amulog: A General Log Analysis Framework for Diverse Template Generation Methods

Satoru Kobayashi
NII
sat@nii.ac.jp

Yuya Yamashiro
The University of Tokyo
yuya@hongo.wide.ad.jp

Kazuki Otomo
The University of Tokyo
otomo@hongo.wide.ad.jp

Kensuke Fukuda
NII/Sokendai
kensuke@nii.ac.jp

***Abstract*—One of the ways to analyze unstructured log messages from large-scale IT systems is to classify log messages with log templates generated by template generation methods. However, there is currently no shared knowledge pertained to the comparison and practical use of log template generation methods because they are implemented on the basis of diverse environments. To this end, we design and implement amulog, a general log analysis framework for diverse log template generation methods. There are three key functions of amulog: (1) parsing log messages into headers and segmented messages, (2) classifying the log messages using a scalable template-matching method, and (3) storing the structured data in a database. This framework helps us easily utilize time-series data corresponding to the log templates for further analysis. We evaluate amulog with a log dataset collected from a nation-wide academic network and demonstrate that it works in a reasonable amount of time even with over 100,000 log template candidates.**

***Index Terms*—Log analysis, network logs, framework**

I. Introduction

System and network operators need to maintain high availability of IT infrastructures by means of efficient troubleshooting. Log data is one of the most effective data sources for this because, unlike other measurable data, it provides contextual information of system behaviors as literal explanations. However, log data from large-scale information systems is too large to investigate manually. For example, SINET5 [1], a research and education network in Japan, reports about 150,000 log lines in a single day. We need automated analysis methods and tools for analyzing such large-scale log data.

Automated analysis of log data is also difficult because log messages include unstructured statements. A major approach to analyzing log data is generating log templates, a format of the statements. Log messages belonging to a log template contain information on a common system behavior, so template generation is an effective way to classify log messages with their behaviors. In contrast to full-text search [2]–[4], which is another major approach used for keyword-based analysis, the template-based approach is suitable for time-series-based quantitative analysis such as anomaly detection [5]–[9] and root cause analysis [10]–[12].

Many log template generation methods have been proposed in past literature [13], [14]. These methods are based on diverse approaches such as source code analysis [15]–[17], clustering [18]–[24], prefix-tree approaches [6], [25], [26], and machine learning [27], [28]. As log data plays an important role in multiple IT fields, the template generation methods also have diverse assumptions, which prevent operators from determining whether the methods match their use in a consistent manner. Therefore, we need a general framework for automated log analyses that does not depend on a specific log template generation method.

There are three requirements for a general log analysis framework. First, the framework needs to preprocess the messages uniformly. Most of the log template generation methods use segmented log messages (i.e., a sequence of words), but many practical log messages cannot be segmented by simple methods (details in § II-A). A suitable log segmentation rule depends on the dataset rather than the template generation method, so we need log segmentation functions in the framework. Second, the framework needs to match log messages with existing log templates. A log analysis framework should consider various use cases, such as accurate log templates being partially available without template generation or manual template modification required for precise analysis. Log template matching enables the framework to remap log templates and their instances to satisfy these requirements. Third, the framework needs to provide the parsed and classified data in an appropriate style for further analysis.

In this paper, we propose amulog, a general framework for template-based log analysis to satisfy these requirements. Figure 1 shows the schematic system architecture of amulog. The key functions provided by amulog are threefold: (A) parsing log messages into headers and segmented statements with a rule-based parser log2seq [29], (B) classifying log messages with log templates by a tree-based scalable algorithm, and (C) storing the parsed data in a database that enables search and aggregation for further analysis. Two processing modes are available on amulog: online processing for real-time analysis and offline processing for hindsight analysis. We implement amulog as open-source software [30].

