Quantifying Failure Risk of Version Switch for Rolling Upgrade on Clouds

Daniel Sun*†, Len Bass*†, Alan Fekete*‡, Vincent Gramoli*‡, An Binh Tran*, Sherry Xu*, and Liming Zhu*†‡

Software System Research Group, NICTA, Australia
†The University of New South Wales, Australia
‡University of Sydney, Australia
Email: {firstname.lastname}@nicta.com.au

Abstract—Rolling upgrade is an industry technique for online dynamic software update. A rolling upgrade updates a small number of instances in an old version to a new version at a time and the operation is repeated in a wave rolling until all of the instances have been upgraded. In many cases, the software needs to avoid interactions between different versions. One common simple approach is to make instances version aware, and then a version switch point can be chosen to deactivate the old service and activate the new service. On a Cloud platform, upgrades can be implemented simply through replacing old virtual machine instances with ones in new versions, and during the process of rolling upgrade various failures may present. If an instance fails, a new instance has to be launched from the backup images, which in most software systems are in an old version and cannot be simply replaced to a new version if the new software and the new service have not been stable for the sake of reliability and stability. Thus the progress of the rolling upgrade is not guaranteed, and indeed the number of upgraded instances can sometimes decrease. We aim to determine the probability that, after switching the versions at a selected point, the number of working instances may sometime fall below the amount needed for a desired Quality of Service. In this paper, we stochastically quantify the risk with a family of discrete Markov chains (DTMC). The evaluation in both Amazon Web Service (AWS) and simulation reveals that our technique can well predict the risks after given version switch points.

I. INTRODUCTION

Software upgrades in enterprise systems, and especially in large scale web service systems, are inevitable and are performed with increasing frequency in practices such as Continuous Integration. For example, Etsy.com had 4004 releases into the production environment in 6 months [1]. Facebook integrates software modifications a few times per day [2]. The statistical data in [3], [4] expose the vulnerability of software upgrades. Once an upgrade fails, the aftermath may be serious, as in the losses of NSA [5]. The problems with online dynamic software updates have been known for some time [6]. Currently, an important practice in industry is to use rolling upgrade [4]. Facilities to support rolling upgrade have been provided in many software systems and products [7]–[9]. A rolling upgrade removes a small number of instances of an old version from service at a time and then replaces them with the instances of a new version. This process is repeated in a wave rolling through the system, until all the instances of the old version have been replaced. Throughout the paper, we denote the old version with $V_1$ and the new version $V_2$.

Both previous studies and industrial experiences have revealed that rolling upgrade is vulnerable to various failures. In large scale systems and applications, a lot of instances may be involved and thus a rolling upgrade can take hours or much longer. During the process of a rolling upgrade, there are many threats such as system hardware failures, software infrastructure faults, and also errors and faults in the virtualisation layers in a Cloud environment. In a real datacenter, failure recovery, system maintenance, and data storage may take hours or days [10]. All of these potential failures can significantly affect the upgrade procedure. Thanks to virtualisation technology, physical failures on and below virtualisation layer in Cloud computing can be efficiently tolerated by replacing and migrating failed virtual machines. AutoScaling which primarily aims at elasticity has been a standard module in popular Cloud software stacks like AWS [11] and OpenStack [12]. AutoScaling also keeps monitoring virtual machine instances and replaces failed instances with new ones which may be allocated in different physical machines.

Some applications are built in the ways where there are complicated dependencies such as dynamic linking, location or address, database schema, configurations, and replications and so on, that can be broken by the upgrade. One important example of dependency is the mixed version race [13], in which asynchronous message exchanges potentially lead to a callback from $V_2$ being processed by $V_1$. To avoid the mixed version race, one can make instances version-aware and then switch the versions: that is, before the switch, only instances running $V_1$ are active and provide the service, while after the switch, only the upgraded instances running $V_2$ are allowed to participate in the service. In this way, only one version provides the service at any time point.

