HAL
open science

LogOS: an Automatic Logging Framework for Service-Oriented Architectures

Stéphane Frénot, Julien Ponge

To cite this version:

Stéphane Frénot, Julien Ponge. LogOS: an Automatic Logging Framework for Service-Oriented Architectures. 38th Euromicro Conference on Software Engineering and Advanced Applications, Sep 2012, Izmir, Turkey. hal-00709534

HAL Id: hal-00709534

https://inria.hal.science/hal-00709534v1

Submitted on 12 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

LogOS: an Automatic Logging Framework for Service-Oriented Architectures

Stéphane Frénot
Université de Lyon
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne, France
stephane.frenot@insa-lyon.fr

Julien Ponge
Université de Lyon
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne, France
julien.ponge@insa-lyon.fr

***Abstract*—As multi-source, component based platforms are becoming widespread both for constrained devices and cloud computing, the need for automatic logging framework is increasing. Indeed, components from untrusted and possibly competing vendors are being deployed to the same runtime environments. They are also being integrated, with some components from a vendor being exposed as a service to another one. This paper presents our investigations on an automated log-based architecture called LogOS, focused on service interactions monitoring. We developed it on top of Java / OSGi to enable identification between bundle providers in cases of failures. We motivate the need for an automatic logging framework in service-oriented architectures, and discuss the requirements of such frameworks design. We present our implementation on OSGi. Finally, we position our approach and give some perspectives.**

***Keywords*-soa; cbse; osgi;logging; service;**

I. Introduction

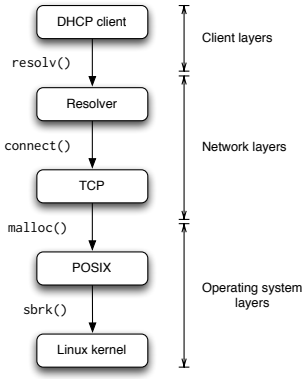
While software developers and integrators have been using logging systems for technical purposes, we argue that a new breed of need of logging approaches is emerging from an increase in transparency requirements from end-users and isolated nature of components. For example, a user may contest having authorized a credit-card payment from a mobile device. In this case, a trusted log mechanism shall be able to clearly spot the software component at fault so as to blame its provider and eventually its chain of responsibilities. Moreover, a user may be concerned in knowing what the applications (s)he is using on a device, actually do: a solitaire game is probably not expected to access a contacts list and connect to the Internet.

This paper presents our investigations on an automated log-based assertion architecture for Java component-based systems called LogOS, applied to OSGi environments. We believe that services such as found in OSGi platforms are the best level of granularity for such logging. Indeed, their implementations are "owned" by a self-contained unit, called a bundle, and each of them originates from a single vendor. Services rely on other services, possibly from third-party vendors, hence stressing out their relevance as a target identification.

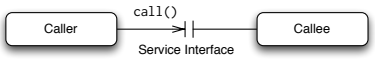
We first motivate the need for a logging framework in service-oriented architectures and focus on the needs for capturing and managing log records. Next, we present LogOS architecture ported to Java / OSGi, and how the aforementioned requirements need to be implemented for this platform. Finally, we present our experiments with our OSGi-based implementation, before concluding with related work and perspectives.

II. Designing an Automatic Log Record Management Framework

Our proposed architecture focuses on service oriented logging for component based architecture. It focuses on two principles for component activity logging: horizontal calls and black-box approaches. In horizontal calls, every available service function is provided by another component, either remotely or locally, and invocations are sent to the same layer. There are no existing trusted relationships between callers and callees, in the sense that callers and callees have the same level of legitimacy within the system. Figure 1 illustrates, horizontal



(a) Vertical calls layers.

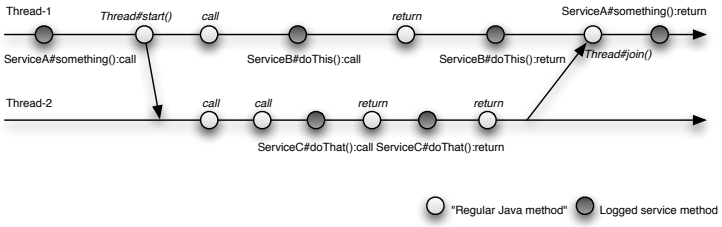


(b) Service-oriented horizontal call.

