ERROR ISOLATION AND MANAGEMENT IN AGILE MULTI-TENANT CLOUD BASED APPLICATIONS

Amrinder Arora

Department of Computer Science, The George Washington University,
Washington DC, USA

ABSTRACT

Management of errors in multi-tenant cloud based applications remains a challenging problem. This problem is compounded due to (i) multiple versions of application serving different clients, (ii) agile nature in which the applications are released to the clients, and (iii) variations in specific usage patterns of each client. We propose a framework for isolating and managing errors in such applications. The proposed framework is evaluated with two different popular cloud based applications and empirical results are presented.

KEYWORDS

Cloud computing, Agile development, Error isolation, Multi-tenancy

1 INTRODUCTION

There are two related, but distinct forces in the business of software, which are in motion today. Firstly, applications are increasingly being developed for, or migrated to, the "cloud". Secondly, applications are being developed and deployed in an agile manner. These two forces support and enable each other, although each could potentially sustain itself on its own. In this paper, we study a core problem that lies at the intersection of these two forces. While a detailed analysis of these two forces is outside the scope of this paper, a brief introduction to cloud based multi-tenant application architecture and agile development is provided next.

In cloud based application architecture, applications are only deployed on a handful of servers as opposed to thousands of client machines. The physical location of the servers may not be inside the organization's premises, rather it is with a cloud hosting provider, such as Amazon, Google or Microsoft (the "cloud"). Clients access the applications via web browsers, or via web services through internet based protocols. Multiple clients can access the same instance of the application. Business applications typically use the concept of a "tenantid", which is a unique key that represents the business or the organization that is accessing the cloud based applications. This key is used to provide data separation so that one organization's datais not exposed to other organizations.

In agile development methodology, applications do not go through a cascading waterfall based development lifecycle. Instead, small changes are made to the code base in each iteration and releases are constantly pushed to the clients. Thus, new features are released rapidly to the clients, sometimes more than once a week and may be released to some clients first and slowly released to all clients. Code development is typically supported by a mix of automated and manual testing. The extent of testing depends upon the maturity of the cloud application provider, and varies from very extensive to almost no testing.

More thorough analysis and background of these topics can be found in [1], [2], [3], [4] and [5].

DOI : 10.5121/ijccsa.2015.5101

2 PROBLEM STATEMENT

The basic error isolation and management problem in a multi-tenant environment is as follows: We are presented with application logs from multiple servers. From these application logs, we need to extract a list of bugs and then to produce an ordered list of those bugs based on the provided logs. A schematic of this is shown in Figure 1.

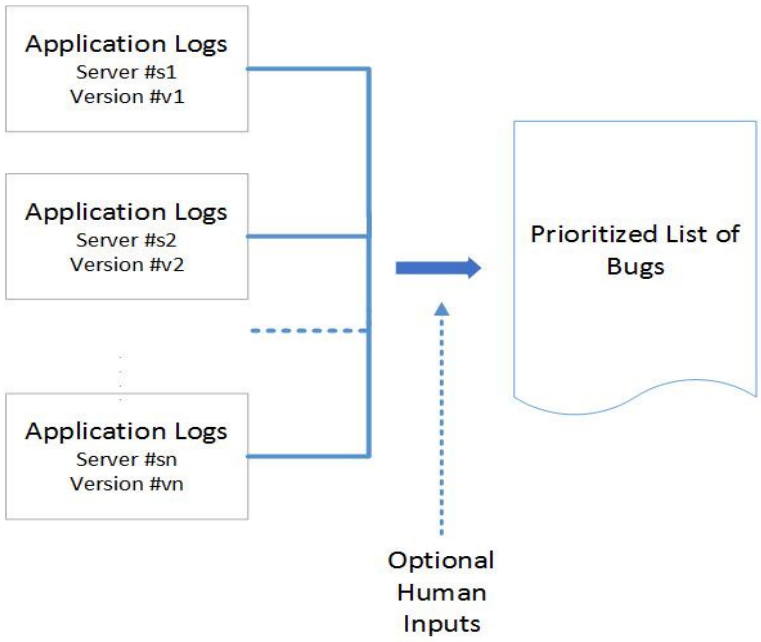

Figure 1: Basic error log and isolation problem in multi-tenant applications. Each server maybe a physical server, a virtual server, or simply a separate instance running on the same physical/virtual server. Each instance may be serving multiple tenants (clients).

