Recovery for Failures in Rolling Upgrade on Clouds

Min Fu, Liming Zhu, Len Bass, Anna Liu
Software Systems Research Group, NICTA, Sydney, Australia
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{Min.Fu, Liming.Zhu, Len.Bass, Anna.Liu}@nicta.com.au

***Abstract*—When cloud consumers are performing rolling upgrade operations on cloud applications, they may encounter failures due to cloud uncertainty. For example, unreliable cloud API calls can make the rolling upgrade operation fail in unpredictable and subtle ways. This paper proposes two recovery strategies for recovering from rolling upgrade failures. The strategies are Compensated Undo & Redo and Reparation. We evaluated our recovery strategies on Asgard-based rolling upgrade operation on Amazon Cloud based on two evaluation metrics: MTTR and Service Performance. The experiment results show that our strategies perform better than the recovery mechanisms provided by Asgard itself. We also conduct a comparison between the two recovery strategies based on the metrics.**

Keywords—cloud consumer; cloud API; rolling upgrade; recovery strategy

I. Introduction

One cloud consumer initiated operation on cloud applications is rolling upgrade[1]. In a rolling upgrade, a subset of instances currently running an old version of a software System are taken out of service and replaced with the same number of instances running a new version of the software system[1]. Rolling upgrade is the industry standard technique for moving to a new version of the software[1]. It can be done manually or with the assistance of automation such as cloud APIs[2] and OpsWorks scripts[4][10][11]. Several existing tools can be utilized to do rolling upgrade on cloud applications. One of them is Asgard [13], an open source cloud management tool provided by Netflix[5].

When Asgard is performing a rolling upgrade, it calls relevant cloud API functions. For instance, it will explicitly call cloud API functions such as “UpdateAutoScalingGroup”[13]. However, due to the uncertainty of cloud APIs[6], the rolling upgrade operation is error-prone[26]. For instance, if the API function of “TerminateInstanceInAutoScalingGroup” fails, the rolling upgrade itself will fail. One way to deal with those failures is to recover from them. From cloud consumer’s perspective, doing such recovery is a challenge because cloud platform only provides cloud consumers with very limited visibility and control[2].

There are several existing recovery mechanisms which can be used to recover from the errors during rolling upgrade on cloud applications[13][8][22][24]. For example, test-driven chef infrastructure[22] uses exception handlers to deal with the failures during rolling upgrades. And cloud management software (such as Asgard[13] or OpenStack[23]) is using built-in exception handlers[24] to take recovery actions. One of the challenges of recovery through exceptions handling is that it has to cater for cross-platform and cross-language exceptions[26]. Instead, our research proposes a recovery method in a non-intrusive manner. Our recovery method does not require any modification to the source code of rolling upgrade, and does not need to change any configuration settings on rolling upgrade as well.

The recovery method we propose currently contains two recovery strategies: Compensated Undo & Redo and Reparation. Compensated Undo & Redo returns the system to the previous expected state and redo the relevant steps; Reparation forcefully makes the current erroneous state into the expected state. Our experiment results on our test case show that our recovery method takes on the order of seconds as opposed to Asgard’s reliance on manual recovery. Our experiment results also show that our recovery is able to recover from more types of failures in rolling upgrade than Asgard itself. Hence, we demonstrate that our recovery method is better than the existing recovery mechanisms provided by Asgard. We also evaluate and compare the two recovery strategies based on a set of metrics: 1) MTTR; and 2) Service Performance (CPU and memory overhead)[3][7]. For each particular step of rolling upgrade, we compare the two different recovery strategies, and select the better one.

Our research has two main contributions: 1) we propose two recovery strategies for failures during operations such as rolling upgrade on cloud applications; 2) we evaluate the recovery strategies and demonstrate how to select the recovery action by using a set of evaluation metrics.

II. Rolling Upgrade on Cloud Applications

This section describes rolling upgrade on cloud and the failures that might happen during the rolling upgrade.