On Clouds, the implementation of rolling upgrade and its fault tolerance become simpler and more flexible than on traditional platforms. However, the price we have to pay is that the dynamics of the overall rolling upgrade process can be poorly understood with the presence of failures. An instance can fail, from hardware or software issues and in either new or old version. The failed instance will typically be replaced by another instance. For a software system running on Clouds, a bundle of backup images are used to recover problematic instances. A software version from the backup images must be stable and reliable, and consequently the backup images remain with the old version until the later version has been confirmed online for a substantial period. Hence, during rolling upgrade any loss of instances will be handled by creating the instances in the old stable version rather than the new version. Therefore the progression of the upgrade may not always be
uniform, with the number of instances running the new version changing in complicated ways as failures occur. Thus there is a risk that, after the version switch, the number of upgraded instances may fall below the threshold needed for QoS.

In this paper, we propose to quantify the risk, based on a stochastic model which can represent the progress of synchronous rolling upgrade. With our model, we can quantify the risk associated with switching to $V_2$ at various steps during a rolling upgrade; this enables a principle method of choosing when to make such a switch. Empirical studies have been performed in Amazon Web Service (AWS). The experimental results prove the effectiveness of our risk control for the version switch. Large scale experiments on real platforms are involved with too much time and monetary cost. Hence, we also examine our work in simulation. The technique shown in this paper is actually effective to most Cloud platforms, although we discuss the issues related to Cloud platforms with a bias to AWS because of the easy access to the matured resources.

The rest of this paper is structured as follows. In Section II, we review related work. In Section III, we discuss fault tolerant rolling upgrade on Clouds, version switch and its risk for better understanding the background of this work. In Section IV, we introduce our modelling and quantifying the risk. In Section V, our experiments are described, and the results are presented. Section VI concludes this paper and justifies our contributions.

II. RELATED WORK

Rolling upgrade has been recognised as an important industry practice for high availability and equipped many production systems [7], [8], but rolling upgrade cannot be performed automatically. In Clouds, instance upgrades (as part of a rolling upgrade process) can be carried out by simply shutting down and restarting virtual machines [7], [9]. Version awareness, forward and backward compatibility are all common techniques used to manage multiple simultaneous versions. In [4], knowledge of faults is assumed and a fault model is proposed to help a system called Imago deal with faults. Through experiments with fault injection, Imago is effective in improving dependability of the system-under-upgrade. In [13], a probabilistic risk model is proposed to understand the risk by comparing the impact of mixed version races and the known bugs. In this way, it is possible to decide whether to upgrade or not.

On most Cloud platforms [11], [12], AutoScaling has been implemented for scaling computing capacity up or down automatically according to the predefined conditions. With AutoScaling, one can ensure that the number of virtual machine instances increases seamlessly during demand spikes to maintain performance, and decreases automatically during demand lulls to minimise costs. In AWS, AutoScaling Group (ASG) also evaluates the health of each Amazon EC2 instance and automatically replaces unhealthy Amazon EC2 instances to keep the fixed size of ASG. Thus, ASG actually provides a certain of fault tolerance to AWS on platform level.

There are also statistics, machine learning and rule-based approaches for diagnosing errors during normal operations [14], [16], which assume a system has normal operational profiles that can be learned from historical data. Deviations from the profiles can help detect, localise, and identify faults. In [3], it has been argued that some problems with software upgrades are caused by a poor integration between deployment, testing, and problem reporting. This argument results in a framework integrating testing and problem reporting into deployment cycle of software upgrades. It has been shown that this approach is effective for real upgrade problems.

Reliability and availability of software and hardware systems have been well analysed with stochastic models [17]–[20]. [18] addresses the relationship between software reliability and the testing resources allocation on DTMC, which aims to quantitatively identify the most critical components of software architecture. Towards more practical models, the techniques of information extraction from software systems have been well surveyed [18].

III. BACKGROUND

With a concern of fault tolerance, this section introduces the issues of fault tolerant rolling upgrade on Clouds and then discusses the version awareness, the version switch, and the risk associated with them.

A. Fault Tolerant Rolling Upgrade on Clouds

To achieve a fault tolerant rolling upgrade on Clouds, it is inevitable to deal with failures: failures on platform level, failures caused by rolling upgrade operations, and failures due to the mixed version race. As we have introduced in Section I, AutoScaling is able to provide the basic fault tolerance to platform and operation failures. Whenever one or some instances are unhealthy, AutoScaling can replace unhealthy instances with new healthy ones. The possibility of all instances being failed and completely unrecoverable is very small on a matured platform and we do not take into account this extremely rare case.

