A Tail-Tolerant Cloud API Wrapper

Qinghua Lu1,2, Xiwei Xu2, Len Bass2, Liming Zhu2, Weishan Zhang1
1College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China
2Software Systems Research Group, NICTA, Sydney, Australia
dr.qinghua.lu@gmail.com; xiwei.xu@nicta.com.au; len.bass@nicta.com.au; liming.zhu@nicta.com.au; zhangws@upc.edu.cn

Abstract—System operations (such as deployment, upgrade, reconfiguration) for cloud applications are failure prone. These failures are because these operations are performed through cloud APIs provided by cloud providers and cloud APIs, in turn, are failure prone. In this paper, we explore the characteristics of cloud APIs using Amazon EC2 as a test bed and propose mechanisms for improving cloud API performance. Specifically, we mined the Amazon EC2 forum and observed that 45% of complaints referred to the timing failure of cloud APIs. We then conducted a series of experiments on the cloud API timing behavior and observed that cloud APIs have long tail characteristics. Finally, we propose a cloud API wrapper that implements mechanisms to avoid the long tail, and demonstrate that our wrapper largely removes the long tail compared with the unwrapped APIs.

Keywords—cloud computing, API, fault tolerance, long tail, dependability

I. INTRODUCTION

Applications running in the cloud are rarely static in terms of their versions, their deployment, or their configurations. Internet companies report upgrading dozens of times a day or more [1]. Performing these system operations whether through web interfaces, command line interfaces or specialized tools relies heavily on cloud infrastructure APIs (Application Programming Interfaces). Regardless of the interface type, upgrading, redeployment, or configuration all use cloud platform APIs behind the scenes to complete the specified operations.

Existing research on system operations focuses on reducing errors and repair time [2-3] rather than investigating latency issues. However, cloud consumers have limited visibility and control of the cloud infrastructure. Cloud platform API calls are the only interaction points between the cloud infrastructure and the system operations. Therefore, the completion time and reliability of operations depends on the reliability and performance of API calls.

We previously published an early version of our cloud API mechanisms in [4]. In this paper, we explore the performance of cloud APIs. We first argue that cloud API calls are unresponsive for a significantly large percentage of invocations. We then show that cloud APIs suffer from a long tail distribution. Finally, we propose a cloud API wrapper with tail-tolerant mechanisms (such as hedging and alternative requests) and show that our cloud API wrappers largely remove the long tail compared with native unwrapped API calls.

II. EMPIRICAL STUDY OF CLOUD API ISSUES

We first observed cloud API issues during the development of our commercial disaster recovery product Yuruware Bolt1 that relies heavily on APIs to perform disaster recovery operations. In the process of exploring these issues we extracted 2087 API failure cases from a wide range of sources (a broader empirical study than our initial study [5] which only covers 922 cases). These cases are not errors, which manifest in an error code or crash-stop fashion or which can be significantly reduced after locating the causes. A large percentage (45%) of the cases are unavoidable latency or timing failures (i.e. stuck API calls and slow response to API calls) that cannot be reduced in a large-scale system and often exhibit a crash-recovery behavior. We then conducted a series of experiments on the timing behavior of the provisioning services and observed long tail characteristics. Those timing failures of API calls are major causes of the long tail of the timing distribution of operation tasks.

A. API Issues Reported in Public Sources

We extracted failure reports from a variety of public sources including the Amazon EC2 discussion forum2 as well as technical analysis of API issues during outages from reputable sources (such as Amazon outage reports3, Netflix technical blogs4 and AvailabilityDigest5). We also explored 32 cloud platforms that provide similar APIs to Amazon EC2 to supplement the main study although Amazon accounts for the vast majority (88.8%) of the items in this study. We classified the 2087 API failures into two sub-types: content failures (55% of API issues) and timing failures (45% of API issues).