We discuss the design of amulog to show its effectiveness in practical log analysis. We evaluate amulog in terms of log parsing validity and log template matching performance with a real log data from SINET4 [31], a nation-wide academic network. We confirm that the proposed tree-based algorithm in amulog matches log messages with given log templates in a reasonable amount of time even if the number of given

978-3-903176-31-7 © 2020 IFIP

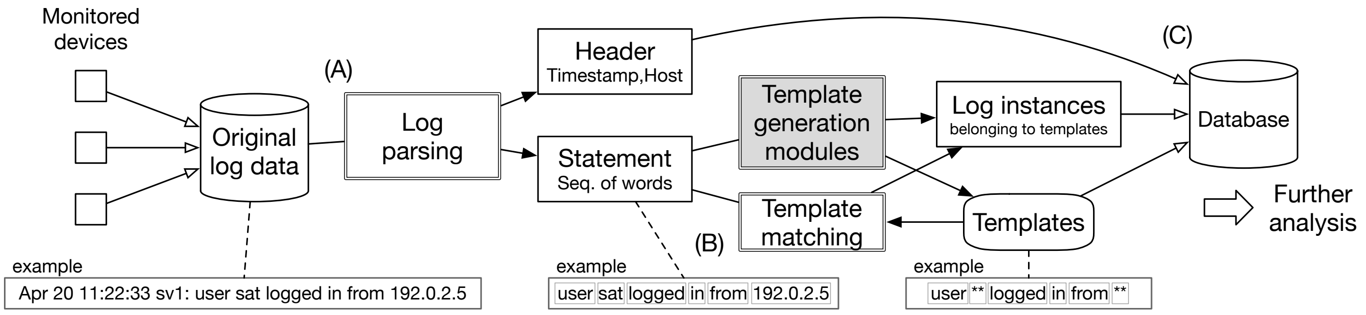


Fig. 1. Overview of proposed log analysis framework.

templates is more than 10^5.

The contributions of this paper are as follows. (1) We find and solve practical issues for a general log analysis framework. (2) We implemented the general log analysis framework and evaluate its performance.

II. AMULOG

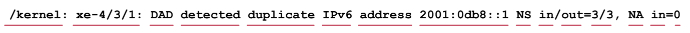
Figure 1 illustrates the system architecture of amulog. It has three main components: (A) log parsing, (B) template matching, and (C) database storing. Amulog assumes line-based log data collected from multiple devices using logging platforms such as syslog as the input. The log data must contain at the very least a time stamp, source hostname, and free-format statement. Users can search and aggregate the log messages with time stamp, hostname, and template identifier from the database storage by means of amulog functions afterward.

The key idea of amulog is to use segmented log statements and segmented log templates consistently. Many log template generation methods include word segmentation in some form, but the implementations of these methods usually restore the segmented templates to the original format [18], [32]. In contrast, we keep the log statements segmented and store them in a database. By applying a constant message segmentation, we can handle the data flow more simply and compare (or combine) template generation methods in a consistent manner.

Amulog first parses an input log message into header items (i.e., time stamp and hostname) and a sequence of words corresponding to the free-format log statement. Next, amulog determines a corresponding log template for the segmented statement by means of log template generation (i.e., estimates log templates dynamically) or log template matching (i.e., uses pre-generated log templates). In amulog, the log template generation methods are external modules. Log template matching can be combined with other log template generation methods (e.g., basically using template matching, and estimating log templates for unfamiliar log messages). Finally, amulog stores the data structured with the above two steps in a database. The stored data include time stamps, source hostnames, log template identifier, and all words in the statement.

A. Log parsing

Log messages have two parts of information in each line: a structured header part (e.g. time stamp and source hostname) and an unstructured statement part (e.g., event description). The statement part is free-format, but partially forms a kind of formal languages because it is output by filling the replacers (e.g., format specifiers) with variables by the system programs. This means that a statement part can be segmented into a sequence of words, just like other languages can be. Many log template generation algorithms use this feature.



```
/kernel: xe-4/3/1: DAD detected duplicate IPv6 address 2001:0db8::1 NS in/out=3/3, NA in=0
```

Fig. 2. Example of log messages with inconsistent symbol string usage.