A. *Consumer-initiated Rolling Upgrade*

A rolling upgrade operation is one example of consumer-initiated sporadic operations on cloud applications[9]. Rolling upgrade operations normally occur less frequently than normal system operations. For example, system upgrade might happen once every week, while system normal operations such as system execution workflow can happen every day. Fig. 1 describes the seven-step procedure of rolling upgrade operation used by Asgard. This rolling upgrade operation procedure is derived from the process mining[12] of operation logs and the analysis of the source code of Asgard[13].

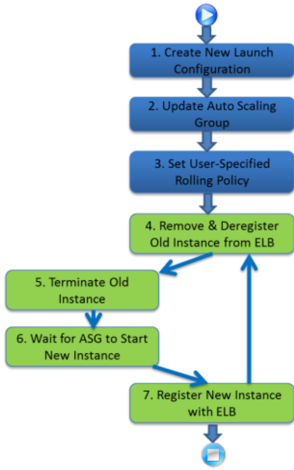

Fig. 1. Asgard Rolling Upgrade Operation. This operation consists of 7 steps, where step 1 to step 3 are sequential, and step 4 to step 7 are iterative. In step 1, new LC pointing to new AMI is created; in step 2, the existing ASG is reattached to the new LC; in step 3, the rolling policy (including instance killing order and killing number) specified by user is set; from step 4 to step 7, the system removes old instance from ELB and terminates it, then it relies on ASG to launch new instance and register new instance in ELB. Steps 4 to 7 are iteratively executed until all the old instances are upgraded.

B. Failures during Rolling Upgrade

Due to the uncertainty and instability of cloud APIs[6], failures could happen during the rolling upgrade operation. Table I enumerates some possible failures that could happen during rolling upgrade operation. We encapsulate the recovery strategies in a recovery service module that is external to Asgard. Although it is feasible to implement the recovery mechanisms in the exception handlers of Asgard source code since Asgard is open source, we choose a non-obtrusive technique to all for future generalization.

TABLE I. SOME FAILURES DURING ASGARD ROLLING UPGRADE

Error	*Step*
1. LC created with wrong parameter value	Step 1
2. ASG attached to another wrong LC	Step 2
3. Rolling policy not set as expected	Step 3
4. Instance termination taking long time	Step 4
5. Instance unable to deregister from ELB	Step 5
6. Instance unable to launch	Step 6
7. Instance unable to register with ELB	Step 7

C. Rolling Upgrade Operation as a Process

Our approach of analyzing the operation in a recoverability-oriented fashion is to analyze the operation as a process consisted of several recoverable steps[14][21]. This approach is inspired by the recoverability of BPEL processes[15][16]. If failures occur inside an operation process, one recovery technique is to roll back the process to the starting point of the process, as with a database transaction rollback[18]. In our research, we make assertion evaluation at the end of each operation process step and then recover from the failures detected. The details of our recovery mechanism will be provided in subsequent sections.

III. OUR RECOVERY APPROACH

System context knowledge is significant for recovery, for example, in message-passing distributed systems, checkpoints are utilized for a backward recovery[20]. In our research, the context knowledge of the system is comprised of two parts: 1) the rolling upgrade operation execution workflow which is modeled based on the operational process and 2) the system's expected global states which are manually predefined based on the rolling upgrade model. The recovery actions rely on context knowledge. Basically, we have two recovery mechanisms: 1) Compensated Undo & Redo; 2) Reparation. Our Compensated Undo & Redo algorithm is implemented by leveraging the system's global expected states during the operation, so is our reparation algorithm. For either of these two mechanisms, there could be more than one recovery actions. For example, there can be multiple ways to undo the current state. We utilize a set of recovery evaluation metrics to evaluate and compare these recovery actions. The metrics contain two aspects: MTTR (Mean Time to Recovery), Service Performance (CPU and memory overhead of the recovery service itself). Then, we select the better recovery action based on these metrics. The overview of our approach is illustrated in Fig. 2.

