Determine Configuration Entry Correlations for Web Application Systems

Conference Paper · June 2016

DOI: 10.1109/COMPSAC.2016.95

CITATIONS
4

READS
33

5 authors, including:

Wei Chen
Chinese Academy of Sciences
31 PUBLICATIONS **116** CITATIONS
SEE PROFILE

Heng Wu
Chinese Academy of Sciences
9 PUBLICATIONS **41** CITATIONS
SEE PROFILE

Jun Wei Wei
Institute of software, Chinese Academy of Sciences
124 PUBLICATIONS **715** CITATIONS
SEE PROFILE

Hua Zhong
Chinese Academy of Sciences
80 PUBLICATIONS **475** CITATIONS
SEE PROFILE

2016 IEEE 40th Annual Computer Software and Applications Conference

Determine Configuration Entry Correlations for Web Application Systems

Wei Chen, Heng Wu, Jun Wei, Hua Zhong, Tao Huang
Institute of Software
Chinese Academy of Sciences
Beijing, China
Email: {wchen, wuheng09, wj, zhongh, tao}@otcaix.iscas.ac.cn

***Abstract*—Web application systems, comprising of heterogeneous and loosely coupled components, are usually highly-configurable due to the large number of configuration entries scattering in the components. The dependencies between components lead their entries correlate to one another, which makes the system deployment and migration daunting and error-prone. For two correlated entries, changing value of one entry requires the value change of the other. Otherwise, some implied constraints would be violated and the system failure will occur. Keeping track of entry correlations, which is essential to system reliabilities, is not a simple work as it often crosses products and requires in-depth domain knowledge.**

This paper proposes a method to automate the process of determining entry correlations. The method first narrows down the exploring scale to those frequently-set entries based on crawled sample data. Then, it generates a correlation score for each entry pair, which is calculated according to entry names, values and inferred types. Thirdly, a set of heuristics are provided to determine a candidate set of the likely correlations. Finally, a rank-ordered list of entry correlations is output so that system administrators can consult it to check system configuration systematically. Based on the method, we implement a tool, Correlation Explorer, and make experiments and evaluations with some real world systems. The result shows that Correlation Explorer is effective in finding a large portion of entry correlations.

I. Introduction

Web application systems (systems for short) are usually constituted of many heterogeneous components, and each of which has a lot of configuration entries. For example, MySQL 5.6 database server has 461 configuration parameters, and Apache HTTP server 2.4 has more than 550 parameters across all the modules [1]. The multi-tiered software stacks and the large system scales result in that there are thousands and hundreds of configuration entries scattering in the systems, making the system configurations error-prone and time-consuming.

Configuration error has become one of the major causes of today's system failures [2]. The misconfiguration-induced outages have been reported from major IT companies, including Microsoft, Amazon and Facebook [3][4][5]. Among the configuration errors, the entry correlation (correlation for short) induced ones take a great proportion. An empirical study [6] shows that a significant percentage (12.2%∼29.7%) of configuration errors are correlation-induced.

In a system, a part of correlations are introduced in by the dependencies between system components. For example, a service component depends on a DBMS, the database connection entries of the service correlate to some others of the DBMS, then the entries *database name*, *user name* and *user password* of the service must be consistent with the values of those ones of the DBMS. Zhang et al. [7] report that 27%∼51% of the configuration entries in their studied open-source software projects have correlations with another ones.

However, keeping track of entry correlations, particularly cross-component ones, is a thorny issue. 1) These entry correlations may cross components and there are no explicit declarations; 2) the documents of systems may be inconsistent with software and even missing; 3) multiple programming languages may be employed and program analysis based method [8] are impractical [9][10]. Though domain experts are involved in, it is difficult to find a single person who has knowledge spanning the various components and software [11]. Once some entry correlations are overlooked, the specific configuration constraints may be violated and system errors will occur.

In this paper, we propose a method to determine the cross-component entry correlations. Unless stated, the entry correlations refer to those cross-component ones in the following content. Instead of troubleshooting configuration errors, our method determines the likely correlated entries for each pair of components within a system. Given a system, the proposed method first narrows down the exploring scale to those frequently-set entries according to the crawled sample data. Then it generates a correlation score for each entry pair, which is calculated according to entry names, values and inferred entry types. Thirdly a set of filters are provided to determine a candidate set of the likely correlations. Finally a rank-ordered list of entry correlations is output so that system administrators can pay close attention to the small part of the entries and consult it to check system configuration systematically. Therefore, the misconfiguration induced system errors can be reduced proactively.

Based on the method, we implement a tool, named *Correlation Explorer*, and make evaluations with some real world systems. The results show that it is effective in determining a large portion of entry correlations in web application systems.

The main contributions of this paper are:

1) ***Method to determine entry correlations.*** We present a method to determine cross-component entry correlation-

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.95

s. Our method uses *data comparison*, *regular expression* based entry type inference and a set of filters to get the likely correlations. By focusing on these correlations, system administrators can effectively reduce their efforts to configure and check systems.

2) ***Tool implementation.*** We implement an extensible and configurable tool, *Correlation Explorer*, for effectively determination of the entry correlations.
3) ***In-depth analysis and evaluations of the experiment.*** We make experiments and evaluations on the accuracy of our method based on some metrics, *recall* and *precision*. We also analyze: 1) the effects of the filters to the final results, 2) the distribution of the entry correlations and 3) the root cause of false negatives.

The rest of this paper is structured as follows. In the next section we analyze the problem with some motivating examples. Section III gives a method overview, and section IV presents the details our method, including the data comparison based entry filtering, the entry type inference, the entry correlation score calculation and the filtering heuristics. Section V introduces the tool *Correlation Explorer*. Section VI presents the experiments and the evaluations. Section VII discusses advantages, limitations and threats to validity of our method. Finally, section VIII introduces the related work and section IX gives a conclusion, respectively.

II. Motivation

An entry correlation means that one configuration entry depends on the other entries, or the environment objects [10]. Entry correlations imply rich semantic information, which constrains what values the entries should be set. Figure 1(a) shows an entry correlation between MySQL and PHP, where *mysql.max_persistent* correlates to *max_connections*. The semantic of this correlation constrains the total volume of the persistent connections can be used in PHP (specified by *mysql.max_persistent*) must not be larger than that provided by MySQL (specified by *max_connections*). Once the constraint is violated, a too many connections error occurs. Another example is shown in Fig. 1(b), where the entry *jndi* of web service *LoginService* correlates to another entry *jndi-name* of the EJB component *LoginEJB*. The correlation semantic requires these two entries must have the same values, otherwise the login function of the application Adventure Builder[1] (Adventure for short) won't work.

