EVALUATION OF REST INTERFACES FOR RELIABLE INVOCATION SEMANTICS

Ricardo Filipe
CISUC, Dept. of Informatics Engineering, University of Coimbra
Coimbra, Portugal

Filipe Araujo
CISUC, Dept. of Informatics Engineering, University of Coimbra
Coimbra, Portugal

ABSTRACT

Nowadays, the REpresentational State Transfer (REST) architectural style plays a central role in distributed software. All major web and cloud providers, such as Facebook, Dropbox, Google, Instagram, Amazon, among many others, expose public REST interfaces for web, mobile and stand-alone clients. This raises the question of knowing the reliability of these REST services, as they run over the network, thus being inevitably prone to failures. While some of these failures might be harmless, and others might be overcome with a simple repetition of the same request, a few others, e.g., payments, need a much more careful treatment. In these cases, repeating the same request might not be an option, because duplicate executions might be worse than no execution at all.

To answer this question, and determine the reliability of top-tier REST interfaces, we evaluated three real services, from leading web and cloud providers. We repeatedly invoked their REST functions for several days, to count the number of unsuccessful interactions. Despite running smoothly for most of the time, our initial results show that all these services suffered perturbations during the course of our experiment. This observation supports a straightforward conclusion: if even at this level we observe this number of problems, then, developers must definitely provide additional logic on the client and server, to ensure reliable interactions in the presence of faults. This experiment is part of a larger effort, to provide clear guidelines for reliable invocation semantics in web pages.

KEYWORDS

REST, Reliability, Invocation semantics

1. INTRODUCTION

REpresentational State Transfer (REST), first defined by Fielding in his PhD thesis in 2000 (Fielding (2000)), has become a major standard for services provided over the Internet. The power of REST comes from the versatility of interfaces that simultaneously support the presentation layer of web pages developed in JavaScript, mobile applications that are native for Android or iOS, and interaction with third-party providers, most notably for the sake of authentication, but by no means limited to it. The modern computing infrastructure, including the cloud, heavily relies on REST services as these became ubiquitous. The number and names of web site providers using REST and making their REST interfaces available for programmers is huge: Facebook, Twitter, Instagram, Amazon, Microsoft, Google, YouTube and countless others.

In spite of this success, critical web interactions still pose a major challenge for developers, whenever operations fail to execute properly. Services involving reservation of some resource, payments, or other business-oriented interactions cannot easily tolerate requests with error responses or no responses at all.

This is neither new, nor specific to REST, as the effect of faults in distributed systems is a well understood topic that deserved thorough attention for decades. However, the extreme popularity of the running REST interfaces, the sheer number of available operations and the always increasing number of users they support, provide us new opportunities to evaluate the reliability of real, highly relevant distributed systems. In particular, we would like to know exactly how likely is it for an operation requiring an exactly-once semantics to fail without providing adequate information to the client.

To achieve this goal, we ran an experiment using successive HTTP invocations to REST services of three major providers of video and file storing and sharing. We did this for several days, keeping notice of response times, unanswered requests, HTTP status of the responses and HTTP headers. To avoid disclosing sensitive information regarding these services, we keep them anonymous and refer to them by the letters A, B and C.

The results are noteworthy: even major providers cannot avoid a significant number of errors in the access to their REST resources and a wide disparity of response times. While some of these problems come from the network, the provider itself is also to blame in many cases. Although small in relative numbers, errors do not only exist, but exist in large amounts, when we consider the vast utilization of these services. Hence, we can clearly conclude that, to build reliable distributed systems, developers must use mechanisms that take connectivity, availability and performance problems into account. The results of this paper aim to motivate developers for the central problem of achieving reliable invocation of services over the network. If it is true that abundant theory exists about the problem (Spector (1982)), it is also true that developers have no widespread guidelines that can help them to properly protect their programs against invocation failures. The solutions we are aware of are ad hoc, error-prone and subject to memory leaks.

2. EXPERIMENTAL SETTINGS AND RESULTS

To evaluate the reliability of online REST services, we ran a client that regularly requested operations from three well-known service providers. All three providers reside in the west coast of the United States, whereas the client is in Portugal, around 9,000 Kms away. We used the crontab scheduler on a Linux machine, to run the cURL command, which invoked the same REST resource during several days, with intervals of 4 seconds. We accessed two file sharing providers (A and B) and a video sharing provider (C). In the file sharing providers, we did a REST request to get the contents of the home directory of a user account; in the video sharing provider, we did a REST request to get the 50 most popular videos. To access data from provider A, we had to use the HTTP POST method; from the other two providers, we used the GET method. Note that according to the HTTP protocol (Fielding and Reschke (2014)), the POST method is non-idempotent. Repeating an invocation might thus have an unintended effect.