1) Content Failures

Content failure means “The content of the information delivered at the service interface deviates from implementing the system function” [6]. 55% of the failures fall in this category. We further classify content failures into four sub-types: 1) failed calls with error messages (49%), 2) missing content (4%), 3) wrong content (2%), and 4) unexpected content (2%). Those content failures are mainly caused by API bugs. For most of the API bugs reported by the consumers, cloud vendors fix the reported bugs and release a new API

1) Yuruware Bolt - http://www.yuruware.com/
3) Amazon outage reports - http://aws.amazon.com/message/65648/
5) AvailabilityDigest - http://www.availabilitydigest.com/

---

1 College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China
2 Software Systems Research Group, NICTA, Sydney, Australia
4 Amazon outage reports - http://aws.amazon.com/message/65648/
5 AvailabilityDigest - http://www.availabilitydigest.com/
Version. However, the correction process normally takes several weeks.

2) Timing Failures
Timing failures means that the arrival time of the delivered information at the interface deviates from implementing the system function [6]. 45% of the reported cases complain about slow responding API calls or stuck API calls.

Sometime, the API response time is slow but a response does eventually arrive. For example, a reported late timing failure is that it took 16 minutes for the instance to stop (posted on Aug 27, 2012 11:57 AM in the EC2 forum). The EC2 engineer advised to try “force stop” twice if this happens next time. Many users experience longer time to complete (e.g. half hour) or stuck API calls. However, the API document does not specify the normal time to complete each API call.

Sometimes, the API calls do not return at all, which causes silent failures. One case reports that the instance is stuck at stopping (posted on June 27, 2012 12:04 AM in the EC2 forum). The EC2 engineer stopped the instance for the user at the EC2 side. However, the user complained that the volume was decommissioned while the user hoped to reuse it.

B. Experiments
We did experiments to measure the 5 most frequent EC2 API calls (“launch instance”, “start instance”, “stop instance”, “attach volume”, and “detach volume”) by calling the API 1000 times, and recording the return time. Figure 1 gives the measurement of “launch instance” API. Since the focus of this paper is long tail, we removed the API calls that failed with error messages. The calculation of the percentage is still based on 1000 calls. The horizontal axis represents the return time of an API call while the vertical axis represents the percentage of the corresponding return time values among the return time of the total 1000 API calls.

Figure 1. Measurement results of EC2 “launch instance”.

We choose 0.5% of the distribution as characterizing the long tail. That is, the long tail constitutes 0.5% of the area under the distribution curve. This cut off corresponds to 36s. Thus, all the data points that are equal and greater than 36s (in red) represent the long tail. This translates into approximately 4.5% of the “launch instance” API calls. The other four API calls have similar timing profiles.

Figure 2. Architecture of API Wrapper.
III. API Wrapper

The architecture of our API wrapper is shown in Figure 2. A cloud user requests an API through the API wrapper. The API wrapper interacts with the original cloud infrastructure API (i.e., EC2 API). We maintain a repository of API timing profiles, which stores the corresponding waiting time for each API. When an original API is issued by the API wrapper, an API is requested. A timeout occurs when a timeout occurs. If the API request fails or there is no response from the cloud, the API wrapper processes the API calls based on the state chart shown in Figure 2. When the API wrapper triggers a tail-tolerant mechanism, the cost estimation component analyses the costs of the selected mechanisms and prints the costs in the log.

There are two state charts superimposed in Figure 2. One is the original EC2 workflow and the other shows the amended workflow that corrects for a long tail delay. Both state charts have six states: Requested, Allocated, Started, Cancelled, Failed, and Completed. The solid arrows depict the transitions between these states of the original EC2 APIs during normal execution. The dashed arrows depict the transitions when a timeout occurs when the wrapper is a particular state. For example, if a timing failure occurs when the request to ec2-describe-instances is force-completed, the API call to the Started state, the API wrapper transfers the API call to the Allocated state. If the command ec2-describe-instances does not return any output, the API wrapper transfers the API call to the Started state.