There is a problem with the word segmentation process of the log template generation; some log messages are inconsistent with the usage of symbol strings because the statement part is free-format. Figure 2 shows an example of such log messages in our dataset (details in § III) in which some of the sensitive variables (such as IP addresses) have been replaced. Focusing on the slash symbols ("/"), some of them are used as separators of words, while others are part of the variable, which is an interface name in this case. If we split the statement with a word segmentation rule that uses slashes as separators, the interface name "xe-4/3/1" will be torn into smaller pieces that lose the information of interface. In contrast, if we leave the slash in the word segmentation, the words "in" and "out" are not available in the log template generation and further analysis. No single static rule can parse this log statement correctly.

In response to this issue, we propose log2seq, a rule-based log message parser that converts string log messages into headers and segmented log statements. The key idea of log2seq is to fix known variable words while parsing. For example, a sample configuration of log2seq consists of four steps. First, we parse the header part using a static rule of regular expressions. Next, we split the statement part into word sequences using standard separator symbols (e.g., spaces and brackets). Then, we fix the known variable words that should not be separated later (e.g., interface names). Finally, we split the words by inconsistent symbols (e.g., slashes).

The configuration for log2seq consists of two rules: the header parsing rule and the statement segmentation rule. The header parsing rule is a simple regular expression to parse the header information. The statement segmentation rule is

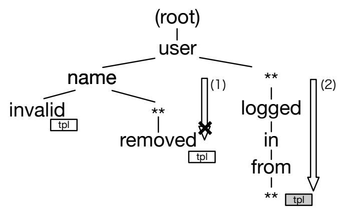


Fig. 3. Example of a template search tree.

a collection of multiple actions for word segmentation. The actions can be classified into two types: SPLIT for separator symbol strings and FIX for known variable words. The user customizes the configuration of these rules by modifying the orders and corresponding regular expressions in these actions.

In our experience, the configuration of log2seq largely depends on the vendor of the devices and applications. Still, the configuration can be easily shared in engineer communities because it does not include any internal information of the systems and devices.

The source code of log2seq is publicly available on GitHub [29], and can be easily installed as a PyPI package.

B. *Template matching*

Log template generation must deal with two issues: determining the log template structure (structure issue) and classifying the log messages with the templates (classification issue). Solving the structure issue enables us to obtain an abstracted log representation and to determine variable words in templates. These are important when we compare variables in the messages and apply NLP-based log analysis methods that focus on descriptive messages [8], [9]. On the other hand, solving the classification issue is required for event-oriented time-series analysis. However, not all template generation methods satisfy both issues. For example, a set of generated templates from source code analysis [15]–[17] is not mapped to the real log messages (i.e., does not solve the classification issue). These methods need template matching to use classified log data in further analysis. Therefore, log template matching is a mandatory function for any general log analysis framework.

To design a log template-matching method, it is important to consider the scalability of the log template matching to accept diverse template generation methods. A straightforward approach for log template matching is to use regular expressions, but this approach is not efficient because the degree of freedom of the regular expressions is too large due to multiple variables in the log messages. Besides, the computation complexity is $O(n)$ in the straightforward approaches, where n is the number of the log templates for template matching.

To resolve this issue, we propose a log template-matching algorithm based on a prefix-tree. In this algorithm, we first make a search tree of the given log templates, as shown in Figure 3. Every node has a word or a wildcard, and some nodes have log template identifier that indicate the end of the templates. This tree consists of three log templates: "user ** logged in from **", "user name invalid", and "user name ** removed" ("**" represents a variable word). The searching algorithm is a kind of depth-first one; the searching process starts from the root node and the next node is determined by matching the words with the nodes from the top of the input statement. There can be several potential branches of the searching paths because some nodes are the wildcard for variable words. In that case, we first select a node with a static word and then search another node with a wildcard. If there is no next matching node, the searching path is discarded and searching for the next path begins. If the node after passing all words in the input statement has a log template identifier, the searching process finishes.

For example, assume the log statement "user name logged in from 192.0.2.5" is given for matching with the tree in Figure 3. First, amulog traces branch (1) and fails with the fourth word "removed". Next, amulog traces branch (2) and succeeds to obtain a corresponding log template identifier.