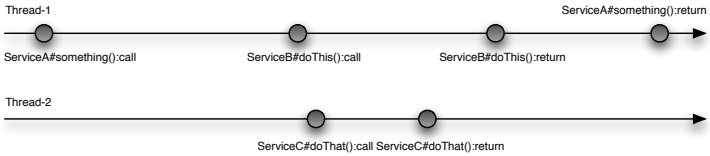
Fig. 1. Vertical and horizontal calls.

and vertical calls. Our architecture also enforces the fact that component must be hosted and logged without knowing their internal behavior. Their interface may be publicly available, but the internal is not. In this context, the exact behavior

is unknown as illustrated figure 2. Figure 2(a) shows the real behavior, whereas a black-boxed architecture may only observe Figure 2(b) behavior.



(a) Observing through instrumentation.



(b) The LogOS view of the observation above.

Fig. 2. Instrumentation *versus* observation in LogOS.

We are focused on service oriented architectures, where each service is associated with a specific owner. Service interfaces are associated to log frontiers and their running implementation must be considered as black boxes. As service interfaces are at the frontier, they are outside of the black-boxes and can have specific log recording behavior. With these constraints, our architecture is able to capture, store and provide the best representation of what previously happened within the system.

A. Log Records Data

A log record contains the following information.

Identification: A log entry must be totally unique and unpredictable.

Timing: When analyzing log information, it is important to have a clear vision of events ordering.

Participants: Providing a clear identification of participants is mandatory to characterize communications.

Content: Depending on the logging objective, the content may be empty or a complete mirror of the event information.

B. Capturing Logs

A log capture system raises five issues: session management, reliability, horizontal logs, data isolation and data volume.

Session Management: A session is a semi-permanent information interchange between two or more communicating devices. A session is set up or established at a certain point in time, and torn down at a later point in time. In a black-box approach, many sessions can be active at the same service frontier of observation. Hence, as part of the log record, one needs to store the identification information for every active session at the time of the service invocation. Indeed, any subsequent invocation may belong to any of those active sessions.

Reliable Interceptions: We need to record events as soon as possible, since any crash may lead to a loss of data. Thus, when client/server communication occurs clients are blocked until initial and response calls have been completely logged, and volatile memory buffers, and remote persistence for storing events are not possible. Shall the logging system be unable to proceed call interception, the intercepted service invocation must fail and possibly stop the framework.

Data Isolation: When sending requests through the Internet, service requesters and service providers run at the very least on different processes. Every transmitted parameter is a deep copy. In an automatic logging framework, frontiers need to be hard frontiers, which means that anything that transits through them is entirely immutable from the other side. The logging framework shall provide a defensive copy before sending any object parameter or return value. In some circumstances it is not possible to make a defensive copy. In these cases, specific contracts are set up between the provider and the requester as a specific trust relationship is assumed between them.

Data volume: When managing service parameters, the data volume is an important matter. Although we can store all parameters, we must provide intermediary approaches such as data compression, hashing of values and even value discarding.

C. Log Entry Storage

When log record events have been produced we need to store them for later queries. The following elements must be taken care of.

Complete: Events must be complete from the very beginning of log records. Although it may not be started at the beginning of the system, once it has been launched the log records must be continuous, and every logged event must be stored. Since it is impossible to anticipate the kind of problem a user may raise, we need to maintain this constraint as long as we can. A new log event must not be captured until the current one as been permanently stored.

Unforgeable and Confidential: Every log record must not be modified, substituted or inverted with another. Log storage contains sensitive data and thus must be protected from external viewers. Log records shall only be accessed by the implied parties including the end-user and no one else. Moreover, implied parties shall only access their concerned record and nothing else.

All or Nothing: In case of failure the log system must be invalidated whatever happens later. Thus the logging framework works in an all or nothing way.

The last step in log management is dedicated to log record querying.

D. Log Querying

Log records should be mostly accessed off-line. We identified the following requirements for an efficient log querying system.

Session graphs rebuilding and navigation: The first goal of the log querying feature is to be able to rebuild a complete session from the log. A complete session must include the starting event, and every possible calling graph formed with all active sessions until the session ends.

Operators for graph comparison: An important feature when dealing with log graphs is the possibility to apply operators to them. The most obvious one is a comparison operator, but it may be complex to compare, since two session runs shall differ with parameters or timing issues. Many works exist when dealing with log analysis and mining. Although many of them work on partial log records and try to cope with incompleteness [1], [2], our framework provides complete log records changing the kind of available analysis.

In the next section we describe our reference implementation focused on OSGi logging.

