Recovery for Sporadic Operations on Cloud Applications

Min Fu  
Software Systems Research Group, NICTA, Sydney, Australia  
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia  
Min.Fu@nicta.com.au

Abstract—During sporadic operations (e.g., deployment, upgrade, and reconfiguration, etc.) on cloud applications, errors could happen and make the operations fail. This is due to several reasons such as cloud APIs uncertainty. One way to handle the errors is to recover from them. Some existing recovery approaches such as exceptions handling mechanisms usually do recovery by a graceful operation exit or issuing waiting time, and they are faced with the challenge of handling cross-platform and cross-language exceptions. Our research proposes a non-intrusive recovery method to recover from errors during sporadic operations on cloud applications. First, based on analyzing an operation as a process we determine the recovery points according to certain criteria. Second, optimal recovery action will be triggered if errors after any recovery point are detected. The optimal recovery action is selected from eight recovery patterns (such as Rewind&Replay, Reparation, etc.) based on certain selection criteria. We take Asgard rolling upgrade as the case study of our research. We have conducted initial evaluation of our method on Asgard rolling upgrade process and obtained some initial evaluation results.

Keywords—Cloud application; sporadic operation; recovery;

I. INTRODUCTION

Sporadic operations on cloud applications, such as deployment, upgrade and reconfiguration, are error-prone[1] and this is because of several reasons such as cloud APIs uncertainty[1]. Gartner research[2] has pointed out that majority of cloud errors are those that happen during cloud operations[2]. As such, handling those errors is a necessity which has also become a heavy research focus[3]. There are several existing recovery mechanisms[4][5][6] to recover from those errors during sporadic operations on cloud. However, those existing methods do not recover from errors in a fine-grained manner. For instance, exception handling mechanism usually do recovery by gracefully existing from operation or by providing a waiting time[4], and it is faced with a challenge of needing to cater for the exceptions from different systems and platforms[7]. Our research proposes a non-intrusive recovery method to recovery from the errors during sporadic operations on cloud. First, by analyzing an operation as a process we determine the recovery points based on certain criteria. Second, recovery will be triggered if there are errors detected after a recovery point. We have eight recovery patterns and the optimal recovery action will be selected for the recovery based on the selection criteria. More details of the method will be described in Section IV. We have implemented the prototype for two recovery patterns—Compensated Undo & Redo and Reparation. We conduct an initial evaluation for this prototype by using Asgard rolling upgrade[5] as the case study. Specifically, we compare the recoverability provided by our prototype and the recoverability provided by Asgard exceptions handling mechanism. Based on the initial experimental results, we can draw a conclusion that our recovery prototype outweighs Asgard exceptions handling mechanism in terms of two aspects: first, our prototype is able to handle more types of errors than Asgard itself can do; second, our prototype can satisfy recovery objectives[8][9] better than Asgard exception handling mechanism. We also make comparison between the two recovery patterns in terms of the recovery metrics: MTTR[8], Consequence[9], Cost[10], and Action Performance.

II. RELATED WORK

The related work for our research is provided below. We discuss exceptions handling mechanism and recovery within long running transactions here.

A. Exceptions Handling Mechanism

Exceptions handling mechanism[12] is one way of doing recovery. First, the errors in the form of exceptions are caught by the exception handler. Then the exception handler will handle the errors by doing certain recovery actions. Existing exceptions handling such as Asgard exception handling usually exist gracefully from the operation or provide a waiting time for certain step of the operation. What’s more, exception handling mechanism is faced with the tricky problem of cross-platform and cross-language exceptions catching and analyzing[7]. Our non-intrusive recovery method targets to provide fine-grained recovery actions rather than just existing from operation or just waiting longer.

B. Recovery within Long Running Transactions

Recovery mechanisms within long running transaction include two strategies: Backward recovery and Forward recovery[11]. An example of Backward recovery is Undo & Redo[11], and an example of Forward recovery is Compensation[11]. Below Fig. 1 shows the details of these two strategies. For Backward recovery, it transits from the error state (State_Err) to the previous state (State 1), and redo step x;
for Forward recovery, it directly transits from the error state (State_Err) to the expected state (State 2) after the step x.

![Fig. 1. Recovery within Long Running Transactions](image)

III. RESEARCH QUESTIONS & CHALLENGES

In order to fulfill our research goal, there are three research questions to be answered: 1) How to recover from operation errors in a real-time way? 2) How to make sure the recovery actions are able to meet recovery objectives[9]? 3) How to make a recovery action itself recoverable? To answer these questions, we start by analyzing the challenges of doing recovery. The main challenges for doing recovery for sporadic operation on cloud include: 1) the limited visibility and control provided by cloud platform restricts our recovery actions design; 2) recovery action may turn into an invalid solution if error reason is not known; 3) recovery due to false positives in error detection and diagnosis might have bad consequence on system; 4) unavoidable tail-latency for some cloud APIs might impact recovery actions’ RTO. We address these questions and challenges in our proposed recovery approach. Generally, our recovery approach will leverage the cloud on-demand nature[10] in a fine-grained way. The recovery approach should be able to minimize the time required for recovery and impose minimum bad consequence on cloud system.