The computational complexity of the tree-based template-matching method is $O(2^m)$ in the worst case where m is the number of words in the input statement. However, it is a rare case that amulog needs to search for multiple paths for a log message with practical log templates. The paths to be searched increase only when a word matches both nodes of the equal static word and that of a wildcard, but template generation methods usually unify them because most of the words in log messages are used for either a static word or a variable.

C. *Implementation*

Amulog is implemented in Python 3. Amulog provides two processing modes of log template generation: online and offline. Online processing is designed for continuous data collection and real-time log analysis. Offline processing is designed for hindsight analysis with parallel processing. Amulog currently supports multiple log template generation methods including online methods (e.g., Drain [24], LenMa [33], FT-Tree [6], and CRF [27]) and offline methods (e.g., Dlog [26]). Amulog can use SQLite or MySQL as data storage. The source code of amulog is available on GitHub [30].

III. Evaluation

We demonstrate a comparison of the processing times of various log template-matching approaches with the proposed tree-based algorithm in amulog. We use a set of backbone network syslog data obtained from SINET4 [31] to evaluate amulog performance. SINET4 is a nation-wide R&E network connecting over 800 organizations in Japan. The network consists of eight core routers and over 100 Layer-2 switches provided by multiple vendors. We manually generate ground truth log templates (1,788 different templates) for the 15-month of log data (35 million lines in total). In the following experiments, we use an Ubuntu 18.04 server (x86_64) equipped with an Intel(R) Xeon(R) Silver 4110 (2.10 GHz) and 64 GB of memory.

Four template-matching approaches are used for the comparison: TREE, TABLE, RE, and RE-Hash. TREE is our proposed tree-based method described in § II-B. TABLE is a simple

TABLE I
COMPARISON OF LOG TEMPLATE GENERATION ALGORITHMS.

Algorithm	Time	No. of clusters	F1-score
Manual	–	1,504	–
Drain	5,478	5,418	0.990
RegEx	7,912	27,039	0.717
CRF	14,146	215,934	0.998
CRFe	14,302	9,072	0.998

method based on log2seq that searches the log template with the same length (i.e., the same number of words) of log statement and matches all words except wildcards. RE uses regular expressions for template matching. We automatically generate a regular expression for a given log template. RE-Hash is also based on regular expressions, but it classifies the regular expressions by the hash of initial characters in the log statement (we use five characters; e.g., "/kern" in Figure 2) to decrease the number of regular expressions to match a message. RE and RE-Hash do not translate the input log statements into a sequence of words with log2seq. The computational complexity of TABLE and RE is $O(n)$, where n is the number of given log templates.

We also generate five template sets for the comparison of template-matching methods using the different log template generation methods implemented in AMULOG. We use first three months of log data for stabilizing the models, and following twelve months for the template generation. For each methods, Table I shows the processing time, number of clusters (i.e., number of templates), and their accuracy in pair-wise F-measure score with the ground truth (Manual). Drain [24] is an online template generation method based on heuristics and clustering with common words. RegEx is a simple method based on manually defined regular expressions for commonly used variables. CRF [27], [34] is an NLP-like approach that estimates template structures with supervised learning (we use 1,000 manually labeled messages for training) for log messages. We also extend the CRF-based template generation in two ways (CRFe): (1) using mid-labels generated by regular expressions (same as RegEx) and (2) sampling training data for manual labeling with a preliminary simple clustering (based on SLCT [35]) to learn more about minor templates.

We evaluate the processing time to generate an amulog database from 1-day log data (76,719 lines) with the four different template-matching methods and the generated templates. The processing time includes all of the standard amulog processes: initializing the matching model from the templates, parsing input messages with log2seq, classifying the messages by the template matching, and generating a database.