III. LogOS: a Logging Framework for OSGi

While there exist numerous service-oriented architecture implementations, OSGi is a noteworthy one, as it provides a mean to assemble and integrate applications within the bounds of a single Java virtual machine. The need for automatic logging is stressed out with the notion of *bundle* in OSGi that unambiguously identifies a component, its version and its vendor.

The first subsection details the specificities OSGi architecture raises when dealing with true logging. The second subsection presents a dedicated language to drive logging behavior. The final subsection summarizes the LogOS OSGi architecture.

A. Impact of OSGi Specifications on Logging

All elements described in Section II are handled by the LogOS architecture yet, some elements need special attention.

Isolation enforcement: OSGi uses the Java platform, the Java language and its semantics. Data is passed by-reference when service method calls are initiated, meaning that both the client and service hold a reference to the same data objects in memory. As Java objects are generally mutable, data may change after a service method has been invoked. Although the code is owned by one party, another one may still have a reference to a service-provided object and modify it. To overcome this, as well as to provide true data isolation between services upon method invocations, LogOS must make defensive deep copies, of the parameters and return values.

However, there are cases where defensive copies are incompatible with the programming model between the parties. This happens when performance is critical, when parties have settled formal agreements, or simply because class objects shall not be deep-copied and/or be proxy-ed. For such cases LogOS provides a specific frontier where both caller and callee parties are deemed liable in case of claims.

Session Management: Many sessions can be simultaneously active at the same logging time, and each of them must be attached to the record. OSGi does not natively provide any session management scheme. LogOS provides a way to identify session services that are the only services allowed to initiate a log record chain.

B. LogOS Domain-Specific Language

We designed an annotation-based domain-specific language [3] to drive log record behavior at the frontier and cater for the constraints of OSGi.

The domain-specific language is defined via Java Annotations[1] that modify how method calls are logged. These optional annotations may be set-up within the service interfaces since they define the frontier and do not break the black-box principle.

C. LogOS Architecture

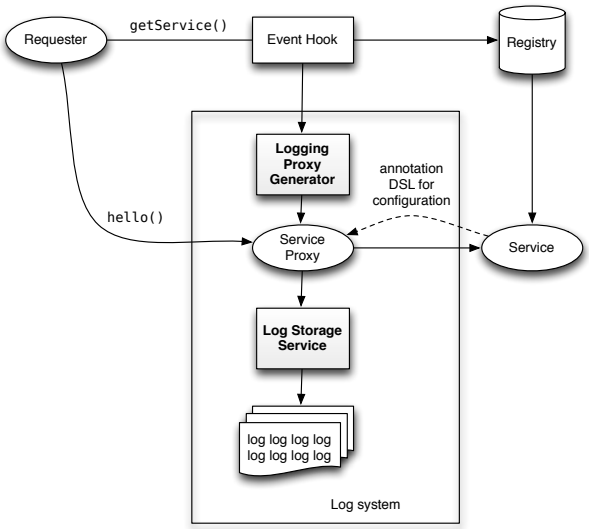


Fig. 3. Implementation of LogOS on OSGi.

Figure 3 shows the global LogOS interception architecture. It is brought through a unique bundle started at bootstrap time. The *eventhook* builds an interception proxy that generates log events. It is based on ServiceHook OSGi specification that enables service implementation substitution while registering. Each time a new implementation is registered, the service hook is triggered and a Java dynamic proxy is forged from the registered interface analysis provided by the *Logging Proxy Generator*. The Service Proxy may be parametrized with the Service annotations. Finally, the *storage* service provides encryption and deflate compression [4] streams to persist log events.

IV. Conclusion

A. Related Work

Logging frameworks are generally inserted and configured within software code. However, certain tools exist to automatically trace and log executions of programs without modifying them. The *strace* tool is well-known for Unix variants, and allows capturing system-level calls. The arguably

[1] http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html

more powerful *dtrace* tool has lead to several publications on its design, implementation and usages [5], [6], [7]. Indeed, it allows focusing on any function and resource usage, and customizable actions can be triggered.

Several works focus on system, middleware and application failures and dependability [8], [9], [10]. Closer to our field of study, [11] characterizes failures on two widely used mobile operating system platforms: Google Android and Symbian. In [12], the authors focus on software dependencies and their impact on failures. A log-based dependability approach for Microsoft Windows operating systems is proposed in [13], and another log-based approach for large-scale systems is detailed in [14].

Log events have been used beyond dependability in several works. They are used for adaptive scheduling in [15]. Business processes are inferred for re-engineering based on actual usage and/or lack of documentation in [1]. There are essentially two types of approaches: either logs are processed in a post-mortem fashion for the needs of some form of analysis, or they are used as live events sent to a monitoring entity which may in turn apply decision rules. We envision to leverage LogOS for both purposes.