Discovering these correlations is essential to ensure the correctness of system configuration. When deploying, migrating, and updating these systems, entry-error-induced system failures often occur due to the constraint violations. If the system entry correlations are provided proactively, the administrators can narrow down their configuration checking scales and focus on those ones correlating to one another. Therefore the human effort and the system failure risk can decrease dramatically.

However, keeping track of entry correlations is a thorny issue, especially for those cross-component ones (such as the examples in Fig. 1). The reasons lie in that:
1) The entries distribute in a large number of system components, while determining the correlations should check the configuration of multiple components at the same time;
2) The program analysis based approaches are impractical due to various programming languages are used [9][10];
3) The documents of the systems and the software may be inconsistent with the code and even missing [12], which making system configuration lacks supporting;
4) A system is usually constituted of various components, it is hard to find a single person who has knowledge spanning the various components and software [11].

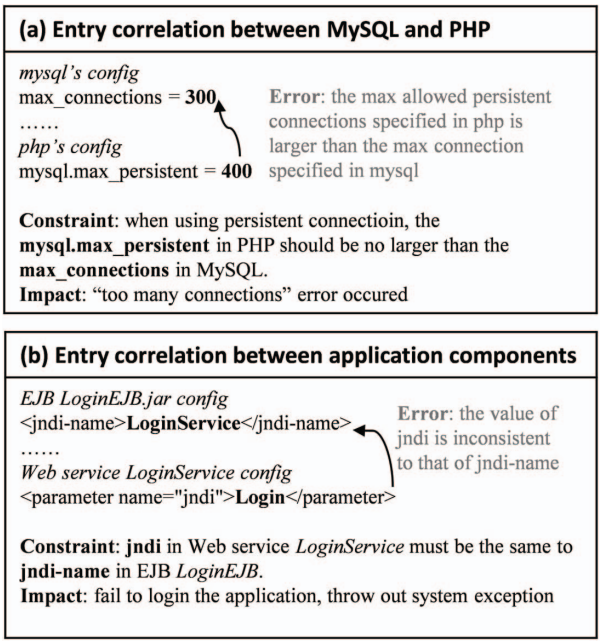

Fig. 1. Examples of entry correlations

To determine the cross-component entry correlations in a proactive way, we studied some representative open source software, including MySQL, PHP, Redis, etc. We present our observations on the characteristics of correlations, which motivates the proposed method.

A. If two components (include runtime environment and system software) depend on each other, it is very likely that there are cross-component entry correlations.

The component dependencies usually denote relationships of *resource provision and utilization*, *function call*, *data sharing and transmission* and so on. The example in Fig. 1(a) is a correlation about resource persistent connection, and the other example in Fig. 1(b) is a correlation involved in by a function call.

B. Configuration entries are with different types according to the syntactic patterns of their keys and values.

There are three common entry types, *numeric*, *Boolean*, and *string* [12], and they can be refined in a further step

[1]https://java.net/projects/adventurebuilder/pages/Home

according to the semantics implied by the syntactic patterns of entry keys and values. For instance, *max_connections = 300* is a numerical entry, and it can be inferred that this entry specifies a kind of resource (*connection*) according to its key. Particularly the string-type entries, their value syntactic patterns usually imply some semantics. For instance in MySQL, *datadir=/var/lib/mysql*, can be inferred that it may specify a file path.

C. Although many software has a large number of configuration entries, only a small part of them are set frequently.

Research work of Xu et al. [1] shows that only a small percentage (6.1%∼16.7%) of configuration entries are set by the majority (50+%) of users, while a significant percentage (up to 54.1%) of entries are with default values. We also make some studies on some open source software. We crawl 60 configuration files of Redis[2] from multiple projects in GitHub[3] and analyze the entry values. It is found that among 38 entries there are only 5 entries whose instance values are very different (13.2%), denoting these entry values change frequently. While the other entry instances in Redis only have no more than 5 different values (86.8%). The frequently-set entries of Redis are shown in Table I.

Based on above observations, we find that: 1) considering both keys and values of entries can capture the semantics much more precisely; 2) entries have their types, specifying *what features of what objects* they relate to. Motivated by the examples and the observations, we propose a method to determine entry correlations.

TABLE I
LIST OF ENTRIES CHANGE FREQUENTLY IN REDIS.

Entry	Occur times	Number of different values
port	56	21
pidfile	42	27
logfile	29	19
dbfilename	48	26
dir	40	20

This table lists the entries whose values are most different, meaning they change frequently. Note that the total occur times (2nd column) are different since not all entries occur in the sample configuration file crawled from GitHub. The 3rd column describes the total numbers of the different values of the entries.

III. METHOD OVERVIEW

Figure 2 shows an overview of the proposed method, which includes several steps as follows.

1) ***Entry filtering.*** We crawl configuration file instances of software from Internet (such as GitHub) as sample data. And then we filter out the entries hardly changed to narrow down the exploring scale and focus on those frequently-set ones.
2) ***Entry type inference.*** Given the remaining configuration entries, we infer their types by matching their keys and values against some regular expressions and keywords.
3) ***Correlation score calculation.*** Based on the keys, values and inferred types, we calculate *correlation scores* for each pair of entries belong to the different components.
4) ***Entry correlation determination.*** We use the *correlation scores* as metrics and propose some heuristics to filter out the false correlations. The filtering result is provided as the final determined correlations of the system.

The final rank-ordered list of the likely correlations is provided to users, who can consult it to check the system configuration.