Figure 4 shows the log-log-scale comparison of the processing time with the four template-matching methods. Our proposed method, TREE, takes about 80 seconds even if more than 10^5 templates are given as template candidates. We can see that the difference in the processing time is small with a smaller number of template candidates, but this difference becomes larger for more template candidates. The processing time of template matching with TREE is nearly consistent, and the

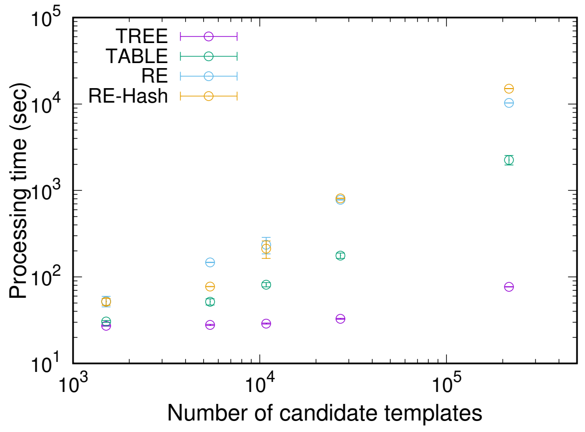


Fig. 4. Processing time of template-matching methods.

increase for larger template candidates stems from the model initialization step (because the number of given templates is larger than the number of log messages). The processing time of TABLE depends largely on the number of templates, but even so, it is even faster than RE and RE-Hash. This is because the message segmentation enabled us the partial matching of messages: the TABLE algorithm does not need to match all words in a template if the top word is neither a matching word nor a wildcard. In contrast, RE and RE-Hash need to match all the message with the regular expressions. Besides, RE-Hash is not effective compared to RE in this result. In our dataset, most templates in the larger template sets start with variable wildcards, so the hash of initial characters cannot decrease the candidate templates for matching. In summary, the amulog design with segmenting messages is effective for efficient template matching, and the proposed tree-based method improves the scalability of template matching.

IV. CONCLUSION

In this paper, we designed and implemented a general framework, amulog, for template-based log analysis. Our design is characterized by a simplified data flow while using segmented log messages consistently. We implemented amulog considering the three issues facing framework design: rule-based log parsing, tree-based template matching, and database schema. Our evaluation demonstrated that the proposed template matching algorithm is scalable enough to match 1-day log messages (76,000 lines) with more than 10^5 templates in 80 seconds.

As future work, we will implement state-of-the-art template generation methods as external modules for amulog. We will also share the log2seq configurations for major devices and applications. In addition, we will consider further log analysis approaches based on amulog.

ACKNOWLEDGEMENTS

This work is supported by JSPS KAKENHI Grant Number JP19K20262, and the MIC/SCOPE #191603009.

References

[1] T. Kurimoto, S. Urushidani, H. Yamada, K. Yamanaka, M. Nakamura, S. Abe, K. Fukuda, M. Koibuchi, H. Takakura, S. Yamada, and Y. Ji, "SINET5: A low-latency and high-bandwidth backbone network for SDN/NFV Era," in *Proceedings of IEEE ICC*, 2017, pp. 1–7.

[2] J. Bai, "Feasibility analysis of big log data real time search based on Hbase and ElasticSearch," in *Proceedings of ICNC*, 2013, pp. 1166–1170.

[3] T. V. Cuong and N. V. Nam, "An Efficient Log Management System," *VNU Journal of Computer Science and Communication Engineering*, vol. 32, no. 2, pp. 43–48, 2016.

[4] H. Abe, K. Shima, D. Miyamoto, Y. Sekiya, T. Ishihara, K. Okada, R. Nakamura, and S. Matsuura, "Distributed hayabusa: Scalable syslog search engine optimized for time-dimensional search," in *Proceedings of AINTEC'19*, 2019, pp. 9–16.

[5] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto, "Spatio-temporal factorization of log data for understanding network events," in *IEEE INFOCOM'14*, 2014, pp. 610–618.

[6] T. Li, J. Ma, and C. Sun, "Dlog: diagnosing router events with syslogs for anomaly detection," *The Journal of Supercomputing*, vol. 74, pp. 845–867, 2018.

[7] K. Otomo, S. Kobayashi, K. Fukuda, and H. Esaki, "Latent variable based anomaly detection in network system logs," *IEICE Transactions on Information and Systems*, vol. E102.D, no. 9, pp. 1644–1652, 2019.

[8] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *IJCAI*, 2019, pp. 4739–4745.

[9] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J. G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of ESEC/FSE*, 2019, pp. 807–817.