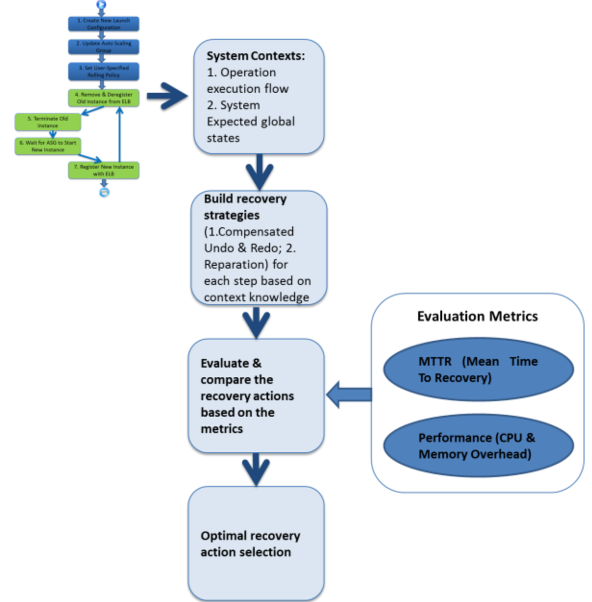

Fig. 2. Overview of Our Apporach. The system context knowledge is first generated, and then generate the recovery actions, and then evaluate each recovery action based on the evaluation metrics and make a comparison between them, and finally select the optimal action.

Step Sequence No.	**Step Description**
Step 1	Create New LC
Step 2	Update ASG
Step 3	Set Rolling Policy
Step 4	Remove & Deregister 2 Old Instances From ELB
Step 5	Terminate 2 Old Instances
Step 6	Wait for ASG to Start 2 New Instances
Step 7	Register 2 New Instance with ELB
Step 8	Remove & Deregister 2 Old Instances From ELB
Step 9	Terminate 2 Old Instances
Step 10	Wait for ASG to Start 2 New Instances
Step 11	Register 2 New Instance with ELB

Fig. 3. Operation Execution Workflow for Asgard rolling upgrade. This workflow contains 11 sequencial steps, and they represent the exection pattern of Asgard rolling upgrade. Since there are totally 4 instances and each time 2 instances are killed, steps 4 to 7 are repeated twice, hence making the whole execution contain 11 steps.

Step	Expected Global States after Step
Start	1. ASG.LC == Old LC 2. ASG.Instances == 4 version-1 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 4 version-1 instances
Step 1	1. ASG.LC == Old LC 2. ASG.Instances == 4 version-1 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 4 version-1 instances 5. LCList. HasNewLC == True
Step 2	1. ASG.LC == New LC 2. ASG.Instances == 4 version-1 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 4 version-1 instances
Step 3	1. ASG.LC == New LC 2. ASG.Instances == 4 version-1 instances 3. ASG.Instances.State == 4 version-1 instances 4. ELB.Instances == 4 version-1 instances 5. ASGRollingPolicy.Exist == True
Step 4	1. ASG.LC == New LC 2. ASG.Instances == 4 version-1 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-1 instances 5. ASGRollingPolicy.Exist == True
Step 5	1. ASG.LC == New LC 2. ASG.Instances == 2 version-1 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-1 instances 5. ASGRollingPolicy.Exist == True
Step 6	1. ASG.LC == New LC 2. ASG.Instances == 2 version-1 instances & 2 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-1 instances 5. ASGRollingPolicy.Exist == True
Step 7	1. ASG.LC == New LC 2. ASG.Instances == 2 version-1 instances & 2 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-1 instances & 2 version-2 instances 5. ASGRollingPolicy.Exist == True
Step 8	1. ASG.LC == New LC 2. ASG.Instances == 2 version-1 instances & 2 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-2 instances 5. ASGRollingPolicy.Exist == True
Step 9	1. ASG.LC == New LC 2. ASG.Instances == 2 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-2 instances 5. ASGRollingPolicy.Exist == True
Step 10	1. ASG.LC == New LC 2. ASG.Instances == 4 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 2 version-2 instances 5. ASGRollingPolicy.Exist == True
Step 11	1. ASG.LC == New LC 2. ASG.Instances == 4 version-2 instances 3. ASG.Instances.State == Running 4. ELB.Instances == 4 version-2 instances 5. ASGRollingPolicy.Exist == True