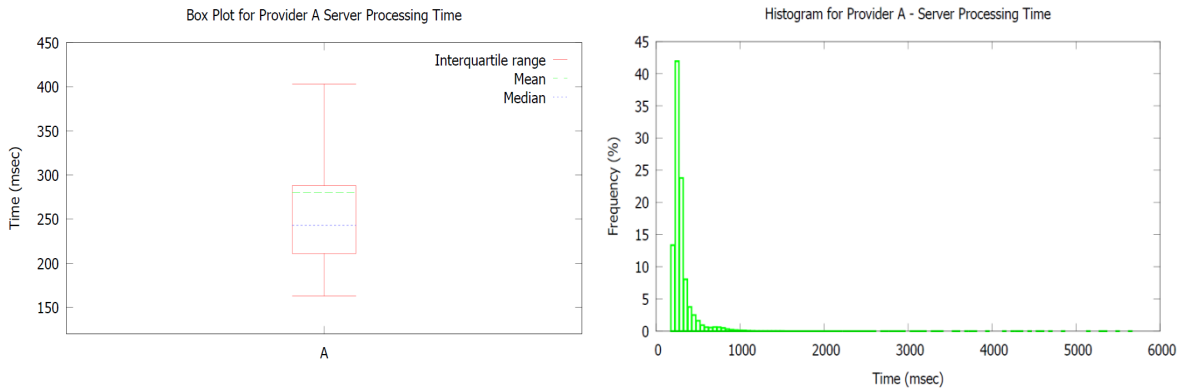

Figure 1 - Provider A server time statistics

We also gathered a specific HTTP extension header from provider A, which stated the time that the server took to answer the request. This extension allowed us to determine server-side times, regardless of network performance. In Figure 1-left, we show the box plot associated with the server response time, while on the right side, we show the histogram of the same samples, distributed by the time it took to answer the request (x axis) and the number of times each time interval occurred (y axis). Looking at both plots, we can observe that the servers of provider A take around 250 *ms* to reply to the majority of requests, once they arrive. However, as we can see in the box plot, the actual times vary a lot. Additionally, if we take a closer look to the histogram, we verify that sometimes the server responses take as much as 6000 *ms*. These response times (plus the network latency) may be treated as a timeout by the client software or users, which might be tempted to reload or resubmit forms several times during this interval.

Concerning the network time between our client and the servers of providers A, B and C, in Figure 2 we show a box plot for the server+network times of our samples. Service invocations will necessarily involve large network latencies, as light takes around 60 *ms* to cover the round-trip (in vacuum). Interestingly, provider C presents a very stable connectivity, having very short interquartile ranges and a mean similar to the median. The results of provider C are truly remarkable, suggesting, not only, that providers A and B take long internal times to fulfill the service, but also that such times have a large variance (i.e., they cannot be attributed to the network). These differences might result from the nature of the services involved. It might be simpler to prepare the response to a request for the most popular 50 videos in store, than it is to provide the list of files in the home directory of a specific user. The precise reasons, however, are difficult to discern, without knowing the internal details of these providers. Furthermore, as far as we could tell, the requests we did to provider C were not served by any cache.

The comparison of Figures 1 and 2 also suggests that the total internal server time of provider A is larger than what we see in Figure 1. Furthermore, since variances for providers A and B are somewhat large, they might pose some problems for client-side software, especially if developers want to wait for (outlier) responses taking more than the 75 percentile to arrive.

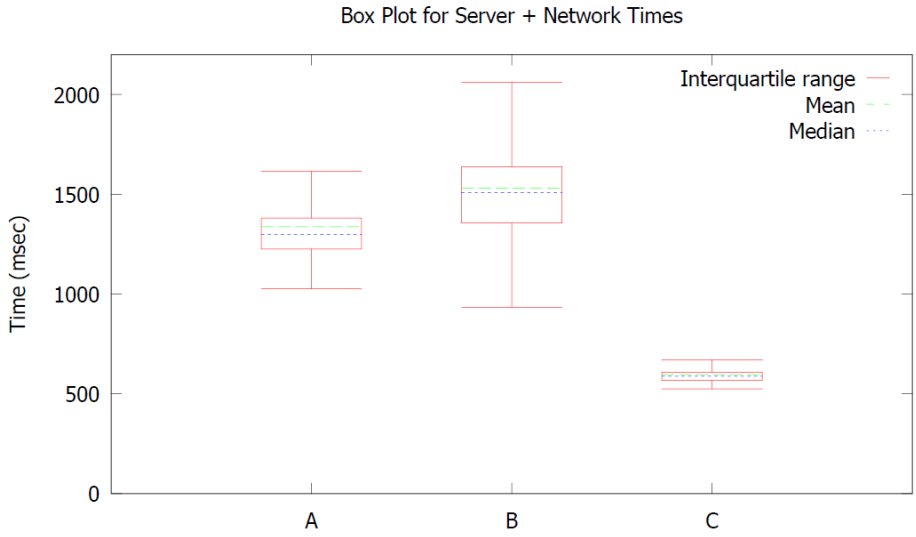

Figure 2 – Server + Network time statistics for the distinct providers

Table 1. Number of errors

	Provider A	Provider B	Provider C
Connection setup errors	104 (0.057%)	43 (0.018%)	6 (0.005%)
Connection crashes	5 (0.003%)	28 (0.012%)	7 (0.005%)
Provider errors	38 (0.02%)	3 (0.0012%)	0 (0%)