IV. OUR APPROACH OVERVIEW

Our recovery approach contains two steps: first, we determine recovery points for the operation process; second, optimal recovery action will be selected and triggered if errors are detected to have occurred after any recovery point. We propose eight recovery patterns: 1) Compensated Undo & Redo; 2) Compensated Undo & Alternative; 3) Rewind & Replay; 4) Rewind & Alternative; 5) Reparation; 6) Direct Redo; 7) Direct Alternative; 8) Farther Undo & Redo. For each of these patterns, maybe more than one recovery actions exist. But only the optimal recovery action will be selected for execution. To select the optimal action, it first selects the feasible actions after state reachability analysis and idempotence analysis, and then it select the optimal action from those feasible ones based on the selection criteria. Fig. 2 shows the overview of our approach.

![Fig. 2. Recovery Method Overview](image)

A. Recovery Points Determination

After analyzing the operation as a process[13], we first determine the recovery points for the operation process, and the recovery points show where recovery will take place if errors occur. The recovery points are determined based on certain criteria[14], and the criteria include: 1) Idempotence to enable the same step to be re-executed for recovery; 2) Granularity to allow re-execution of combination of internally dependent steps during recovery; 3) Recovery Actions Identifiable to make sure certain recovery actions can exist. To justify and demonstrate the recovery points determination method, we use Asgard rolling upgrade as the case study, as shown in Fig. 3. Due to the page number limitation, we do not describe Asgard rolling upgrade process here. Based on the criteria, there are 6 recovery points (hence 6 sections) determined.

![Fig. 3. Recovery Points Determination](image)

B. Recovery Patterns

We provide eight recovery patterns and they are illustrated in Fig. 4. The recovery is triggered after process section X+1. S0/S1/S2 is the predefined expected state before and after a process section. C0/C1 is the checkpointed state before and after a process section. S_err is the actual erroneous state after the process section. A process section could have its alternative which leads to the same running result as the process section itself (Alternative X+1 has the same running result as Process Section X+1). For Compensated Undo & Redo, the recovery flow is “S_err->C1->Section X+1->S2”. For Compensated Undo & Alternative, the recovery flow is “S_err->S1->Alternative X+1->S2”. For Rewind & Replay, the recovery flow is “S_err->C1->Section X+1->S2”. For Rewind & Alternative, the recovery flow is “S_err->C1->Alternative X+1->S2”. For Reparation, the recovery flow is “S_err->S2”. For Direct Redo, the recovery flow is “S_err->Section X+1->S2”. For Direct Alternative, the recovery flow is “S_err->Alternative X+1->S2”. For Farther Undo & Redo, the recovery flow is “S_err->C0->Section X->Section X+1->S2”. One potential challenge here is that transition from error state to another state (e.g. S_err->S1) may not always be feasible. For example, deleted resources are difficult to be reversible. To check for state reachability, we can rely on several existing techniques such as state reachability checking[6].

![Fig. 4. Recovery Patterns](image)
Recovery Action Selection

Optimal recovery action is selected based on two steps: 1) Select feasible recovery actions and 2) select optimal one from feasible actions based on selection criteria consisted of MTTR, Consequence, Cost, and Action Performance.

MTTR (Mean Time to Recover) 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.

Consequence on System refers to the judgment of bad consequences and impact on the rolling upgrade procedure and the cloud system itself. Recovery actions should not interrupt the performance of rolling upgrade procedure and the cloud system, even when they are unexpectedly executed due to false positives of error detection and diagnosis.

Commercial Cloud services are not free, and AWS has its own benchmark of calculating the service price cost. Based on the benchmark provided by AWS, we can calculate if additional price is caused when our recovery actions are functioning.

Action Performance means the CPU consumption rate and memory usage volume of the recovery actions. 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.

The optimal recovery action will be the one whose overall cost of the four criteria is the least, and such recovery action will be selected.

V. INITIAL IMPLEMENTATION & EVALUATION

We have implemented the prototype of two recovery patterns: Compensated Undo & Redo and Reparation. For each of these recovery patterns, we have provided one recovery action (algorithm). In fact, for each recovery pattern, there can be two or more recovery actions. And in the future work we will implement more. There are several process sections, however, due to the page number limitation, we only describe the recovery actions for process section 2 which is “Update ASG with New LC”.

Fig. 4. Recovery Patterns for Operations on Cloud

![Recovery Algorithm of Compensated Undo & Redo](image)

Fig. 5. Recovery Algorithm of Compensated Undo & Redo

![Recovery Algorithm of Reparation](image)

Fig. 6. Recovery Algorithm of Reparation

In our experiment, in order to evaluate our recovery algorithms, we first inject errors into Asgard rolling upgrade process. The errors injected are illustrated in below Table I.