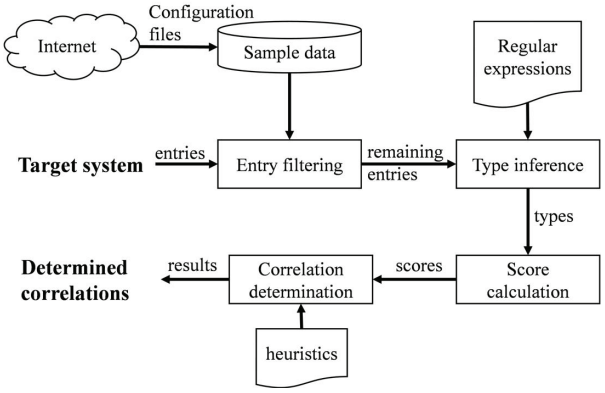

Fig. 2. Method overview

IV. METHOD DETAILS

A. Entry filtering

For a software, particularly those popular open source projects (e.g. Web server, database, messaging middleware, etc.), we crawl instances of its configuration files from well-known online forum and website (including ServerFault[4], Stack Overflow[5], Database Administrators[6] and GitHub).

With the large number of sample data, we make statistics of instance values for each entry. Two heuristic rules are proposed to get the frequently-set entries.

1) **[Multiple-value filter]** *For a configuration entry, if the values of its instances in the sample data are very different, the entry is a frequently-set one.*

For example, we gather 56 instances of entry port and there are 21 different values (shown in Table I). In fact, *port* is utilized to specify the monitoring port of Redis, and it is often configured with system specific value. The other examples in Table I are similar.

2) **[Different-value filter]** *If the entry instance value in the target system does not occur in the sample data, the entry is considered as a frequently-set one.*

For a specific system, some entries may be configured with specific values not occurring in sample data, such as those entries specify file path, user name and password.

[2]http://redis.io/
[3]https://github.com/
[4]http://serverfault.com/
[5]http://stackoverflow.com/
[6]http://dba.stackexchange.com/

B. Entry type inference

Most configuration entries fall into some common types, including *Numeric*, *Boolean* and *String*, and each type may further have several specific subtypes. For example, *pidfile* and *bind* are both String entries of Redis, and *pidfile* is a *file path* while *bind* denotes an *IP address*.

Inferring likely types for each configuration entry helps to get its semantics. In our method, given an entry, its value is matched against each regular expression. There are several rules should be followed.

1. If the returned type is a common one (particularly *Numeric* and *Boolean*), there is little semantic information with the entry value. For example, *port* and *timeout* are two *Numeric* entries in Redis, and there is no information about what properties/features they specify if just according to their values. In such a case, the entry name must be analyzed with the other regular expressions and keywords.

2. If a common type and a subtype are both inferred, the more specific one is used. For example, *IP* is a specific type of *String*, if an entry is an *IP*, it is also a *String* meanwhile.

3. Multiple subtypes may be inferred when leveraging the keywords, all of them are kept as the likely ones. For example, *jdbc.pool.maxIdle* is a *Numeric* entry specifies the upper bound of database connection resource, and it can be inferred from the entry name that *Size* and *Resource* are likely types.

Finally, the inferred types of each entry are stored in an entry type vector $T_{entry} = (t_1, t_2, \ldots, t_m)$, where each $t_i (1 \leq i \leq m)$ represents an entry type. The value of t_i is 1 only if the entry belongs to this type, otherwise is 0. It worth to note that, 1) the length of T_{entry} is fixed, equal to the number of defined entry types, and 2) there may be multiple elements in T_{entry} whose values are 1 since an entry may be inferred with several types.

Table II lists some representative entry types. Note that the entry type definitions are extensible and adaptable with new type definitions, regular expressions and keywords.

C. Correlation score calculation

This paper proposes *consistency correlation score* and *type correlation score* as the metrics to determine whether two entries are correlated or not. Given an entry pair, these two metrics are calculated respectively.

1) Consistency correlation: Consistency correlation is the most common one among entry correlations, which specifies that a pair of entries must be with the same values (or one value is the substring of the other). *Consistency correlation score* is proposed to determine such type of entry correlations.

For two entries, their score is calculated based on their keys, values and types since that, if two entries are consistency-correlated, then, 1) their values are the same, or very similar at least; 2) they have similar semantics and thus their keys and types are also similar.

Given a pair of entries, $e_i =< k_i, v_i >$ and $e_j =< k_j, v_j >$, whose type vectors are T_i, and T_j, we calculate the similarities between their keys, values and types, and then the average of these similarities are taken as the *consistency correlation score*. The algorithm are shown in formula 1- 4, where,

- Similarities of keys and values $sim(k_i, k_j)$, $sim(v_i, v_j)$ are calculated based on *longest common substring* algorithm. In formula 1 and 2, $maxlong(k_i, k_j)$ and $maxlong(v_i, v_j)$ denote the lengths of the longer strings of the keys and the values respectively.
- Type similarity $sim(T_i, T_j)$ is calculated based on Cosine vector, which is shown in formula 3.
- Finally, formula 4 takes the average of the all above results as *consistency correlation score* $Consis(e_i, e_j)$, whose value is in the range [0, 1]. The higher the score is, the more likely the entries are consistent.

$$sim(k_i, k_j) = \frac{mostCommonSbuStr(k_i, k_j)}{maxlong(k_i, k_j)} \tag{1}$$

$$sim(v_i, v_j) = \frac{mostCommonSbuStr(v_i, v_j)}{maxlong(v_i, v_j)} \tag{2}$$

$$sim(T_i, T_j) = \frac{T_i \cdot T_j}{\|T_i\| \|T_j\|} \tag{3}$$

$$Consis(e_i, e_j) = \frac{sim(k_i, k_j) + sim(v_i, v_j) + sim(T_i, T_j)}{3} \tag{4}$$

Note that currently we just think of the influences of these three factors to the final result are equal, and some weighting based algorithms will be studied in future work.

2) Type correlation: Type correlation implies that if one entry changes its value, the other should be changed to a corresponding value instead of the same. It is observed that the correlations can be inferred from the entry types. For example, a *RESOURCE* type entry may relate to the other entry of *SIZE* type, meaning that the latter specifies the volume of the resource denoted by the former. Another example is *URL* and *IP*, these two types usually correlate to each other.

We define some common relations between the entry types, and each of them implies the semantics between two kinds of entities. For example, $< FILEPATH, USER >$ implies that two entries of these types are correlated due to the authority, and $< HOST, IP >$ denotes that a *HOST* entry has this *IP* address. Note that the pre-defined type correlations are also extensible.