1) Detection of Failures: If there is any failure or error with which virtual machine instances can work normally but the software system is out of service, AutoScaling is not responsible to this type of errors since platform level fault tolerance is only effective to virtual machines but not the application software inside. Only if application level failures and errors lead to failed virtual machine instances, AutoScaling takes the duty. The upgrade or installation operations are often vulnerable as shown in Section II. Detection tools and systems are available to deal with application level failures and errors, and therefore in this paper we assume such a detector is running. When a failure is detected, we simply kill this instance and then AutoScaling will launch a new one in $V_1$, because the replacement is efficient and cheap on Clouds.

2) Synchronous Operation: Rolling upgrade takes a small number, named the granularity $G$, of instances out of service at a time and then $V_2$ is installed into those instances. After the installation is finished, the newly and previously installed instances need to be tested. The test can use the received requests to the online service in $V_1$ and examine if correct responses can be made by $V_2$. During this test, there should not be any change to the new software environment in $V_2$, because the attendance of a new instance may change settings and configurations of software environment. To avoid unexpected tests and operations, the $G$ upgrading operations, i.e., the $G$
installations of $V_2$, are synchronous and start at the same time. After the installations are finished, the new instances are tested also at the same time. Thus, the fault tolerant rolling upgrade becomes periodic. Each period is a stage in the whole process and one can choose the period time to be reasonably longer than the sum of installation time and test time. If there is an upgrade longer than the predefined period, the instance can be replaced too.

B. Version-Aware, Version-Switch, and Risk

Although techniques have been studied for solving the mixed version race problem, a simple and reliable method is to install $V_2$ by rolling upgrade to each instance and meanwhile to test the functionality and the correctness of the new software environment. In this way, the mixed version race problem can be fundamentally avoided. During a rolling upgrade, the service is still provided by $V_1$ and the software system $V_2$ may accept the requests to the service but do not respond. This is called version-aware, i.e., the service being aware of which version is in function. $V_1$ can be switched to $V_2$ after all instances have been installed $V_2$. However, switching after all are ready is conservative. In most systems redundancy and replication are adopted for better availability and reliability. Thus, there must be a minimum number of instances for a certain quality of service. Let such a number be $R$. Then, actually it is possible to do version-switch just after $R$ instances are ready. This is meaningful for large scale systems with considerable redundancy since in this way $V_2$ can be brought into online service much earlier.

For the fault tolerant rolling upgrade and the version switch introduced above, let us consider such a system: It consists of a set of instances that initially are each ready to be upgraded from $V_1$ to $V_2$. The total number of instances is $N$. Each time $G$ instances are being upgraded. Failures can be detected and then unhealthy instances are replaced. The version awareness and switch are adopted to fully avoid mixed version race problem. To provide satisfying QoS, there must be at least $R$ working instances. After the number of upgraded instances has reached $R$, the versions can be switched, and if we switch just at $R$ upgraded instances there will be only $R$ working instances right after the switch. The number of working instances will gradually increase to $N$ as the rolling upgrade proceeds. The risk is associated with the decision of switch-point if early version switch is required, since if we switch at $R$ and some failures happen, the number of working instances is possibly lower than $R$. This problem can be formally defined: Given $R$ in $N$ instances, what is the risk or how to quantify the risk when there are $R, R+1, R+2, \cdots$ or $N$ upgraded instances if we switch the versions from $V_1$ to $V_2$ at $R, R+1, R+2, \cdots$ or $N$? As shown in the following sections, this question can be answered well after we derive an accurate model for the fault tolerant rolling upgrade.