<table>
<thead>
<tr>
<th>Step</th>
<th>Error Injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>New LC creation</td>
<td>New LC missing after creation</td>
</tr>
<tr>
<td>ASG update</td>
<td>ASG still uses old LC</td>
</tr>
<tr>
<td>Instance termination</td>
<td>Instance not terminated</td>
</tr>
<tr>
<td>Instance deregistering from ELB</td>
<td>Instance still registered with ELB</td>
</tr>
<tr>
<td>Instance launching</td>
<td>Instance launching fails</td>
</tr>
<tr>
<td>Instance registering with ELB</td>
<td>Instance not registered with ELB</td>
</tr>
</tbody>
</table>

According to our evaluation, our recovery service far outweighs the existing recovery mechanism provided by Asgard itself. Below table II provides the comparison between our recovery service and Asgard recovery mechanism.

<table>
<thead>
<tr>
<th>Step</th>
<th>Error Injected</th>
<th>Asgard Recovery</th>
<th>Our Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>New LC creation</td>
<td>New LC missing after creation</td>
<td>Log error and</td>
<td>Error fixed by either of two</td>
</tr>
<tr>
<td></td>
<td></td>
<td>graceful exist</td>
<td>recovery</td>
</tr>
</tbody>
</table>

Fig. 5 shows an example of Compensated Undo & Redo pattern. It firstly undoes to the expected global state before section 2 from the erroneous state and then redoes section 2. Fig. 6 shows an example of Reparation pattern. It directly repairs the current erroneous state into the expected global state after section 2. One challenge here is that the recovery action itself should also be made recoverable since it is also calling those delicate cloud APIs[1].
Now we show the comparison between these two recovery algorithms. Due to the page number limitation, we only provide the details of error recovery for step “ASG update” and step “Instance registering with ELB”.

The experimental result for step “ASG update” recovery is as below:

<table>
<thead>
<tr>
<th>Step</th>
<th>Error Injected</th>
<th>Asgard Recovery</th>
<th>Our Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASG update</td>
<td>ASG still uses old LC</td>
<td>No Action</td>
<td>Error fixed by either of two recovery strategies</td>
</tr>
<tr>
<td>Instance termination</td>
<td>Instance not terminated</td>
<td>Log error and wait for instance to terminate</td>
<td>Error fixed by either of two recovery strategies</td>
</tr>
<tr>
<td>Instance deregistering from ELB</td>
<td>Instance still registered with ELB</td>
<td>No Action</td>
<td>Error fixed by either of two recovery strategies</td>
</tr>
<tr>
<td>Instance launching</td>
<td>Instance launching fails</td>
<td>Log error and wait for instance to start</td>
<td>Error fixed by either of two recovery strategies</td>
</tr>
<tr>
<td>Instance registering with ELB</td>
<td>Instance not registered with ELB</td>
<td>No Action</td>
<td>Error fixed by either of two recovery strategies</td>
</tr>
</tbody>
</table>

We can see that MTTR of Reparation algorithm is less than that of Compensated Undo & Redo algorithm, and Performance of Reparation algorithm is better than Compensated Undo & Redo algorithm. For Consequence and Cost, both of the two algorithms are the same. Hence, Reparation is better in this case.

The experimental result for step “Instance registering with ELB” is as below:

<table>
<thead>
<tr>
<th>Cost Metrics</th>
<th>Compensated Undo &amp; Redo</th>
<th>Reparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTTR</td>
<td>6245ms</td>
<td>4490ms</td>
</tr>
<tr>
<td>Consequence</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Cost</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Performance</td>
<td>CPU: 2%-3%; Mem: 14680-15200K</td>
<td>CPU: 1%-2%; Mem: 14670-15100K</td>
</tr>
</tbody>
</table>

We can see that MTTR of Reparation algorithm is less than that of Compensated Undo & Redo algorithm, and Performance of Reparation algorithm is better than Compensated Undo & Redo algorithm. For Consequence and Cost, both of the two algorithms are the same. Hence, Reparation is better in this case.

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. Moreover, state transition is not necessarily always guaranteed. In our further experiments, we will try to address the cases where it is unfeasible or difficult to transit from the erroneous state to another state.

VI. CONCLUSION

During sporadic operations on cloud applications, such as deployment, upgrade, and reconfiguration, errors could happen. This is due to several reasons such as cloud APIs uncertainty[1]. One way to handle the errors is to recover from them. Our research proposes a non-intrusive recovery method to recover from such errors. First, we determine the recovery points for operation process. Second, the optimal recovery action will be triggered if errors after any recovery point are detected. The optimal recovery action is selected from eight recovery patterns (such as Rewind&Replay, Reparation, etc.) based on the selection criteria. We intend to evaluate our recovery method by comparing our method with Asgard exception handling mechanism.

So far, we implement the prototype for two recovery patterns (Compensated Undo & Redo and Reparation) and conduct the initial evaluation. Our initial evaluation results show that our prototype recovery service outweighs the recovery mechanism provided by Asgard itself.

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