With the entry types, *type correlation score* $Correl(e_i, e_j)$, the metric to determine such type of entry correlations, is calculated based on formula 5- 7, where $correlNum(T_i, T_j)$ denotes the number of correlated types between the type vectors of two entries. The final value is normalized into range [0, 1).

$$Correl(e_i, e_j) = \frac{2 \times \arctan CorrelNum(T_i, T_j)}{\pi} \tag{5}$$

$$CorrelNum(T_i, T_j) = \sum_{i=1}^{n} \sum_{j=1}^{n} CorreVal(t_i, t_j) \tag{6}$$

TABLE II
ENTRY TYPE DEFINITION

Common Type	Subtype	Syntactic/keyword	Semantic
String	URL	[[a-zA-z]+\\:]+//[^\\s]*	Internet URL
	IP	(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])+(:(0|[1-9][0-9]{0,3}|[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5]))?	IP address
	FILE PATH	/?([^/]+/)+[^/]*	File path
	DATE TIME	[0-9]{4}-(((0[13578]|(10|12))-(0[1-9]|[1-2][0-9]|3[0-1]))|(02-(0[1-9]|[1-2][0-9]))|((0[469]|11)-(0[1-9]|[1-2][0-9]|30)))	Date time
	EMAIL	^\\w+([-+.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*$	E-mail address
	RESOURCE	buffer|cache|pool|thread|disk|cpu|memory|queue|message|table|connection|block|packet|file|socket|session|port|host|jdbc|dir|directory	Common resources
	HOST	server|host	Server
	DATABASE	db|jdbc|database	Data base related
	ID	identity|identifier|id|name|uri|jndi	Identity
	USER	user|usr	User
	GROUP	group	Group
	PASSWORD	password|pwd|pass	Password
Boolean		[tT][rR][uU][eE]|[fF][aA][lL][sS][eE]|[oO][nN]|[oO][fF]{2}|[yYnN]|[yY][eE][sS]|[nN][oO]	Switch of operation or feature
	MODE	mode|enable|disable	
Numeric		[+-]?\\d+[.]?\\d+	Number denoting size, counter, etc.
	PORT	port|listen	Port
	TIME	time|interval|day|month|year|hour|minute|second|millisecond	Date and time related
	SIZE	size|number|length|max|min|threshold	Size
	COUNT	count	counter

Some representative types are shown. Type inference is made according to those expressions and keywords, or both. For example, if an entry whose value matches the expression of Boolean and whose name contains the keywords of subtype MODE, the entry is with MODE type.

$$CorreVal(t_i, t_j) = \begin{cases} 1 & t_i, t_j \text{ are correlated} \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

Note that *consistency correlation* and *type correlation* are orthotropic. That is, for an entry pair e_i, e_j, if $Consis(e_i, e_j)$ is big then $Correl(e_i, e_j)$ is definitively small and vice versa. As a result, there would be no more than one correlation between an entry pair.

D. Entry correlation determination

With $Consis(e_i, e_j)$ and $Correl(e_i, e_j)$ for each pair of entries, some heuristics are proposed to filter out those false results, including:

Threshold-based filter. Threshold-based filter is the basic one in our method. Two thresholds H_c, H_t are set up, the former is for *entry consistency correlation* and the latter is for *entry type correlation*. A correlation is very likely true when its correlation score exceeds the threshold (H_c or H_t).

Redundant filter. Intuitively, it is impossible that an entry e_i of a component relates to many entries of the other one. Therefore, if an entry occurs much more than three times in a set of entry pairs of two components, we take the first three with highest correlation scores (*consistency correlation score* or *type correlation score*) as the likely correlated ones.

Top-k filter. For the entry correlations of two components, we rank them in descending order by their correlation scores, and get two rank-ordered lists, one for *consistency correlations* and the other for *type correlations*. We just take both of the first *k* ones in these two lists as the candidates.

The filtering heuristics are configurable, user can apply one or more of them. Many false correlations will be filtered out, and the union of the two lists (the *entry consistency correlations* and the *entry type correlations*) is taken as the final candidates.

V. IMPLEMENTATION

Based on the proposed method we implement a tool named *Correlation Explorer*, which is constituted of several modules, including: *data crawler*, *type inferior*, *calculator* and *result filter*.

Correlation Explorer provides interfaces for users to upload the system packages (particularly the configuration files) and declare the component dependencies. It also provides interface for user to declare some basic environment properties of those machines, such as CPU, memory size, disk space, and some other information (e.g. IP address, JAVA HOME, etc.). Users also can add their own specific properties in the form of key-value pairs.

Data crawler collects configuration files from some well-known online forums and communities. For each software, the all instances of the configuration file are analyzed to find those frequently-set entries. Note that data crawling is a one-time task and the result can be used repeatedly.

For each component (particularly the system software) in the system, its entries are narrowed down to those frequently-set ones. *Type inferior* infers the types of such entries.

Calculator generates *consistency correlation score* and *type correlation score* for each entry pair. The entries in a pair must satisfy two conditions: 1) the two entries are of the different components, 2) there is a dependency between the components.

Result filter outputs the final candidate set of entry correlations. The *threshold-based* heuristic is first applied. The thresholds H_c, H_t are both configurable, and their values are set with 0.6 as defaults. Then some other filters can be applied further, including *redundant filter*, *top-k filter* or both. The default value of factor *k* in *top-k filter* is 5.

The final output of *Correlation Explorer* is a list of the most likely entry correlations. When an administrator makes a

system deployment or migration, the list, as a reference, helps to make systematical checking to avoid constraint violations.

VI. Experiment and evaluation

We evaluate the accuracy of our method by answering the following questions:

RQ1. How accurate is *Correlation Explorer* in determining cross-component entry correlations?

RQ2. How dose *Correlation Explorer's* ability compare to a method without using entry type?

RQ3. What are the effects to final results when *Correlation Explorer* uses the different heuristic filters?

Besides answering the above questions, we also analyze:

1) The root cause of the false negatives in our experiment.
2) The distribution of the entry correlations in systems.