IV. QUANTIFYING THE RISKS WITH DTMC MODEL

A. The Creation of the Model

1) Characterisation and the DTMCs: With regard to fault tolerance as mentioned in the above, upgrade operations are synchronous and hence a software system is upgraded in discrete time. After a number of discrete hops, a rolling upgrade is finished. Therefore, a discrete model is suitable to represent fault tolerant rolling upgrade and for this purpose DTMC is adopted. A process of rolling upgrade is characterised by the number of upgraded instances. $N$ instances imply $N$ states during a process. The state $i$ implies that there are exactly $i$ healthy instances in $V_2$. The state 0 represents that all instances are in $V_1$ and while the state $N$ represents that all instances are in $V_2$. Hence, there are $N + 1$ states in the system, where the initial state is always 0. When the upgrade process reaches the state $N$, the rolling upgrade is successfully finished. For the synchronous operations, each period is also called a stage, which moves the system from one state to another.

We classify the failures into two categories: One category contains the failures which present in each instance with equal possibility and hence the failures in this category are called platform failures; the other contains the failures which only happen in the instances being upgraded and consequently we call the failures in the other category operation failures. To deal with the failures, we only need to replace the unhealthy instances.

Let $K$ denote the maximum number of failures during one stage. In each stage there are $0 \leq k \leq K$ platform failures need to be counted on. $P_f(k)$ is the probability mass function. The following discussion is based on $K \leq N$. As for $K \geq N$, we will discuss at the end of this section with the retrieve of $P_f(k)$ through software test and measurement. For the operation failures, $P_{ops}$ denotes the possibility of a successful upgrade operation. For the easy discussion in the following sections, a list of notations and symbols are laid out in Table I.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of instances/vms</td>
</tr>
<tr>
<td>$G$</td>
<td>Granularity, the number of instances in one stage</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of non-upgraded instances</td>
</tr>
<tr>
<td>$w$</td>
<td>Number of failures in upgrading instances</td>
</tr>
<tr>
<td>$u$</td>
<td>Number of failures in upgraded instances</td>
</tr>
<tr>
<td>$z$</td>
<td>Number of failures in non-upgraded instances</td>
</tr>
<tr>
<td>$\nu$</td>
<td>Number of operation failures</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of platform failures in one stage</td>
</tr>
<tr>
<td>$K$</td>
<td>Maximum number of platform failures</td>
</tr>
<tr>
<td>$R$</td>
<td>Minimum number of instances</td>
</tr>
<tr>
<td>$V_1$</td>
<td>The old version</td>
</tr>
<tr>
<td>$V_2$</td>
<td>The new version</td>
</tr>
<tr>
<td>$P_{ops}$</td>
<td>Probability of a single successful upgrade operation</td>
</tr>
<tr>
<td>$P_f(k)$</td>
<td>pmf of platform failures</td>
</tr>
<tr>
<td>$P_{ops}(X,x)$</td>
<td>pmf of $x$ successful operations out of $X$ upgrade operations</td>
</tr>
<tr>
<td>$P_{con}(w,u,k,i)$</td>
<td>pmf of platform failure configuration</td>
</tr>
</tbody>
</table>

We regard the states as ordered by $i$, from the initial state 0 to the final state $N$, when a rolling upgrade has been completed; thus the state $N$ is an absorbing state. When a stage produces more upgraded instances than there were before, we consider this a forward transition in the DTMC; conversely, the case where there are fewer upgraded instances after a stage is called a backward transition. It is also possible for the state after the stage to stay in the same.

The values of parameters $G$ and $K$ place limits on how far the state can move in a single stage, that is, in one transition. As an upgrade operation is attempted on only $G$ instances in a stage, the state after the stage is at most $G$ greater than the
state before, and because of $K$ any backward transition drops at most $K$ states. With different $G$ and $K$, the DTMC may have different appearances, and actually there is a family of discrete time Markov chains. We name them to be $G-K$ chains. Some examples are shown in Fig.[I] Given specific $G$ and $K$, and given transition probabilities, a DTMC is defined.