The initial state is Requested. When an API call request is intercepted by the API wrapper, it enters the Requested state. The Requested state may choose to make a normal request, a hedged request or an alternative request if the first request fails. The hedge-request mechanism is similar to “hedges requests” idea in [9]. For certain operations, we will issue more requests than we need (e.g., launching 12 instead of the 10 we need) and then cancel the remaining immediately after the required number is successfully reached. The alternative-request mechanism requests an alternative API call at the same time as the original API is requested.

Allocated state

API calls that need to allocate resources, such as attaching volumes, then transfer to the Allocated state, other API calls transfer to the Started state.

At the Allocated state, the API wrapper could use mechanisms of continue-allocate, re-allocate, cancel-allocate or force-complete-a. The continue-allocate mechanism schedules the request to be sent to the same instance at a future time if the EBS volume is not able to be attached to the instance, the application can try to attach it to a different instance within the same availability zone. Another example is that, the application can ignore the current request and send the API request again to cloud infrastructure. The reallocate mechanism transfers the API call back to the Allocated state. The cancel-start mechanism can be used when a started API call is stuck at a state. It transfers the API call to the Completed state.

The initial version of our API wrapper wraps around Amazon EC2 APIs. For supporting multiple cloud infrastructures, we will have separate wrappers for different cloud providers. Our solution is not about a standardized API across clouds, which requires users to change their code. However, it is possible our mechanisms could be across clouds behind the scenes.

Our initial API wrapper6 provides a tail-tolerant version of five API calls: launch-instance, start-instance, stop-instance, attach-volume and detach-volume. These five EC2 API calls are the most frequently used and have significant latency issues according to both our own experience and our empirical study of EC2 forum [5]. The implementation details of the initial version of our API wrapper is shown in Table 1. We built a timing profile for each API call. After the waiting time reaches a configurable 90 percentile of historical return time, we resort to other mechanisms immediately.

6 API Wrapper - https://sites.google.com/site/cloudapiwrapper/
Table 1. Initial version of our API wrapper

<table>
<thead>
<tr>
<th>API wrapper</th>
<th>Mechanism</th>
<th>Implementation details</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch-instance</td>
<td>hedge-request; continue-allocate</td>
<td>The API wrapper launches one or more redundant instances through making multiple launch-instance API calls simultaneously when it receives a request to launch instances. If enough instances are launched within the time specified in the time profile of launch-instance, the API wrapper will kill the redundant ones when it is launched. If there is not enough instances launch successfully, then the API wrapper re-launches more instances.</td>
</tr>
<tr>
<td>start-instance</td>
<td>alternative-request</td>
<td>The API wrapper starts an instance and launches a new instance using the same image simultaneously, and cancels the one with longer return time.</td>
</tr>
<tr>
<td>stop-instance</td>
<td>force-complete-a</td>
<td>The API wrapper launches a call to the stop-instance API, waits for the time specified in the time profile of stop instance. If the call is not completed, the API wrapper forces the instance to stop using the API of “force stop”.</td>
</tr>
<tr>
<td>attach-volume</td>
<td>alternative-request</td>
<td>The API wrapper attaches volume to an instance and launches a new instance at the same time. The wrapper waits for the time specified in the time profile of attach volume. If the call is not completed, it re-attaches the volume to the newly launched instance.</td>
</tr>
<tr>
<td>detach-volume</td>
<td>force-complete-a</td>
<td>The API wrapper waits for the time specified in the time profile of detach-volume. If not completed, then the API wrapper force-detaches the volume.</td>
</tr>
</tbody>
</table>

The hedge-request mechanism and alternative-request mechanism will lead to extra costs when the API calls create resources which are launch-instance, start-instance and attach-volume in the current version of our API wrapper. The extra cost is based on the price of the rented resource and usage duration. In our API wrapper, hedge-request issues one extra API call, thus, the extra cost of launching a t1.micro instance is 0.02 dollar. Larger instances will cost more. Users need to be aware of these costs.