TABLE III
Component of target systems

System	component	version	number	function
Cloudshare	Cstenant.war	-	1	Portal for multiple tenants
	Cloudshare.war	-	2	Front end for users
	Service.war	-	2	underlying common services
	Index.war	-	1	searching & indexing
	Messageing.war	-	1	Instant messaging service
	Apache Tomcat	7.0	6	Runtime environment
	Nginx	1.6.2	1	Http reverse and proxy agent
	Redis	2.8.9	1	Caching service
	ActiveMQ	5.8	1	Message queue middleware
	JDK	1.7	4	Java virtual machine
	cs_global	-	2	System information database
	cs_tenant_default	-	2	User document database
	cs_msg_tenant_default	-	2	User message database
	MySQL	5.6	2	Database server, one master, one slave
Adventure	Login EJB		1	EJBs realize underlying business logic and database accessing
	Mountain EJB		1	
	Island EJB		1	
	Transport EJB		1	
	Hotel EJB		1	
	Bank EJB		1	
	Login Service		1	Web services providing multiple services
	Mountain Service		1	
	Island Service		1	
	Transport Service		1	
	Hotel Service		1	
	Bank Service		1	
	Travel Plan Process		1	WS-BPEL process of building a travel plan
	Adventure.war		1	Website for users
	JDK	1.7	2	Java virtual machine
	Tomcat	7.0	1	Web application runtime environment
	Web Service Container		1	Web service runtime environment
	WS-BPEL Container		1	WS-BPEL runtime environment
	EJB Container		1	EJB runtime environement
	travelplan		1	Application database
	MySQL	5.6	1	Database server

A. Target systems and experiment setup

We evaluate *Correlation Explorer* with two systems, one is a service-based application Adventure and the other is a real-world cloud-based storage service *Cloudshare*[7].

Adventure, providing touring arrangement services, is a Java blueprint application reference originally, and we refactor it based on SOA (Service Oriented Architecture) with web services, WS-BPEL process, Java Enterprise Beans (EJB) and the other components. In total, all of the 22 components (including application components and system software, such as Tomcat and MySQL) distribute on three servers.

Cloudshare provides multiple services, such as file storage and sharing, collaborative working and instant messaging. The production system is hosted on Aliyun[8], and we setup a small scale experiment environment. The all 28 components distribute on five servers with Intel(R) Core(TM) i7 3.4 GHz CPU, 4GB RAM, and CentOS 6.5 operating system.

The components of these two systems are listed in Table III, and their deployment topologies are shown in Fig. 6 and Fig. 7 respectively, where the lines between components and the hierarchical structure within each server represent the dependencies between components.

B. Evaluation procedure and results

The evaluation procedure is as follows.

1) Step 1 Entry filtering: The filtering is useful to those open source system software since there is much more sample data on the Internet. On the contrary, it is useless to those application specific components and the vendor specific software. It can be seen in Table IV that the result of Cloudshare is better than that of Adventure, which because that there is much more open source software used in the former and a large part of the entries in the software are filtered out. For example, in Cloudshare we filter out over 50% and over 80% entries for Nginx and Redis respectively.

TABLE IV
Entry filtering results

	Initial number	Remaining number
Cloudshare	886+	107
Adventure	457+	202

Note that 886+ and 457+ are not the accurate numbers of entries in these 2 system. We don't count some complex ones and never used ones.

TABLE V
Entry type distribution of Cloudshare

Common Type	Subtype	Number
String		11
	URL	23
	IP	22
	FILE PATH	31
	DATE TIME	0
	EMAIL	3
	RESOURCE	5
	HOST	1
	DATABASE	22
	ID	0
	USER	12
	GROUP	0
	PASSWORD	18
Boolean		5
	MODE	1
Numeric		9
	PORT	28
	TIME	4
	SIZE	17
	COUNT	0

Note that the sum is bigger than the total number of filtered entries since there are some entries have more than one types.

2) Step 2 Entry type inference: Table V shows the type inference results. It is found that most of the entry types are inferred correctly according to our manual review. Note that

[7]http://123.56.40.165:8280/cloudshare/

[8]http://www.aliyun.com

in Table V the sum is bigger than the number of remaining entries (in Table IV) since there are some entries have more than one type. For example, $db.default.user = app$, used to set database username, is inferred with types *DATABASE* and *USER* according to the keywords in its key.

It is found that there are only eleven false types in Cloudshare, accounting for 5.45% (11/202). For example, $mail.username = noreply@cloudshare.im$ is inferred with EMAIL with regular expression. However, this entry is a user name actually, which is set with an e-mail address. Another example, $server\_id = 1$ is considered as a NUMERIC entry, however the value is used as a server identity. The type inference result of Adventure is similar, there are 17 false ones, accounting for 8.02% (17/212).

3) Step 3 Correlation score calculation and result filtering: To answer questions **RQ1** and **RQ3**, we make accuracy evaluations based on two metrics, *precision* $A = C_f/T$ and *recall* $R = C_f/C_t$, where C_t represents the total real correlations we manually find and C_f denotes the real correlations discovered by *Correlation Explorer*; the total number of candidates found by *Correlation Explorer* is represented as *T*.

For the two target systems, we manually find the real correlations as baselines (91 in Cloudshare, 84 in Adventure) with the help of the system developers and administrators. The method finds 65 true correlations in Cloudshare and 69 in Adventure. Therefore the recalls are $71.43\%(65/91)$ and $82.14\%(69/84)$, respectively.

As to the precision, the filters have the different effects to the final results, which are shown in Fig. 3, where 1) *T* means basic filter *threshold*, 2) *T+R* means *threshold* and *redundant filter*, 3) *T+Top K* denotes *threshold* and the *top k filter*, 4) *T+Top K+R* means all these 3 filters.

In Fig. 3(a), the precision of just using threshold (T) is the lowest (65/140=46.43%) since many false positives are involved in. Then with the other different filters and their combinations, a part of the false candidates are removed, and the precision increases, the highest of which reaches to 67.01% (65/97).

The results of Adventure are similar, which are shown in Fig. 3(b), the lowest precision is 52.27% (69/132) when just using threshold filter (T), then increases slightly (53.91%) since some false correlations are removed by redundant filter (R). The precisions of the last two experiments are the same (78.41%), which because most false positives are filtered out by Top K and in this case there are no redundant candidates left. Therefore, when applying the redundant filter further, there is no effects to the final results.

We also evaluate the method accuracy by analyzing the rank-ordered lists of final results. Note that the method provides a list per component pair. It is shown that most of the true entry correlations rank at the top.