![Fig. 1. Examples of chains. 1-1, 2-1, 1-2, 2-2 chains from top to bottom. The transition probabilities are omitted.](image)

2) General Solution to The Model: In this paper, we deal with the general solution with arbitrary $G$ and $K$, i.e. $N \geq G \geq 1$ and $N \geq K \geq 1$. We need to calculate the transition probabilities in terms of the underlying probabilities of a platform failure and an operational failure, and then we solve the dynamics of the DTMC. Given $P_{\text{ops}}$, the probability of success in a single upgrade of an instance, then we derive the probability of $x$ successful operations out of $X$ attempted upgrade operations as a Binomial distribution, which is

$$P_{\text{ops}}(X, x) = \binom{X}{x} C_x \cdot P_{\text{ops}}^x \cdot (1 - P_{\text{ops}})^{X-x}. \quad (1)$$

We also need to determine the probabilities for a given configuration of platform failures, that is, the number of failures that affect instances running $V_i$, the number of failures in the instances being upgraded, and the number of failures that impact instances upgraded already to $V_2$. The distribution $P_j(k)$ that gives the probability that $k$ platform failures occur during a stage can be obtained by repeatedly testing and simulation of a software system as shown in [13]. Suppose $k$ instances fail, and we use $u$ to denote the number of failures in already upgraded instances, while $w$ denotes the number of failures in upgraded instances, and $z$ denotes the failure number of non-upgraded instances; thus $w + u + z = k$. For each configuration of specific $w$, $u$, and $z = k - w - u$, we need to know its probability. Note that a failure happens to each instance with equal probability. Let $P_{\text{conf}}(w, u, k, i)$ be the probability of a configuration of valued $w$, $u$ and $z$ under $k$. When $k$ failures happen in $N$ instances, each configuration is a partial permutation because an instance can only be lost once. The total number of the permutations should be $N_p$.

For a configuration $w$, $u$, $z$, it is necessary to select $w$ from $G$ instances that are being upgraded in this stage, $u$ from $i$ upgraded instances, and $z$ from $n = N - G - i$, and then we do permutations on $k = u + w + z$. Hence, the number of permutations for a given configuration must be $G^w \cdot i^u \cdot n^z \cdot k!$. Thus, the probability is

$$P_{\text{conf}}(w, u, k, i) = \frac{G^w \cdot i^u \cdot n^z \cdot k!}{N_p} = \frac{G^w \cdot i^u \cdot (N - G - i)^z}{N^z \cdot k}. \quad (2)$$

We can now write the transition probability for each possible move in the DTMC, based on determining the possible numbers of failures and successful upgrades that occur during a stage. For a forward transition from the state $i$ to $i + j$, the minimum successful upgrades should be $j$. The number of successful upgrades can be greater than $j$, when some upgraded instances are lost due to failures. Since in every stage, the rolling upgrade operates on $G$ instances, it is necessary to understand what have happened to the other $G - j$ instances. If $w$ upgrading instances fail, the effective upgrades are actually $G - w$, but $G - w \geq j$ because at least $j$ successful upgrades must be performed for the transition from $i$ to $i + j$. Hence, we know $w \leq G - j$. Given the number of failures $k$ in one stage, $w \leq k$ too. Thus, we have $w \leq \min(G - j, k)$. Then let $u + v = G - j - w$, there must be $u$ failures in upgraded instances and $v$ operation failures, otherwise the state will move further than $j$ steps. In other words, the number of effective upgrades should be $u + v = G - w$, among which the number of successful upgrades should be $u + j$ since $v$ upgrades must fail. Here, $u$ should be bounded by the number of platform failures $k$, i.e., $u \leq k - w$, $u \leq i$ and meanwhile $u \leq G - j - w$. Therefore, $u \leq \min(G - j - w, k - w, i)$. Thus, given $k$, the ranges for $w$ and $u$ are known. By summing up, the forward transition probabilities is

$$P_{k,i+j} = \sum_{k=0}^{K} \sum_{w=0}^{\min(G-j,k)} \sum_{u=0}^{\min(G-j-w,k-w,i)} \alpha, \quad (3)$$

where $\alpha = P_{\text{ops}}(G - w, u + j)P_j(k)P_{\text{conf}}(w, u, k, i)$.