[10] Z. Zheng, L. Yu, Z. Lan, and T. Jones, "3-Dimensional root cause diagnosis via co-analysis," in *Proceedings of ICAC*, 2012, pp. 181–190.

[11] S. Kobayashi, K. Otomo, K. Fukuda, and H. Esaki, "Mining causes of network events in log data with causal inference," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 53–67, 2018.

[12] T. Li, J. Ma, Q. Pei, Y. Shen, C. Lin, and S. Ma, "AClog : Attack Chain Construction Based on Log Correlation," in *Proceedings of IEEE GLOBECOM*, 2019, pp. 1–6.

[13] D. El-Masri, F. Petrillo, Y. G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology*, vol. 122, pp. 1–23, 2020.

[14] M. Landauer, F. Skopik, M. Wurzenberger, and A. Rauber, "System log clustering approaches for cyber security applications: A survey," *Computers and Security*, vol. 92, no. 101739, pp. 1–17, 2020.

[15] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," *Proceedings of SOSP*, pp. 117–132, 2009.

[16] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models," in *Proceedings of IEEE IC2E*, 2016, pp. 62–67.

[17] M. Zhang, Y. Zhao, and Z. He, "GenLog: Accurate Log Template Discovery for Stripped X86 Binaries," in *Proceedings of COMPSAC*, 2017, pp. 337–346.

[18] R. Vaarandi and M. Pihelgas, "LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs," in *Proceedings of CNSM*, 2015, pp. 1–8.

[19] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of ACM KDD*, 2009, pp. 1255–1264.

[20] L. Tang, T. Li, and C.-s. Perng, "LogSig : Generating System Events from Raw Textual Logs," in *Proceedings of ACM CIKM*, 2011, pp. 785–794.

[21] M. Mizutani, "Incremental mining of system log format," in *Proceedings of IEEE SCC*, 2013, pp. 595–602.

[22] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "LogMine: Fast Pattern Recognition for Log Analytics," in *Proceedings of ACM CIKM'16*, 2016, pp. 1573–1582.

[23] T. Kimura, A. Watanabe, T. Toyono, and K. Ishibashi, "Proactive failure detection learning generation patterns of large-scale network logs," *IEICE Transactions on Communications*, vol. E102.B, no. 2, pp. 306–316, 2019.

[24] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proceedings of IEEE ICWS'17*, 2017, pp. 33–40.

[25] T. Qiu, G. Tech, Z. Ge, F. Park, D. Pei, and J. J. Xu, "What Happened in my Network ? Mining Network Events from Router Syslogs Categories and Subject Descriptors," in *Proceedings of IMC*, 2010, pp. 472–484.

[26] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, and D. Pei, "Syslog Processing for Switch Failure Diagnosis and Prediction in Datacenter Networks," in *Proceedings of IEEE/ACM IWQoS'17*, 2017, pp. 1–10.

[27] S. Kobayashi, K. Fukuda, and H. Esaki, "Towards an NLP-based log template generation algorithm for system log analysis," in *Proceedings of CFI*, 2014, pp. 1–4.

[28] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-Supervised Log Parsing," *arXiv*, pp. 1–16, 2020.

[29] "log2seq," https://github.com/cpflat/log2seq.

[30] "amulog," https://github.com/cpflat/amulog.

[31] S. Urushidani, M. Aoki, K. Fukuda, S. Abe, M. Nakamura, M. Koibuchi, Y. Ji, and S. Yamada, "Highly available network design and resource management of sinet4," *Telecomm. Systems*, vol. 56, pp. 33–47, 2014.

[32] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," in *Proceedings of ICSE-SEIP*, 2019, pp. 121–130.

[33] K. Shima, "Length Matters: Clustering System Log Messages using Length of Words," *arXiv*, pp. 1–10, 2016.

[34] J. Lafferty, A. Mccallum, and F. C. N. Pereira, "Conditional Random Fields : Probabilistic Models for Segmenting and Labeling Sequence Data," in *Proceedings of ICML*, 2001, pp. 282–289.

[35] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of IPOM*, 2003, pp. 119–126.