Fig. 4. Expected System Global States for Asgard rolling upgrade.

A. *Rolling Upgrade Operation Execution workflow*

Rolling upgrade operation execution workflow serves as the first type of contextual knowledge used for recovery. It represents the execution logic and execution flow of a rolling upgrade operation process. For example, in the Asgard rolling upgrade, if we put 4 instances in the auto scaling group (ASG) and we set the rolling depth to be 2 (that means 2 instances will be killed each time), the whole rolling upgrade process will contain two rolling waves: first, 2 old version instances will be deregistered from the elastic load balancer (ELB) and then terminated, then 2 new version instances will be launched and registered with ELB; second, another 2 old version instances will be deregistered from the ELB and then terminated, then another 2 new version instances will be launched and registered with ELB. In this case, the whole rolling upgrade process will contain 11 steps in total, and Fig. 3 shows its detailed execution workflow.

B. *Expected System Global States during Operations*

The expected system global states serve as the second type of contextual knowledge used for recovery. This contextual knowledge integrates all the expected global states of the system after each step during the rolling upgrade operation. The expected system global states during Asgard rolling upgrade operation are shown in Fig. 4. This time, we still have 4 instances in the ASG and the rolling depth is still 2. Hence, there are totally 11 steps, and after each step there is an expected global state. In our current research, we only care about the cloud infrastructure level of the states, such as how many virtual machines are there, or the launch configuration version attached to auto scaling group, etc. Take step 1 as the example, as denoted in Fig. 4, the expected global state after step 1 is that "New LC has been created and ASG is using old LC and 4 running version 1 instances are in ASG and they are in ELB".

IV. OUR RECOVERY ACTIONS

Our recovery actions contain two types of mechanisms: Compensated Undo & Redo and Reparation. Compensated Undo & Redo mechanism is to undo the system to the previous expected state and redo the relevant steps; Reparation mechanism is to forcefully make the current erroneous state into the expected state. The expected states will be notified to recovery service by our POD error detection and diagnosis service[26]. Fig. 5 illustrates how these two recovery mechanisms work. There is one step x and there are three global states: S_0, S_1 and S_Err. S_0 is the expected system global state before step x, and S_1 is the expected system global state after step x. S_Err is the erroneous global state after step x execution. The blue dashed arrows represent the mechanism of Compensated Undo & Redo, and the green dashed arrow represents the reparation mechanism. For the blue dashed arrows, the recovery mechanism is to first rollback system to the previous consistent state S_0 from the erroneous state S_Err, and then replay step x to make it to S_1. For the green dashed arrow, the recovery mechanism is to make the erroneous state S_Err into the expected global state S_1 for step x. However, for a general operation, the transition from S_Err to S_0 or from S_Err to S_1 is not always guaranteed to be feasible, depending on what steps are involved in the operation. For example, deleted resources are difficult to reverse. Several existing techniques[27] can be utilized to check the state reachability. After applying an existing undoability checking tool[27] on the rolling upgrade, we

fortunately found that our rolling upgrade is not bothered by such issue.