Similarly for the backward transition probability, $w \leq \min(G, k)$ and the net loss of upgraded instances should be $j$. Hence, the minimum number of failures should be $j$ since only failures can move the states backward. The number of successful upgrades is $G - w - v$ and hence $j = u - (G - w - v)$. Consequently among $G - w$ effective upgrades, there are $u - j$ successful upgrades. Here $u$ is bounded by $j \leq u \leq \min(G - w + j, k - w, i)$. The backward transition probability is

$$P_{k,i-j} = \sum_{k=j}^{K} \sum_{w=0}^{\min(G-k-j)} \sum_{u=0}^{\min(G+j-w,k-w,i)} \beta, \quad (4)$$

where $\beta = P_{\text{ops}}(G - w, u - j)P_j(k)P_{\text{conf}}(w, u, k, i)$. In the similar way, we order $\gamma = P_{\text{ops}}(G - w, u)P_j(k)P_{\text{conf}}(w, u, k, i)$ in the probability of loop transition to the same state:

$$P_{k,i} = \sum_{k=0}^{K} \sum_{w=0}^{\min(G-k)} \sum_{u=0}^{\min(G-w,k-w,i)} \gamma. \quad (5)$$

The absorbing state $N$ is always associated with a loop transition probability 1.

With all the transition probabilities, the stochastic matrix $P$ can be established and the sum of each row is 1. The matrix
P can be partitioned. I is an identity matrix and here its only one entry is 1.

\[ P = \begin{bmatrix} Q & H \\ 0 & I \end{bmatrix}. \]  

(6)

Then the fundamental matrix \( M \) is

\[ M = \sum_{j=0}^{\infty} Q^j. \]  

(7)

Several techniques can be used to obtain \( M \) with each entry \( m_{i,j} \) and in this paper we adopt the one for absorbing Markov model in Trivedi [17].

B. Quantifying the Risks

For a system without any redundancy and replication, it is not so realistic to switch versions early in rolling upgrade with the presence of failures. Hence we only discuss that a system is designed with redundant instances, and there must be at least \( R \) instances available, where \( R < N \). If failures never occur, the version switch would be safe when the system state \( i \) reaches \( R \), that is, the minimum required number of instances have been upgraded. However, as failures are possible, it might be that after switching, the service capacity requirement is violated. Thus the version switch point \( s \) must be chosen somewhat greater than \( R \), with regards to the probability that the system will evolve and pass through a state with fewer than \( R \) upgraded instances.

For this analysis, we modify the DTMC by coalescing all the states below \( R \) into the other absorbing state which we call the death state \( D \). This death state is connected to the states \( R, R+1, R+2, \ldots, R+K-1 \). That is, the new absorbing state is connected to all the states from which a one step transition may produce fewer than \( R \) upgraded instances.

Consequently, the chain to represent the process after a fault tolerant rolling upgrade reaches \( R \) upgraded instances is as shown in Fig. 2. Since from \( R \) to \( R+K-1 \) the backward transitions will fall into the dead state, it is necessary to know the transition probabilities \( p_{R+i,D} \), \( 0 \leq i \leq K-1 \). From \( R+i \), there are \( K-i \) transitions to the death state. Thus, the sum of all those transition probabilities must be \( p_{R+i,D} \).

\[ p_{R+i,D} = \sum_{j=i}^{K-i} p_{R+i,R+i-(j+1)} \]  

(8)

With careful substitution of different \( i \) and \( j \), we can easily find \( p_{R+i,D} \). Let us consider the \( P \) matrix for this case. With appropriate linear transformation, the matrix can be in the form of Eq. 6. Consequently, \( Q \) is a \( N-R \) square matrix. Without losing generality, we can arrange the last row to represent the dead state. It is not difficult to locate \( Q \) and \( H \) matrices in \( P \). Also, we can derive the fundamental matrix \( M \) in the same way. The chain will be absorbed into either the final state or the death state. Let \( b_{i,N} \) and \( b_{i,D} \) denote the probabilities of being absorbed into \( N \) and \( D \) respectively. Then there exists a \( N - R \) by 2 matrix \( B \), in which the left column is the probabilities \( b_{i,N} \) and the right is \( b_{i,D} \).

\[ B = M \times H. \]

(9)

Thus, to select an early switch point \( s \) we find \( b_{s,D} \) which is small enough (equivalently, make \( b_{s,N} \) large enough). For example, if \( b_{s,N} \) is 0.99, after the version switch on \( s \), there is at most 0.01 probability that the rolling upgrade fails at some point to deliver the service requirement. The early version switch is a tradeoff between the risk of providing insufficient capacity and having a long waiting time before \( V_2 \) comes online.

