[31] M. Paolieri, M. Biagi, L. Carnevali, and E. Vicario, "The oris tool: quantitative evaluation of non-markovian systems," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1211–1225, 2021.

[32] V. G. Kulkarni, *Modeling and analysis of stochastic systems*. Chapman and Hall/CRC, 2016.

[33] A. Horváth, M. Paolieri, L. Ridi, and E. Vicario, "Transient analysis of non-markovian models using stochastic state classes," *Performance Evaluation*, vol. 69, no. 7-8, pp. 315–335, 2012.

[34] Sirio, "The Sirio Library for the Analysis of Stochastic Time Petri Nets." [Online]. Available: https://github.com/oris-tool/sirio

[35] E. J. Ghomi, A. M. Rahmani, and N. N. Qader, "Load-balancing algorithms in cloud computing: A survey," *Journal of Network and Computer Applications*, vol. 88, pp. 50–71, 2017.

[36] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of software rejuvenation using markov regenerative stochastic Petri net," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 1995, pp. 180–187.