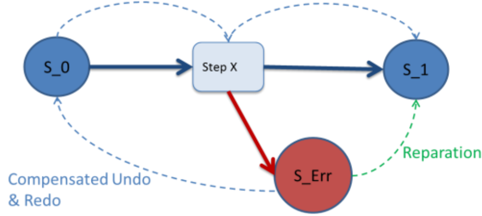

Fig. 5. Recovery Mechanisms. Two recovery mechanisms are provided: Compensated Undo & Redo, and Reparation.

A. *Compensated Undo & Redo*

The mechanism of Compensated Undo & Redo is described in Fig. 6. One example of Compensated Undo & Redo strategy for step 2 is shown in Fig. 7. It firstly undoes to the expected global state before step 2 from the erroneous state and then re-executes step 2. Fig. 6 is an abstract description of the mechanism and Figure 7 is the mechanism instantiated by the global state knowledge of Figure 4.

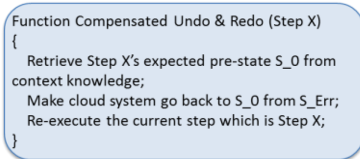

Fig. 6. Recovery Algorithm of Compensated Undo & Redo.

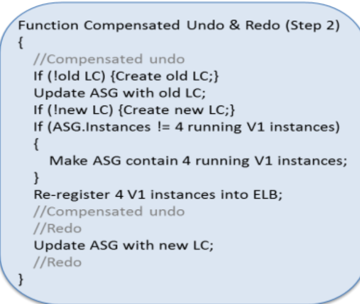

Fig. 7. Recovery Algorithm of Compensated Undo & Redo for Rolling Upgrade Step 2.

B. *Reparation*

The mechanism for Reparation is described in Fig. 8. We present one example of the reparation strategy for step 2, as shown in Fig. 9. It directly repairs the current erroneous state into the expected global state after step 2. Again, Figure 8 is an abstract specification and Figure 9 utilizes the global state knowledge of Figure 4.

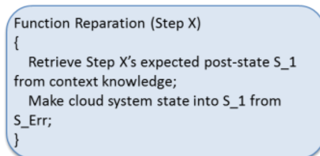

Fig. 8. Recovery Algorithm of Reparation.

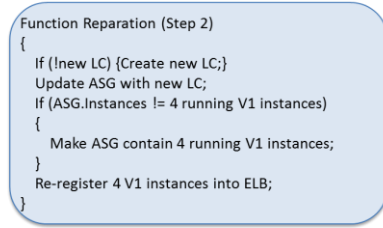

Fig. 9. Recovery Algorithm of Reparation for Rolling Upgrade Step 2.

V. RECOVERY ACTION EVALUATION METRICS

In order to evaluate the performance of our recovery actions, we utilize a set of recovery action evaluation metrics to achieve this goal. The evaluation metrics are determined by analyzing the different aspects of recovery objectives[3][7]. The recovery evaluation metrics currently contain two aspects: 1) MTTR (Mean Time to Recover) and 2) Service Performance which means the CPU and memory overhead of the recovery service itself. The mean time required for recovery is usually defined by cloud operators as a bounded time value. The recovery action which takes longer than this time constraint will be invalid. Moreover, when recovery service is running on cloud, for example, in one of the VMs, the recovery service should not introduce too much overhead to the VM itself. Otherwise the original cloud system might be impacted. There is no ordering priority for these two metrics, and cloud users may determine the priority based on their own requirements and scopes. The detailed explanation of these metrics is described below.

A. *Mean Time to Recover*

MTTR (Mean Time to Recover) is the first metric for evaluating our recovery actions. It means the time required for a recovery action to make the current erroneous state after a step into the expected state after the step. We calculate MTTR by computing the recovery algorithm running time when it is doing recovery on our system.

B. *Service Performance*

We intend to use CPU consumption rate and memory usage volume to evaluate the recovery service's performance. When the recovery service is running in the cloud, e.g. in one of the VMs, the overhead introduced by the recovery service itself should be evaluated.