It becomes an organization’s decision whether to incur these extra costs. If the upgrade, for example, fixes a significant bug then an organization may decide to assume the additional costs. If the upgrade is of lower priority, the organization may decide not to pay the additional charges.

IV. EFFECTIVENESS OF API WRAPPER

We evaluated our long-tail tolerant mechanisms through experiments. We compared the return time of the API wrapper to the original API.

A. Experiment Setup

Our experiments ran on AWS EC2 us-east-1d availability zone. We used an AMI which installed AMP (Apache 2.2.22, MySQL 5.5.35, and PHP 5.3.10) software stack on a t1.micro type instance.

For each of the APIs we wrapped in our API wrapper, we measured the return time 1000 times respectively. The 1000 API calls were issued in sequence within AWS EC2. As before, we removed the API calls that failed with error messages. The calculation of the percentage is still based on 1000 calls.

![Figure 3. Basic statistics of return time (unit: second).](image-url)
Our experimental result shows that the API wrapper and the original EC2 API have similar distribution of the return time, including mean, median, STD, 95 percentile and 99 percentile. The y axis represents the percentage of the corresponding return time value among the return time of the total 1000 API calls. The blue dots represent the original EC2 API, which uses the same dataset as Figure 1. The small triangles represent the API wrapper. The longest return time of our API wrapper is 51s.

Although the probability of long-tail return time is low, the time of a long tail is very long. Sometimes it could be as long as 10 times of the mean of the return times. Our experimental results show that our API wrapper with API tail-tolerant mechanisms can substantially reduce the long tail of the original EC2 API. Our API wrapper can significantly reduce the impact of API issues on operations long-tail and improve the reliability of operations in the cloud.

Since our solution provides the same API to the user through the API wrapper it does not require users to change their scripts/code calling the API.

One limitation is that this paper only evaluated the current version of our API wrapper which implements 4 out of 9 proposed mechanisms. We have confidence in the other version of our API wrapper which implements 5 out of 9 proposed mechanisms because they are alternative mechanisms covering the same types of issues and some have been implemented in Yuruware Bolt commercial workflow systems.

V. CONCLUSION

In this paper, we proposed a new set of tail-tolerant mechanisms to tolerate long-tail issues of operations in cloud. We implemented our mechanisms as a tail-tolerant wrapper around Amazon cloud APIs which are heavily used in system operations of applications. Our evaluation shows that the mechanisms can significantly reduce the long tail.

ACKNOWLEDGMENT

This project is supported by National Natural Science Foundation of China (Grant No. U1402533), “the Fundamental Research Funds for the Central Universities” (No. 14CX02140A) and “the Scientific Research Foundation of China University of Petroleum” (No. Y1307021).

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

REFERENCES


Qinghua Lu is a lecturer at Department of Software Engineering, China University of Petroleum, Qingdao, China. She received her PhD from University of New South Wales (UNSW) in 2013. Her research interests include software architecture, dependability of cloud computing, and service engineering.

Xiwei Xu is a researcher at NICTA, Australia. She received her PhD degree from University of New South Wales (UNSW), in Sydney, Australia, in 2011. Her research interests are in software architecture, business process and cloud computing.
**Len Bass** is a senior principle researcher at NICTA, Australia. He has written two award winning books in software architecture as well as several other books and numerous papers in area of software engineering. His research interests include software architecture, software engineering, and DevOps.

**Liming Zhu** is a principle researcher at NICTA, Australia. He holds conjoint academic positions and teaches software architecture courses at both University of New South Wales (UNSW) and The University of Sydney. His research interests include software architecture, service engineering and system development methodologies.

**Weishan Zhang** is a full professor, deputy head for research of Department of Software Engineering, China University of Petroleum. His current research interests are big data platforms, pervasive cloud computing, and service oriented computing. Weishan has published over 50 papers and his current h-index according to google scholar is 10.