C. Scope of Validity

In this paper, we do not assume optimistically the probability distribution of platform failures following a known one such like exponential distribution, although the model can be extended to adopt a theoretical distribution. The reason is that assuming such a theoretical distribution does not match real software systems in practice. Not like hardware systems, software failures are difficult to be modelled formally due to the complexity of software stack, human factors, and software bugs and flaws. Therefore, we require that through repeated testing and simulations, the possibility of platform failures can be acquired or estimated statistically. We assume for this paper that the distribution of platform failures is independent among instances; in particular, that an upgraded instance is just as likely to suffer a platform failure as any other instance. Similarly the probability of operation failures can also be obtained. Pietrantuono et al [18] addresses how to extract information for software reliability. In Section V the experiments on AWS comprehensively take into account the reported data in the literature and our tested data, and the other experiments are based on a simulation controlled by Binomial distribution. \( K > N \) implies that all instances fail, i.e. the crash of AutoScaling Group, a rare case in practice and addressing this problem is equivalent to adding a transition from each state to the death state.

This paper only presents such a stochastic technique to qualify failure risk. In the practical usage, one has to comprehensively decide acceptable risk levels or thresholds, which are impacted by many factors, such as service types, time, software and hardware conditions. In other words, the acceptable risk levels or thresholds should be decided with respect to the requirements of system reliability and availability. Technically, this is involved with single-objective or multiple-objective optimisation problems and hence is out of the scope of this paper.

V. Empirical Studies

We empirically evaluate the risk quantification for early version switch through some experiments on Amazon Web Service [14] and a simulation. The model is capable of dealing with a big number of instances. However, in a real Cloud platform, the time and monetary costs are too high for us to perform massive experiments, even if only micro and small
instances are used. Therefore, the numbers of instances and stages in a rolling upgrade have been limited to fit our research budget. In order to inspect the effect on large scale systems, we adopt a simulation. Thus, the experiments are two-fold: the evaluation on AWS, and the scaling up on a simulation. The impacts from the parameters and configurations can be observed clearly from the figures.

A. Evaluation on AWS

Each of the experiments in Fig. 3 has been repeated at least 20 times. The model has been coded in R [21] and accessible with the control script in [22]. In order to easily observe the system dynamics quickly, we inject failures at the rates which are determined in terms of the report in the literature and our own observations. In all these experiments, we do not allow the mean of the failure numbers to be greater than the granularities and actually we maintain a moderate failure rate, since if the mean of the failure numbers is too high the running time can be too long or even the experiments cannot stop.

As explained in Section IV-C the information can be extracted from software test, analysis, and the statistical frequency of detected faults. In this empirical study, we set the values in terms of our tests and those reported in the literature. In [4] it has been shown that during a single upgrade the failures due to system configuration errors may happen with a probability of 14.6% in a range of [0%, 38.0%], and the probability for failures due to data access is 18.7% with a range of [0%, 45.1%]. We also performed a series of tests on AWS that the probability of upgrade failure varies from 3% to 57% in terms of the type of virtual machine images and specific operations. Based on the above observations, we set the discrete probabilities of failures and also guarantee

\[ \sum_{k=0}^{K} P_f(k) = 1 \]

always. The mean of the failures \( \mu \) and the other parameters are shown in each figure.

The version switch and the risk control are tested in Fig. 3. In each experiment, we collect the data of the frequencies among all runs where the capacity falls below the threshold, for a particular switch point shown on the x-axis. We calculate the risk by using the DTMC model as in section IV-B. The smaller \( R \) implies that there are more replicated instances. It is reasonable that the probability to fail for \( R = 20 \) is also smaller than \( R = 25 \) since there are 10 replicated instances and only 5 for \( R = 25 \). Given \( R \), the model can accurately tell when it is suitable to make the version switch. Obviously in the case of \( R = 25 \) the early version switch is not a wise choice at all. In the case of \( R = 20 \), the versions can be switched after 25 instances have been upgraded if a risk of 5% is tolerable. A smaller \( R \) implies more replicated virtual machine instances and thus the early version switch can benefit more. In a larger system, the effect is more clearly visible.