VI. EXPERIMENTS & EVALUATION

Our experiment is conducted on Asgard rolling upgrade with AWS EC2 platform. Our recovery service prototype is implemented in C# language and running in Windows 7 64 bit operating system. We have 10 instances in the ASG, and rolling depth is 2.

A. *Errors Injected*

In our experiment, the errors injected are illustrated in below table II. Those errors injected here are a subset of the errors injected in our POD error detection and diagnosis service[26].

TABLE II. ERRORS INJECTED IN ASGARD ROLLING UPGRADE

Step	*Error Injected*
New LC creation	New LC missing after creation
ASG update	ASG uses unknown LC
Instance termination	Instance not terminated
Instance deregistering from ELB	Instance still registered with ELB
Instance launching	Instance launching fails
Instance registering with ELB	Instance not registered with ELB

B. *Evaluation of Recovery Actions*

According to our evaluation, our recovery service is better than the existing recovery mechanisms[13] provided by Asgard itself. Below table III provides the comparison between our recovery service and Asgard recovery mechanism.

TABLE III. OUR RECOVERY SERVICE VS ASGARD RECOVERY

Step	*Error Injected*	*Asgard Recovery*	*Our Recovery*
New LC creation	New LC missing after creation	Log error and graceful exist	Error fixed by either of two recovery strategies
ASG update	ASG still uses old LC	No Action	Error fixed by either of two recovery strategies
Instance termination	Instance not terminated	Log error and wait for instance to terminate	Error fixed by either of two recovery strategies
Instance deregistering from ELB	Instance still registered with ELB	No Action	Error fixed by either of two recovery strategies
Instance launching	Instance launching fails	Log error and wait for instance to start	Error fixed by either of two recovery strategies
Instance registering with ELB	Instance not registered with ELB	No Action	Error fixed by either of two recovery strategies

Now we compare our two recovery strategies by using the metrics based on the results obtained from running the two recovery strategies for each step for 10 times. Due to the page number limitation, we only provide the recovery details for step “ASG update” and step “Instance registering with ELB”.

The experimental results for step “ASG update” recovery are shown in Fig. 10 and Fig. 11. We can see that Reparation is selected as the better recovery strategy.

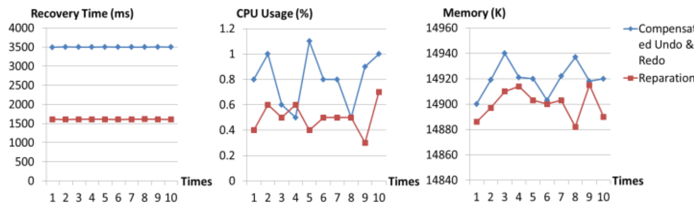

Fig. 10. Experiment Results from Running Recvoery Actions for 10 times.

Metrics	*Compensated Undo & Redo*	*Standard Deviation*	*Reparation*	*Standard Deviation*
MTTR	3499ms	0.049	1610ms	0.067
Service Performance	CPU: 0.8% Mem: 14920K	0.680 0.096	CPU: 0.5% Mem: 14900K	0.490 0.089

Fig. 11. Comparison between 2 Recovery Strategies based on metrics .

The experimental results for step “Instance registering with ELB” are shown in Fig. 12 and Fig. 13. We can see that Reparation is selected as the better recovery strategy.

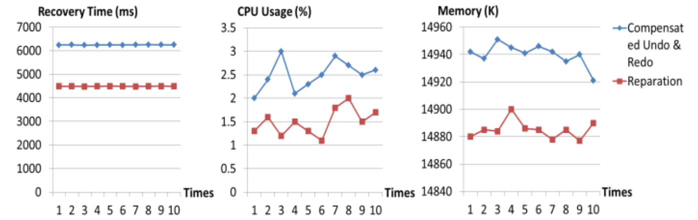

Fig. 12. Experiment Results from Running Recvoery Actions for 10 times.