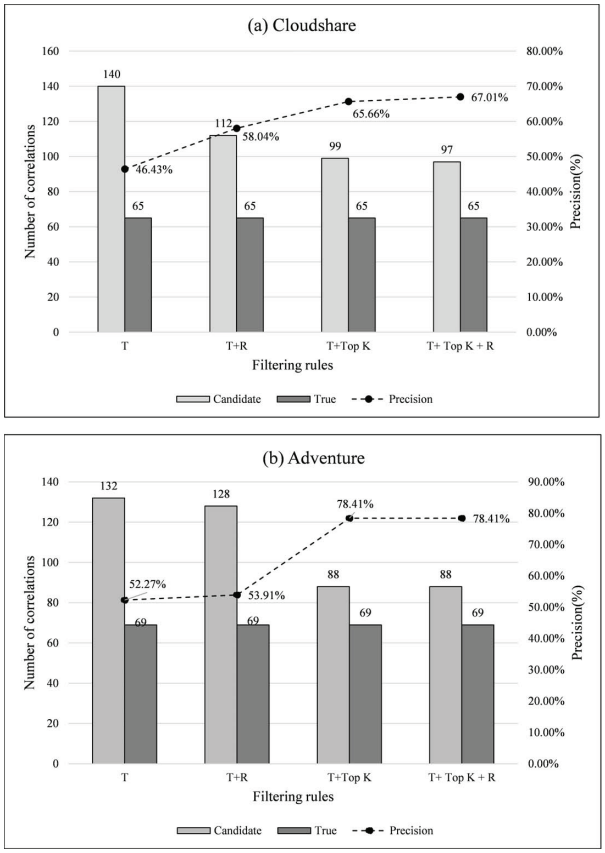

Fig. 3. Final results of the target systems

It is shown in Table VI that, in these two systems, all of the true correlations determined by the method rank top 5, and the numbers of those ranking 1st are 45 and 54, respectively, accounting for 69.23% and 78.26%.

TABLE VI
RANK ORDER OF THE TRUE ENTRY CORRELATIONS

Rank	1st	2nd	3rd	4th	5th	total
Cloudshare	45	11	7	2	0	65
Adventure	54	3	6	6	0	69

C. Method comparison

Ramachandran [11] proposed an approach to determine the configuration parameter dependencies. This approach determines two parameters are likely-dependent to each other only if: 1) their values are the same strings, or 2) one value is the sub-string of the other one.

To answer question **RQ2**, in our experiment we use this approach to determine the entry correlations of the two systems and compare the results with those of our method. The results are shown in Table VII.

It can be found that the recalls of these two methods are almost the same, while their accuracies are very different.

On one hand, many false positives are involved in since the compared method only compares entry values, and an example

is shown in Fig. 4(a). On the other hand, there are also some false negatives. Fig. 4(b) is an example, where two entries are type-correlated, but they are missed due to their dissimilar values.

TABLE VII
RESULT COMPARISON

	Candidate	True	Recall	Precision
Cloudshare	97	65	71.43%	67.01%
Cloudshare-N	227	65	71.43%	28.63%
Adventure	88	69	82.14%	78.41%
Adventure-N	203	72	85.71%	35.47%

Cloudshare and Adventure denote the results of our method, and Cloudshare-N and Adventure-N present the results of just comparing entry values.

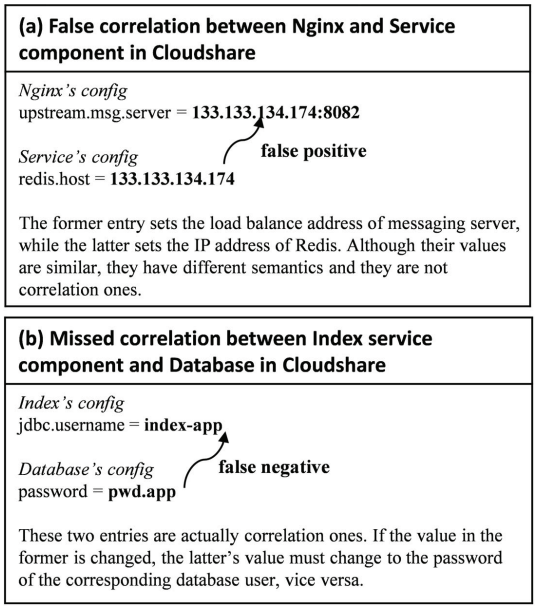

Fig. 4. Examples of false positive and false negative of the compared method

As a result, the accuracy of compared method is much lower than that of ours. Note that there are also some filters proposed in the compared approach, such as *Different Valued*, *Infrequent Valued* and *Normalized Google Distance*. However, these filters are impracticable with some limitations.

1) *Different Valued* and *Infrequent Valued* require that there are multiple instances of a component in a system. However, many components in our experiment just have single instance;
2) *Normalized Google Distance* leverages the occurrence of two entries in Google searching results as filtering metric. However, the application components are specific ones, and it is difficult to find occurrences of entry pairs if there is at least one entry that is of application component.

D. False negatives

We look into the system configurations and find that most of the missed correlations involve more than two entries, two cases are shown in Fig. 5.

Figure 5(a) shows a *one-to-multiple* correlations, where the entry of Nginx relates to the IP address of server node 2 and the port of Tomcat. The correlation in Fig. 5(b) is more complicated, involving 4 entries, where *jdbc.url* relates to the other three entries of MySQL and server node.

Our method just focuses on those *one-to-one* correlations, it can determine the correlations of *upstream.service.server* and *IP* shown in Fig. 5(a), while fails to find the other *one-to-multiple* ones. The reason lies in that the very different strings make the final correlations scores low.

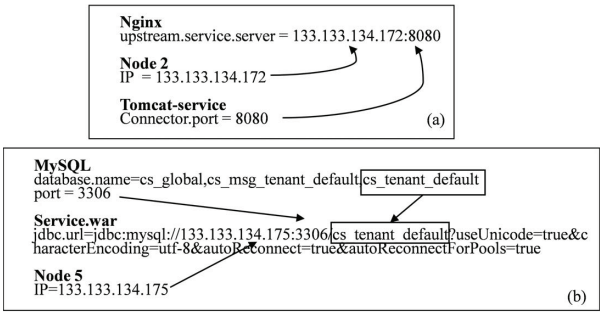

Fig. 5. Two examples of the missed entry correlations

E. The distribution of the correlations

We classify the components of a system into two categories:

1) Application components (App.) provide business related functions and services, such as the all web modules (in the form of *.war* package) in Cloudshare and the web services, the EJBs and the WS-BPEL process in Adventure;

2) Common components (Com.) provide common services to support those application components, such as Nginx, Redis, Tomcat, ActiveMQ and MySQL in these two systems.