The prioritization may require input from a human user regarding the severity of various errors. Further, this prioritization may then be used within the larger product management and feature prioritization process.

3 PROPOSED SOLUTION

Some versions of this and related problems have been studied in [2] and [5]. Similarly, significant work has been done in the closely related field of logging [6]. We propose a framework consisting of eight different phases. Using the initials of the names of the 8 phases, the framework is named Mapricot.

3.1 Mapricot Framework

In this section, we describe the Mapricot framework in detail. We describe the various steps of the framework as well as the importance of those steps in the overall solution.

3.1.1 M – Measurable space

The first step in the Mapricot framework is to store (or load) the errors in a measurable, analyzable space. Such space may be a database table. Due to the availability of many big data tools and plugins now a days (such as Hadoop, Flume, Kafka, etc.), there may be some variations in what works best for different applications, but in general, this step can be accomplished using following sub steps:

1. Design a database table to hold all the errors or use a big data framework that does not need a predefined table structure.
2. Define a utility (or a web services end point)that does the following:

    a. Catches (or receives)exceptions or service errors,
    b. Creates the hash of the exception/error to create a simplified error, and
    c. Stores the exceptions/errors to the database (which can be a relational database or HDFS storage).

The hashing and extraction of exception/error details (step 2b above) can be accomplished using the following logic.When an exception happens, usually the exception stack trace is available. We use an extraction (hashing) algorithm as specified in Figure 2 to create an error code from the trace.

// Algorithm EST2EC creates an error code from a given exception trace
// Exception Stack Trace consisting of *n* exception trace elements.

Algorithm EST2EC(Input: ExceptionStackTrace, Output: String errorCode)

1. Ignore/prune any external components that are not in organization's application logic. For example, if the application (written in Java) has the prefix com.mycompany.myproduct. The application may use external components such as Oracle or Java classes in the code and therefore these external components may also appear in the stack trace. However, in case these external components are not in control of the organization, they can be removed from the exception trace so that organization can focus on the aspects of code that it can directly fix or improve.
2. Create a hash of the product and module names itself. The product and module names are usually reflected in the package structure and the class names.
3. Using steps 1 and 2, we reach a simplified stack trace that is of form:
    1. M1, F1, line 232
    2. M2, F2, line 230
    3. M3, F3, line 400
    4. etc.
4. In case stack trace is longer than 10, prune trace elements except first 5 and bottom 5.
5. Create a hash of the pruned string using a standard hashing algorithm, such as MD5.

Figure 2: Algorithm to create an error code from an exception stack trace

Time complexity of Algorithm EST2EC: We observe that while the given exception stack trace may consist of up to *n* elements, the algorithm EST2EC is a constant time O(1) algorithm. This is due to the fact that in step 4, the given exception trace is limited to only the first 5 and bottom 5 elements. Given an array or an array list, a sub-list can be extracted in constant time.

3.1.2 A – Analyze the Errors

In the analysis phase of Mapricot, errors are categorized and a frequency count/histogram generated. This phase can be initiated in an automated fashion and can be performed using automated batch scripts (for example, using pig/HIVE scripts). Any business intelligence application can be used to visualize weekly and monthly histograms from the error tables. A typical output is as follows:

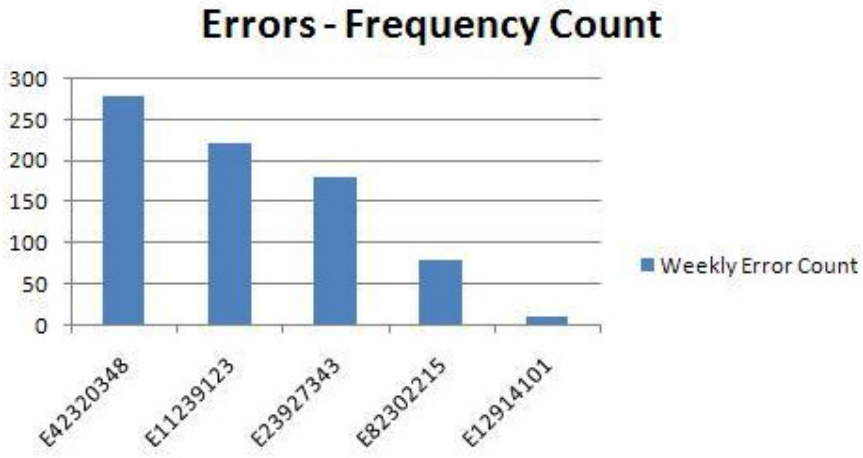

Figure 3: Weekly histogram of error counts. Error codes are as generated in Step 2 of Mapricot

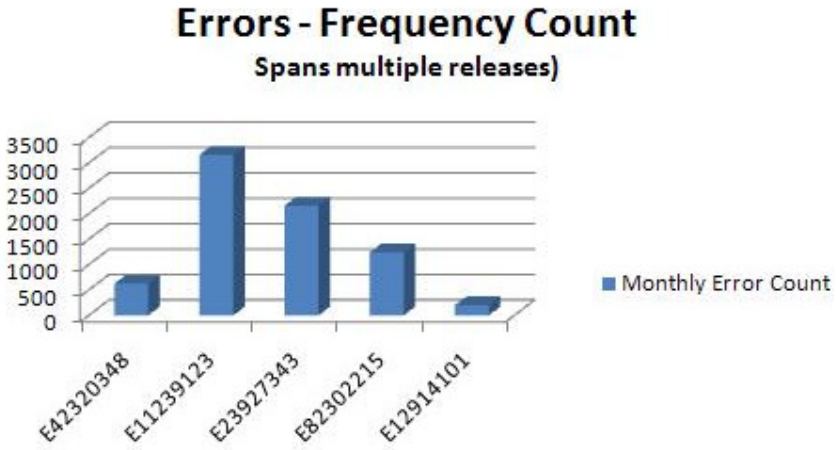

Figure 4: Histogram of error codes that may span multiple releases

Using these histograms, it can be relatively quickly decided which errors require follow up actions (improved logging/offer urgent production/code improvements/training). The analysis phase thus serves as the router and creator for the follow up tasks in the Mapricot framework.

3.1.3 P – Prioritization of Errors

The prioritization of errors is usually the most critical decision point. The analysis phase of Mapricot framework creates histograms and basic analytical structures that can be used in prioritization. While the error counts are obviously important, a few other aspects of the error may be even more important in the overall prioritization. These aspects include:

1. **Effect of an error**: For example, if an error is simply causing an error to be logged when there is no data loss or incorrect functioning of the system, that may result in the error receiving lower prioritization even if it happens significantly more often. As a counterexample, an error which does not happen very often but causes the system to incorrectly invoice a customer may receive a higher prioritization even if the frequency of the error is significantly lesser. The effects of an error can only be determined by the business analysts or application owners and we do not attempt to automate this step.
2. **Effort required for a fix**: Similar to the effect of an error, the effort required to fix an error also has a significant impact on the prioritization of the error. If an error is easily fixable, then it may receive a higher prioritization compared to an error that needs much more effort to be fixed.
3. **Possibility of a buildless fix**: Many errors can sometimes be fixed without the need to reprogram the system, simply by changing some of the configuration options of the system. In other words, they require very little effort to fix them. Consequently, such errors may receive high priority, even if their counts are low and the effects of such errors are not very significant.

Cost Benefit Analysis: Using the standard cost benefit analysis approach, we observe that the "effect of the error" is the benefit analysis, and the "effort for the fix" is the cost analysis for each of the possible action items.Therefore, the objective of the prioritization phasecan be stated as follows: given the output of the analysis phase, update the cost benefit values. As noted earlier, this may involve using manual inputs from the team of application owners, business analysts and technical architects.