Metrics	*Compensated Undo & Redo*	*Standard Deviation*	*Reparation*	*Standard Deviation*
MTTR	6245ms	0.083	4490ms	0.081
Service Performance	CPU: 2.5% Mem: 14940K	0.607 0.063	CPU: 1.5% Mem: 14885K	0.693 0.051

Fig. 13. Comparison between 2 Recovery Strategies based on metrics .

Nevertheless, our experiment results do not necessarily mean that Reparation mechanism is always better than Compensated Undo & Redo. It really depends on the specific operation steps and recovery actions.

VII. THREATS TO VALIDITY

There are some threats to validity in our research. First, we don’t test the scalability of our methods. Our experiments are only conducted by using a rolling upgrade operation which contains 10 VMs in the ASG. Hence, one of our future work items is to evaluate our methods’ scalability.

Second, our methods are based on an assumption that error diagnosis is not known. However, sometimes recovery methods require diagnosis information of errors, e.g. the error occurs due to environment issues. Moreover, our recovery does not consider error mitigation actions. Hence, our future work also includes research on error diagnosis assisted recovery which also takes mitigation actions into account.

Third, our current recovery evaluation and comparison metrics only contain two aspects, while in fact they can include more aspects (such as consequence on cloud system[3] and monetary cost[2]) . In our future work, we will also evaluate and compare recovery strategies based on finer-grained set of metrics.

VIII. RELATED WORK

A. *DDG for System Recovery*

The key system context that supports our recovery strategies is the usage of the expected system global states. This takes similarity with DDG (Data Derivation Graph) work[19][25] from UMass. DDG records how data is produced

by a running process by documenting such information as which inputs, passed to which steps, executed by which agents, resulted in the creation of which outputs[19]. One difference is that DDG is automatically generated during the process execution[19], while in our research the expected system global states are defined according to operation requirements. DDG is serving as the main system contextual knowledge which is used by process recovery actions such as undo and redo[17][25], and our expected system global states are the main contextual knowledge that is used by the two recovery strategies proposed by us for the recovery of cloud rolling upgrade operation.

B. Recovery within Long-Running Transactions

For long running transactions, recovery strategies usually involve backward recovery and forward recovery[17]. Backward recovery refers to the strategy which first reverts the current erroneous state to a previous correct state before attempting to continue execution. Forward recovery attempts to correct the current erroneous state and then continues normal execution. Our recovery strategy of Compensated Undo & Redo takes similarity to backward recovery and our recovery strategy of Reparation takes similarity to forward recovery. Another form of forward recovery in long running transactions is called compensation[17], which means to attempt to correct the state of a system given some knowledge of the previous actions of the system[17]. Generally, our recovery strategies take similarity to the recovery mechanisms for long running transactions but there are some challenges introduced such as state reachability check.

IX. CONCLUSION & FUTURE WORK

During cloud consumer initiated rolling upgrade operation on cloud applications, errors are prone to happen due to several reasons such as cloud uncertainty. To recover from errors in rolling upgrade operation, we propose a non-intrusive recovery method which contains two recovery strategies to recover from the errors happen during cloud rolling upgrade: 1) Compensated Undo & Redo and 2) Reparation. Our recovery method does not require the modification to rolling upgrade source code, nor does it require any configuration changes on rolling upgrade. We evaluate our recovery strategies by using Asgard and EC2 platform, and our experiments show that our recovery method is better than the existing recovery mechanisms provided by Asgard. We also evaluate those two recovery strategies and make a comparison between them.

In our future work, we will evaluate the scalability of our recovery strategies. And we will also evaluate the recovery strategies based on a finer-grained set of metrics which includes other aspects such as consequence on cloud system and monetary cost. Moreover, we will also figure out and implement more recovery strategies which take error diagnosis and error mitigation into account. And we also would like to make our recovery method cater for other sporadic operations on cloud such as deployment or reconfiguration on cloud.