The dependencies between components are classified into three categories: 1) App. to App., 2) App. to Com. and 3) Com. to Com. We group the correlations based on the three types of component dependencies, and the distributions are shown in Table VIII. We find most correlations exist between App. and App. (53.85%, 42.86%), or App. and Com. (32.97%, 52.38%).

On one hand, the application components depend on the services provided by those system software, which leads to many entry correlations. As an example, *service.war* depends on the caching service of *Redis*, and there are three entry correlations between them, specifying *port*, *IP address* and *password*. On the other hand, the data communications and the function dependencies between application components also generate many App.-App. entry correlations. For example, there are four correlations between web service *HotelService* and EJB *HotelEJB*, specifying *jndi name*, *jndi provider URL* and other parameters. Therefore most entry correlations are

App.-involving ones, which also can be taken as the proof of the limitations of the filters proposed in the compared method.

TABLE VIII
THE DISTRIBUTION OF ENTRY CORRELATIONS

	Cloudshare		Adventure	
	number	pct.	number	pct.
App.-App.	49	53.85%	36	42.86%
App.-Com.	30	32.97%	44	52.38%
Com.-Com.	12	13.19%	4	4.76%
Total	91		84	

VII. DISCUSSION

A. Advantages

Compare to the other similar work, the advantages of our method lie in:

Light weight, requires a little of domain knowledge. Although Encore [7] addresses various types of entry correlations, it is heavy-weight in semantic verification, involving many domain knowledge. Our method is light weight, only requiring the predefined entry types and the correlations at entry type level. *Correlation Explorer* is program language independent since it would not look in-depth into the source code of system. It just focuses on the entries in configuration files, making the method has general applicability.

Highly configurable and extensible. *Correlation Explorer* is highly configurable with 1) the thresholds and 2) applying strategies of the candidate filters. The extensibilities lie in 1) the entry type definitions and 2) the type correlations. New entry types can be added in just by specifying its regular expressions, and their correlations can be declared.

B. Limitations

This method cannot find all of the entry correlations of a system.

1) *Correlation Explorer* only focuses on the entry correlations between components. However, there are also entry correlations within a single component and such correlations are determined with program analysis based method [8].

2) Not all configuration entries are in files, some may be hard coded as constant and variables. *Correlation Explorer* does not consider these entries. In such situations, the program analysis based methods are applicable, and it is complementary to our work.

The entry type definitions, the type correlations and the filtering heuristics should be improved further.

1) We only declare some coarse-grained common type correlations in this method, which affects the accuracy of determining entry type correlations. We should refine them in our future work.

2) It is impossible to declare keywords exhaustively, some NLP (Nature Language Processing) technique [13] should be employed.

C. Threats to validity

The system in our experiment may not be representative. Although Cloudshare is a real world service-based system and many popular open source software (e.g. Tomcat, Nginx, MySQL, etc.) are used, it only presents a kind of multi-tiered and distributed systems. Likewise, those application components in refactored Adventure are specific and might not be representative. More experiments should be conducted with various systems to evaluate the effectiveness further.

VIII. RELATED WORK

The existing approaches to tackling configuration errors fall into several primary fields: 1) troubleshooting misconfigurations, 2) easing system configurations, and 3) hardening systems against configuration errors.

Troubleshooting misconfigurations. Some troubleshooting approaches are based on program analysis, including static analysis, dynamic analysis or both. A static dataflow analysis based approach [14] is proposed to pre-compute possible configuration errors. ConfAid [15] uses dynamic taint tracking (and short-distance speculative execution) to find error root causes. ConfDiagnoser [16] identifies the root cause of a configuration error by using static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options. CODE [17] is a statistical analysis based tool, automatically detecting software configuration errors by identifying invariant configuration access rules. Signature based approaches [18] [19] diagnose configuration errors by extracting a signature of the program behavior associated with a particular misconfiguration. Replay-based approaches, including Chronus [20], AutoBash [21] and Triage [22], fix configuration errors by trying possible configuration changes in a sandbox, without affecting the rest of the system. Comparison-based approaches, including Strider [23] and PeerPressure [24], compare misconfigurations with correct ones to find out possible root causes, and they also can fix the errors according to the correct configurations.

Easing system configurations. Some work tries to lower the misconfiguration rate by 1) providing automatic deployment and configuration methods, 2) minimizing the number of configuration entries and finding out those frequently-set ones, 3) implementing user-friendly configuration constraints. The study of Xu et al. [1] reveals a series of interesting findings to motivate software architects and developers to be more cautious and disciplined in configuration design. ConfValley [25] is a generic framework to make configuration validation easy, which consists of a declarative language, an inference engine and a checker.

Hardening systems against configuration errors. Some work addresses the problem of how to harden system against configuration errors by denying erroneous settings and print useful log messages to pinpoint errors. ConfErr [26] is a tool for testing and quantifying the resilience of software systems to human-induced configuration errors. Xiong [27] proposes range fix specifying the options to change and the ranges of values for these options. SPEX [8] is a white-box

tool that generates configuration errors based on the inference of the constraints of configuration entries. ConfDiagDetector [28] detects inadequate diagnostic messages for configuration errors issued by a configurable system based on the techniques of configuration mutation and NLP text analysis.

The most related work to ours are those determining configuration entry correlation and inferring entry type.

Inferring entry type. Based on static program analysis, Rabkin's work [12] infers entry types by looking for patterns in how programs use entry values. It classifies entries into 4 types: numeric, mode, identifier and others. SPEX [8] infers basic type of an entry from source code and determine the semantic type by searching some patterns along a parameter's entire data-flow path. Encore [7] is capable of referring entry type in a two-step process that leverages both syntactic patterns of data values and the environment information of the system. Similar to Rabkin's work and Encore, Correlation Explorer infers entry type with a set of predefined regular expressions. However, it is found that an entry may be of multiple types, and then we propose some rules to generate a type vector for each entry.