We also show the aftermath of a version switch in Fig. 4 because the experiments in the above and the below are only for risks. It is interesting to see what will happen at a switch point. The version switch is observed by such an experiment: The clients send continuously HTTP requests to the instances running HTTP service and then we observe the change of performance after the version switch. In AWS, it is very easy to observe the CPU utilisation of a bunch of virtual machines via CloudWatch [11]. Thus, we use the CPU utilisation to reflect the service capacity in this experiment. We switch the versions after 20 instances have been upgraded and then the data on CPU utilisation are collected. The sample frequency is once per minute and we record the data from the beginning to the end of two processes of rolling upgrade. Two dramatic vibrations can be observed in each data series. The reason to the peaks is the version switch. When the versions are switched, the available number of instances in the new version becomes smaller, since the switch points are still far away from the end of the rolling upgrade (around 40 minutes ahead of the end of each rolling upgrade). Because the version switch reduces the service capacity and the load balancer only dispatches the requests to the upgraded instances, the aggregated CPU utilisation increases on the switch point. If the workload is heavy, the CPU utilisation will be capped at 100%. Hence, we only use a light work load for better observation here. After the switch, the service capacity decreases a few and this raises the CPU utilisation but do not endure the peak of utilisation too long. When more upgraded instances come into the service, the CPU utilisation returns to its normal level. If the work load is heavier or the switch points are badly chosen, the service capacity may not be sufficient for the incoming requests and thus risks came up.

B. Scaling-up in Simulation

Each of the experiments in Fig. 5 has been repeated 1000 times in the simulation. Injecting failures makes non-sense for a simulation and consequently we consider such a case: each instance may fail in a fixed rate, which is available or can be estimated for most platforms. The fault tolerant rolling upgrade behaves in a periodic way. Hence, it is not difficult to compute the failure probability of an instance during a period. As a result, we control the presence of failures by Binomial distribution with the failure probability \( p \) and the random number of platform failures can be computed from the distribution for a period. Like in the above section, we do not allow \( p \) to be too big for the same reason as choosing moderate \( \mu \). For the DTMC model, we generate a probability
mass function in terms of a Binomial distribution under $p$ and guarantee the mean number of failures is the same as the Binomial Distribution. These experiments show the impact of both $R$ and the failure rate $p$. A small change in $p$ results in very different risks for the version switch. In both $R = 216$ and $R = 200$, a higher $p = 0.008$ leverages the risks up to an unacceptable level. Even with 248 upgraded instances already, the numbers of healthy and upgraded instances may drop below 200 and 216 respectively. In this case, with a high failure rate the early version switch is unrealistic. On the contrary, a smaller $p$ can totally produce different risks. For $R = 200$ with $p = 0.007$, it is very possible to switch the versions after 216 instances have been upgraded. For $R = 200$ with $p = 0.007$, as we have observed on AWS, a greater $R$ implies less possibility to switch safely.

From the above figures, we can observe the impacts from $R$, given $N$, $p$ is actually equivalent to $\mu$ given a fixed period of synchronous rolling upgrade. With a greater $G$ or $P_{ops}$, the rolling upgrade can proceed faster and thus given the same $R$ and $p$ the risks are lower as shown in Fig. 6. As indicated in the figure, a greater $G$ can tolerate more faults and lower the risks. So does the probability of operation failures. In turn, that means a greater granularity allows more failures for the same risk level. A more interesting research question is to choose $G$ for given switch points and failures. Since a greater $G$ implies lower availability, the problem becomes an optimisation problem of the parameters. In this paper, we do not extend our work to the optimisation of parameters.

VI. Conclusion

In this paper, we aim at quantifying the risk associated with the version switch for the synchronous rolling upgrade. We model and analyse the synchronous rolling upgrade with discrete time Markov chains and then we study the version switch that can occur to avoid the mixed version race problem, and we show how to use the DMTC model to choose a version switch point so we can control the risk. We evaluate the risk quantification on AWS through a set of experiments and also through a simulation. The experiment results indicate that the risk of early version switch can be effectively managed. For a large scale system, the techniques in this paper can be helpful in managing and scheduling software update. In the paper, we do not discuss the optimisation of the parameters. Thus that is our future work.

Acknowledgment

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References