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

[1] T. Dumitras and P. Narasimhan, "Why Do Upgrades Fail andWhat CanWe Do about It? , Middleware 2009, pp. 349-372, 2009 .

[2] AWS official Website: http://aws.amazon.com/cn/ (last access time: 4th Mar 2014, 12:09).

[3] T. Wood, E. Cecchet, et al., "Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges", HotCloud, 2010.

[4] OpsCode official Website: http://www.opscode.com/ (last access time: 4th Mar 2014, 20:06)

[5] Netflix official website: https://www.netflix.com/global (last access time: 2nd Mar 2014, 12:30).

[6] Q. Lu, L. Zhu, et al., "Cloud API Issues: an Empirical Study and Impact", Proc. 9th ACM SIGSOFT conference, 2013.

[7] J. Zhu, et al., "System Recovery Benchmarking", DSN Workshop on Dependability Benchmarking, June 25 2002.

[8] J. Behl, et al., "Providing Fault-tolerant Execution of Web-service–based Workflows within Clouds", 2nd International Workshop on Cloud Computing Platforms, 2012.

[9] M. Fu, L. Zhu and L. Bass; "A Recoverability-Oriented Analysis for Operations on Cloud Applications", WICSA 2014.

[10] AWS OpsWorks official website: http://aws.amazon.com/cn/opsworks/ (last access time: 15th Feb 2014, 15:17).

[11] Chef official website: http://www.opscode.com/chef/ (last access time: 2nd Mar 2014, 12:40).

[12] X. Xu, I. Weber, et al.; "Detecting Cloud Provisioning Errors Using an Annotated Process Model"; proc. MW4NextGen'13, no. 5, 2013.

[13] Asgard official website: https://github.com/Netflix/asgard (last access time: 2nd Mar 2014, 12:30).

[14] M. Fu, L. Zhu, A. Liu, L. Bass and X. Xu; "Process-Oriented Recovery for Operations on Cloud Applications", proc. SOCC, no. 50, 2013.

[15] L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL processes", ICSOC 2005, Springer, 2005.

[16] J. Simmonds, et al.; "Guided Recovery for Web Service Applications"; proc. 18th ACM SIGSOFT, pp. 247-256, 2010.

[17] C. Colombo and G. J. Pace, "Recovery within Long Running Transactions", ACM Transactions on Computational Logic, pp. 1-40, August 2011.

[18] T. Haerder, and A. Reuter; "Principles of transaction-oriented database recovery"; ACM CSUR, December 1983.

[19] X. Zhao, et al.; "Supporting Process Undo and Redo in Software Engineering Decision Making"; proc. ICSSP, 2013.

[20] E. N. M. Elnozahy, et al., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", (CSUR), vol. 34, no. 6, pp. 375-408, September 2002.

[21] X. Xu, L. Zhu, J. Li, L. Bass, Q. Lu and M. Fu, "Modelling and Analysing Operation Processes for Dependability", DSN 43rd Annual IEEE/IFIP International Conference, 2013.

[22] S. Nelson-Smith, "Test-Driven Infrastructure with Chef", published by O'Reilly Media, Inc., Copyright©2011 Atalanta Systems LTD, First Edition, June 2011.

[23] OpenStack official website: http://www.openstack.org/ (last access time: 4th Mar 2014, 15:17).

[24] H. Chang, et al.; "Exception Handlers for Healing Component-Based Systems"; ACM Transaction on Software Engineering and Methodology, vol. 22, no. 4, October 2013.

[25] X. Zhao, E. R. Boose, et al.; "Supporting Undo and Redo in Scientific Data Analysis"; Evaluation, 2013.

[26] X. Xu, I. Weber, et. Al.; "POD-Diagnosis: Error Diagnosis of Sporadic Operation on Cloud Applications"; Submitted to DSN 2014.

[27] I. Weber, H. Wada, et al.; "Supporting Undoability in Systems Operations"; USENIX 2013.