Determining entry correlation. There are two kinds of methods to determine the correlations, one focuses on the correlations of a single software and makes determinations based on program analysis, and the other try to find those correlations across components. SPEX [8] proposes control dependency and value relationship between configuration entries. The control dependencies are inferred by analyzing control flow of entries, and the value relationships are inferred by looking for comparison statements in entries' usage. Research work [11] determines configuration entry dependencies in a probabilistic sense. It considers the similarities between entry values and filters the results by querying the Internet to provide an estimate of a dependency between a pair of configuration entries. Encore [7] specifies entry correlations in the form of templates by applying machine learning technique with a large volume of training data.

Unlike SPEX and Encore, *Correlation Explorer* determines entry correlations only based on configuration files instead of analyzing source code. As a result, it is program language independent, and it has better performance. *Correlation Explorer* is similar to the research [11]. However, our method analyzes keys, values and entry types comprehensively. Furthermore, our filtering heuristics don't require additional information. The comparison results in Section VI-C show that *Correlation Explorer* is more accuracy and practical.

IX. Conclusion and future work

In web application systems, entry correlations can result in configuration errors, and manually determine the correlations is tedious and time-consuming, requiring domain knowledge spanning multiple software.

We propose a method to automate the process of determining the entry correlations between components. This method calculates correlation scores for entry pairs and effectively filters out most of the false ones. We implement a tool *Correlation Explorer* based on the method and evaluate its accuracy.

We will address the following topics in the future:

Make sufficient experiments and evaluations. Currently we make a preliminary experiment in this paper, and we plan to evaluate our work sufficiently with some other representative systems.

Method improvement. We will improve this method in several aspects: 1) propose more effective filters; 3) leverage some NLP techniques to get entry semantic more accurately.

Take one-to-many entry correlations into account. Our method is incapable of discovering the correlations involve more than two entries. Furthermore, there may exist transitive correlations between several components [8]. In future work, we will consider these correlations and improve the method to find them out.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China under Grant No. 61402453, the National High Technology Research and Development Program of China under Grant No. 2013AA041301, the National Key Basic Research Program of China under Grant No. 2015CB352201.

References

[1] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 307–319.

[2] Z. Dong, A. Andrzejak, and K. Shao, "Practical and accurate pinpointing of configuration errors using static analysis," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 171–180.

[3] Y. Svrdlik, "Microsoft: 10 things you can do to improve your datacenters," http://cc4.co/USWU, 2012.

[4] A. Team, "Summary of the amazon ec2 and amazon rds service disruption in the us east region," http://aws.amazon.com/message/65648/, 2011.

[5] R. Johnson, "More details on todays outage," http://cc4.co/CGL, 2010.

[6] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 159–172.

[7] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014.

[8] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 244–259.

[9] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 215–224.

[10] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 70, 2015.

[11] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 169–178.

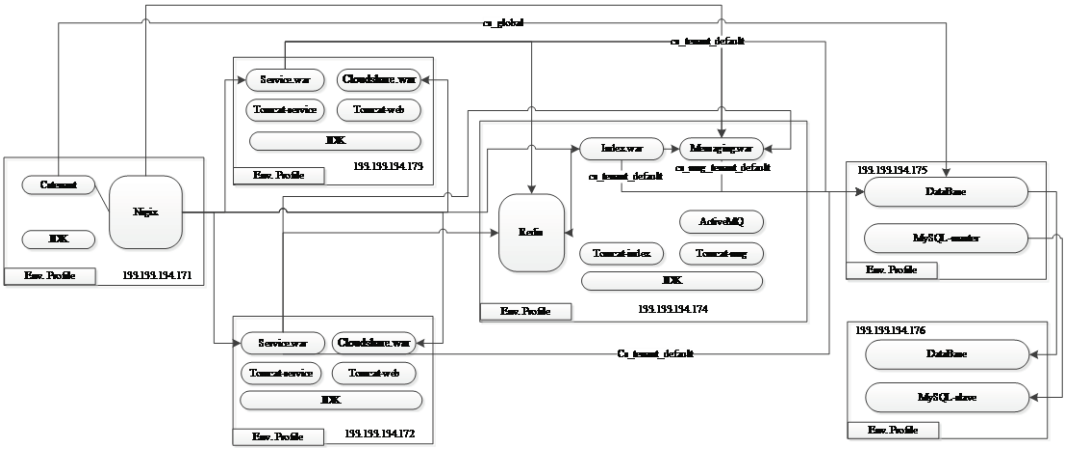

Fig. 6. Deployment topology of Cloudshare

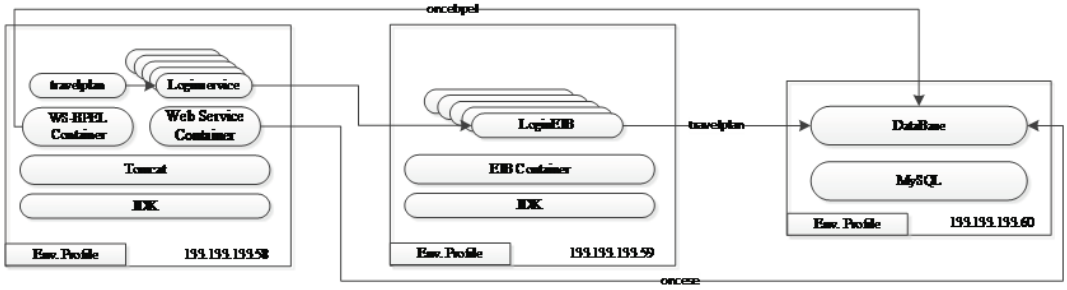

Fig. 7. Deployment topology of Adventure

[12] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 131–140.

[13] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *AAAI*, vol. 6, 2006, pp. 775–780.

[14] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 193–202.

[15] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, 2010, pp. 237–250.

[16] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 312–321.

[17] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association, 2011, pp. 28–28.

[18] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 375–388, 2006.

[19] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang, "Automatic software fault diagnosis by exploiting application signatures." in *LISA*, vol. 8, 2008, pp. 23–39.

[20] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack." in *OSDI*, vol. 4, 2004, pp. 6–6.

[21] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 237–250.

[22] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 131–144.

[23] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," *Science of Computer Programming*, vol. 53, no. 2, pp. 143–164, 2004.

[24] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure." in *OSDI*, vol. 4, 2004, pp. 245–257.

[25] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: a systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 19.

[26] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 157–166.

[27] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 58–68.

[28] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 12–23.