Error ID	Description	Count (Observation)	Effect (Potential Benefit)	Effort to Fix (Cost)
E2384842	Unable to send hourly notifications	168	Loss in user engagement	Runtime property change (SMTP host)
E2171922	Unable to change password	365	Security vulnerability, user trust loss	3 person days
E9419421	User over invoiced	5	Regulatory issue, user trust loss, revenue loss	14 person days

Table 1: List of errors including the effect and effort to fix.

3.1.4 R – Release correlation with errors

The objective of the 4th phase in Mapricot is to correlate the errors with specific releases. Considering agile methodology, new releases are constantly released. Therefore, the normalized error counts per day with different releases are used to generate an error correlation.

Error Code	Count
E42320348	279
E11239123	221
E23927343	181
E82302215	81

Table 2: Error frequency chart for Release R4.2.1

Error Code	Count
E11239123	1124
E23927343	741
E82302215	445
E12914101	73
E42320348	16

Table 3: Error frequency chart for Release R4.2.2

Error correlation can then be easily generated using a standard set difference algorithm.

3.1.5 I – Improved Logging

The step 5 of Mapricot process consists of modifying code to improve the logging even, based on the observations of the analysis phase. This can include unknown exceptions, or other exceptions with inadequate details in the log. An interestingmistake that is commonly observed is in the exception handling code, in which the catch clause simply absorbs an exception, logs some generic message, does not print any details about the specific exception and also does not rethrow the exception.

Example 1:

```
try {
// functionality
} catch (Exception e) {
log (“Exception in the functionality!”);
// No details of the exception e, and generic statement
}
```

In this, the log statement is generic, and loses out on all the details of the exception. A much better alternative is to rewrite it as follows:

```
try {
// functionality
} catch (Exception e) {
log (“Exception in saving: ” + e, e); // Logs the message, and the exception
// throw e; // Rethrow the exception if suitable.
}
```

Example 2:

```
try {
// functionality
} catch (Exception e) {
throw new Exception (“Exception message”); // bad code – lost the original
exception!
}
```

This code throws a NEW exception, which is not a wrapper on the old exception. More appropriate is to do:

```
try {
// functionality
} catch (Exception e) {
throw new Exception (“Exception in functionality”, e); // Wrap the exception
// May also be possible to rethrow the exception
}
```

Some exception classes do not have a constructor that takes the inner exception as an argument. In that case, the initCause method can be used to wrap the original exception.

Another category of code instances where “I” phase is helpful is where the root cause of the error is not being diagnosed despite logging of the exception already. In this case, more log statements can be added that print the actual data/payload contents and then, the logging configuration can be adjusted in this case to either enable or disable those logs as needed. This is illustrated in the following example:

```
if (matchingLocations == 0) {
       throw new IllegalArgumentException (“No matching location found, for lat,
long:” + latitude + “, “ + longitude);
}
```

One possible improvement that we may then be able to make is:

```
if (matchingLocations == 0) {
       throw new IllegalArgumentException (“No matching location found, for lat,
long:” + latitude + “, “ + longitude + “, OBU: ” + onboardUnit.toXML());
// “Improved” logging- While it may not fix the error,
// it may help to identify the cause of the error in next release logs
}
```

Therefore, the “I” phase of MAPRICOT is a way to improve some logging about errors for which there is insufficient information. The problems for which the “I” phase applies in the current build, will likely be problems that for which the prioritization phase applies in the next build.

3.1.6 C – Code improvement for preventing and/or diagnosing errors

In the sixth phase of the Mapricot framework, the idea is to change or improve any code that can help in preventing or diagnosing errors. Sometimes the error is simply in logging, for example, logging a communication specification (which can happen anytime from a user’s browser) as an application exception. In that case, the code can be improved to not log that as an error.

3.1.7 O – Offer urgent help

The seventh phase of the framework involves more aspects of customer service and less aspects of software, still it is an important phase. The essence of this phase is to remind that while the errors may be resolved in later releases, the current set of users does need urgent help by way of workarounds and manual data corrections.

3.1.8 T – Training with respect to error handling

Assuming errors are not critical, but still a user annoyance (for example, record cannot be saved if using a certain option), then it may be preferable to offer any user training that can be given to alleviate situation.This may be imparted as a training to the users, or a training to the customer support team where they can offer the help to the users contacting them. Similar to the seventh phase of the framework, this step also has more aspects of customer service, and therefore we have kept the description of these two phases brief.