This work, in the continuation of [16], differs from those approaches in the sense that we do not focus on identifying bugs and failures. Instead, we aim at providing a middleware where interactions between vendor components can be assessed and isolation can be automatically enforced.

B. Perspectives

This article introduced LogOS, an automated logging framework for service-oriented architectures. We discussed the requirements and motivations behind such system. We implemented it in the context of Java / OSGi. There exists a necessary performance impact of activating a strong logging framework, as every intercepted method invocation triggers a filesystem synchronous write operation. In the meantime, services are generally used as coarse-grained interfaces to more complex, CPU-intensive components, hence the logging overhead can be mitigated most of the time.

LogOS architecture is agnostic of any application and work as soon as service service calls are used. We successfully used LogOS in our internal OSGi based projects as a way of efficiently logging applications behaviors. LogOS runs without any implementation modifications and is able to log service interactions, so as session life-cycles, from every OSGi based applications we have. Besides this, we successfully extended the framework to do more things than events interceptions. For instance we plugged LogOS to a formal method monitoring tool called Larva [17] to control service execution behavior.

Future works will tackle the behavior-based analysis of the captured interactions. This will be used for manual post-mortem liabilities analysis, as dealing with raw log outputs can be hard to do, especially when concurrent interleaving happens. This will also be useful for the automated reasoning about service behaviors, including the need for fine-grained compatibility and replacement assessment like it has been done on Web Services in [18], or the need to perform some model checking on the fly.

One question we regularly have is that of knowing when a service shall be defined rather than a regular class. Indeed, the distinction between a library call and a service call is not always obvious. When services are run on the same system, they directly impact performances without providing too much advantage. LogOS identifies services as a liability separation between requester and publisher. Each time a service interface appears, it means an interaction exists where liabilities could be inferred. We found that LogOS to be an excellent tool for designing service oriented applications since it makes service contracts mandatory at service frontiers.

Acknowledgment

This project was supported by the ANR LISE grant (*ANR-07-SESU-007*). We would also like to thank Stéphane Chevalier and Denis Beras, two software engineers who helped us on this project.

References

[1] H. R. M. Nezhad, B. Benatallah, F. Casati, R. Saint-Paul, P. Andritsos, and A. Guabtni, "Exploration of discovered process views in process spaceship," in *ICSOC*, ser. LNCS, vol. 5364, 2008.

[2] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters, "Workflow mining: A survey of issues and approaches," *Data Knowl. Eng.*, vol. 47, no. 2, 2003.

[3] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley, 2010.

[4] P. Deutsch, "Deflate compressed data format specification version 1.3," United States, 1996.

[5] B. Cantrill, "Hidden in plain sight," *ACM Queue*, vol. 4, no. 1, 2006.

[6] B. Cantrill, D. Price, and L. Praza, "Solaris 10: System/dtrace/zones/smf," in *LISA*. USENIX, 2005.

[7] B. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems." USENIX, 2004.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001, pp. 73–88.

[9] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *DSN*. IEEE Computer Society, 2000, pp. 97–106.

[10] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability: A study of field failures in operating systems," in *FTCS*, 1991, pp. 2–9.

[11] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *ISSRE*. IEEE Computer Society, 2010, pp. 249–258.

[12] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Software Eng.*, vol. 35, no. 6, pp. 864–878, 2009.

[13] C. Simache, M. Kaâniche, and A. Saïdane, "Event log based dependability analysis of windows nt and 2k systems," in *PRDC*. IEEE Computer Society, 2002, pp. 311–315.

[14] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN*. IEEE Computer Society, 2007, pp. 575–584.

[15] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *EuroSys*, 2010.

[16] D. Le Métayer, M. Maarek, E. Mazza, M.-L. Potet, S. Frénot, V. Viet Triem Tong, N. Craipeau, R. Hardouin, C. Alleaune, V.-L. Benabou, D. Beras, C. Bidan, G. Goessler, J. Le Clainche, L. Mé, and S. Steer, "Liability in Software Engineering Overview of the LISE Approach and Illustration on a Case Study," in *ACM/IEEE 32nd International Conf. on Software Engineering (ICSE 2010)*, Cape Town, South Africa.

[17] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)," ser. SEFM '09. Washington, DC, USA: IEEE Computer Society, 2009.

[18] J. Ponge, B. Benatallah, F. Casati, and F. Toumani, "Analysis and applications of timed service protocols," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 4, 2010.