Our next step was to count the errors in response to our requests (refer to Table 1). We divide errors as seen by clients in three different classes: network errors related to the connection setup phase (first line), network errors occurring *after* the connection is setup (second line), and application errors (third line). Network errors include connection timeouts, connection crashes, read timeouts, or unreachable hosts. Application errors occur when the client gets an HTTP error in the ranges 4xx or 5xx (we only got 5xx, since we kept using a limited list of addresses in our requests). We can see that the number of failed requests is quite relevant, especially when one accounts for the intensive utilization of these services. Most errors are due to the network interaction, whereas only a few occur in the application, caused by some temporary server overload, for example.

While service providers can invest in resources and software to improve their network and the response of the application, connection setup errors and connection crashes might be out of their control, as they mostly depend on third party providers. On the other hand, for developers, the most difficult problems to overcome are connection crashes and provider errors, because applications can hardly be sure about the outcome of the REST invocation, when these happen. This situation might be even worse for mobile clients, as intermittent

connections will certainly cause more errors, thus raising the importance of tolerating ambiguous, absent or erroneous responses.

3. DISCUSSION AND CONCLUSION

We are by no means the first ones to evaluate or benchmark online services. For example, Palma et al. (2014), made a study to identify the lack of REST patterns in major operators. In Laranjeiro et al. (2009), authors evaluated the robustness of web services, using invalid call parameters, to observe programming or design errors. Although we also care for web services, our main concern is on the reliability of invocations, like in Ivaki et al. (2015) or Shegalov et al. (2002). Indeed, we are not aware of other work comparing distinct providers in terms of error counting, request response times, and their distribution.

The importance of knowing the normal behavior and the frequency of errors in online services comes from the paramount importance of ensuring that non-idempotent REST requests execute at most once. However, when the user receives an error and risks to resubmit the request, s(he) will be left without knowing whether the request was executed zero, one or more times. This is an inevitable source of mistrust about the service provider and ultimately contributes to the degradation of the provider's reputation.

Although we recognize the need for collecting additional information from other top web providers, the evidence we collected so far strongly supports the idea that developers need to employ fault-tolerance techniques, if they use non-idempotent services. Depending on the provider, the fraction of invocations that fail with ambiguous results might be as large as 0.02 percent. Unfortunately, despite being a well-known problem, we are not aware of any widespread technique that can result in a standard implementation of the at-most-once or (in certain conditions) exactly-once execution semantics in HTTP.

From previous interactions with web site developers, we realized that one common technique they use, to ensure that all operations are idempotent, is to include a unique identifier with the request, and provide the same identifier on subsequent submissions of the same request. The difficulty of this approach concerns the timely deletion of old responses from the server cache. Another possibility would be te use the HTTP Header *If-Unmodified-Since* with the date of the last (supposed) update. The server would either respond with success, if the resource was not modified after the date, or respond with an *HTTP 412 PreCondition Failed* otherwise. However, this mechanism is not ensured by most providers; furthermore, this approach would force the server to associate a date to each resource.

As a final remark, we can say that 1) we showed in this paper that problems in non-idempotent REST invocations occur frequently, and 2) there is no ready-to-use reliable invocation solution for web developers. This motivates us to work on such a proposal, based on our own review of all protocols that might ensure at-most-once and exactly-once invocations (Ivaki et al. (2015)).

REFERENCES

Fielding, R. & Reschke, J., (2014), *Hypertext transfer protocol (http/1.1): Message syntax and routing*, RFC 7230, RFC Editor. http://www.rfc-editor.org/rfc/rfc7230.txt.

Fielding, R. T. (2000), *REST: Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine.

Ivaki, N. et al, (2015), A taxonomy of reliable request-response protocols, *in Proceedings of the 30th Annual ACM Symposium on Applied Computing*, New York, USA, pp. 456–463

Laranjeiro, N. et al, (2009), Improving web services robustness, *in 'Web Services, 2009. ICWS 2009. IEEE International Conference*', Los Angeles, USA, pp. 397–404.

Palma, F. et al, (2014), Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach, *Service-Oriented Computing: 12th International Conference, ICSOC 2014*, Paris, France, pp. 230–244.

Shegalov, G. et al, (2002), Eos: Exactly-once e-service middleware, *in 'Proceedings of the 28th International Conference on Very Large Data Bases', VLDB '02*, Hong Kong, China, pp. 1043–1046.

Spector, A. Z. (1982), *'Performing remote operations efficiently on a local computer network'*, Communications of the ACM 25(4), 246–260.