4 EMPIRICAL RESULTS

4.1 Context and Setting

Mapricot framework was evaluated in two applications. Application 1 is leading business management application used for resource management. Application 2 is a leading reverse supply chain management and optimization application. Both applications are cloud based, multi-tenant applications. Both applications support configuration and customizations, and different clients (tenants) use those applications differently.

To evaluate the performance of the Mapricot framework, error logs were collected for 3 months of control period, followed by 3 months of usage period. Errors by week were plotted by week. Since a higher usage translates into higher number of errors, errors were normalized by usage.

4.2 Results

In case of application 1, which was more mature application, the error rate was decreasing per usage in the control period also. However, after the adoption of Mapricot, the error rate dropped consistently week after week and tapered off to a small value.

In case of application 2, which was in earlier stages of the maturity curve, the data was much more noisy. The errors by week increased and decreased a bit randomly, but the overall trend was a decrease as well.

Subjectively, the application architects suggest that the mere fact that the errors were being tracked and were being counted and analyzed made them more cognizant of the application errors and increased the priority. This change can be attributed merely to the very first step (measurable space) of the Mapricot framework.

5 CONCLUSIONS

In this paper, we have studied the problem of managing errors in multi-tenant cloud based applications. This problem remains challenging, due to the fact that in such a setting we typically have multiple versions of application serving different clients, and also due to the agile nature in which the applications are released to the clients. Further, every client uses the application differently. We have proposed a framework for isolating and managing errors in such

applications, which was evaluated with two different popular cloud based applications. We have presented the context and the results from those experiments.

ACKNOWLEDGEMENTS

Authors would like to thank the management team of BizMerlin for partnering in evaluating the suggested framework and for sharing the anonymized results of the program.

REFERENCES

[1] Youseff, L.; Butrico, M.; Da Silva, D., "Toward a Unified Ontology of Cloud Computing," Grid Computing Environments Workshop, 2008. GCE '08, pp.1-10, 12-16 Nov. 2008doi: 10.1109/GCE.2008.4738443

[2] Rimal, B.P.; Eunmi Choi; Lumb, I., "A Taxonomy and Survey of Cloud Computing Systems," NCM '09. Fifth International Joint Conference on INC, IMS and IDC, 2009, pp.44-51, 25-27 Aug. 2009, doi: 10.1109/NCM.2009.218

[3] Buyya, R.; Ranjan, R.; Calheiros, R.N., "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities," HPCS '09. International Conference on High Performance Computing & Simulation, 2009, pp.1-11, 21-24 June 2009, doi: 10.1109/HPCSIM.2009.5192685

[4] Gong C.; Liu J; Zhang Q; Chen H; Gong Z, "The Characteristics of Cloud Computing," Parallel Processing Workshops (ICPPW), 2010 39th International Conference on , vol., no., pp.275,279, 13-16 Sept. 2010 doi: 10.1109/ICPPW.2010.45

[5] Fehling, C., Leymann, F., Rütschlin, J., &Schumm, D. (2012). "Pattern-based development and management of cloud applications". Future Internet, 4(1), 110-141.

[6] Shams Zawoad, Amit Kumar Dutta, and Ragib Hasan. 2013. "SecLaaS: secure logging-as-a-service for cloud forensics", 8th ACM SIGSAC symposium on Information, computer and communications security (ASIA CCS '13). ACM, New York, NY, USA, pp.219-230. doi=10.1145/2484313.2484342

AUTHORS

Dr. Amrinder Arora is an adjunct professor in the Department of Computer Science at the George Washington University. As an adjunct faculty member in the Department of Computer Science at the George Washington University, Prof. Arora teaches graduate and undergraduate courses in computer science, mostly related to design and analysis of computer algorithms and design of data structures.Dr. Arora is a leading expert in risk targeting and data analytics and the author of the book "Analysis and Design of Algorithms" (Cognella/University Readers). He is also the author of the popular software blog at http://www.standardwisdom.com