A View of Monitoring and Tracing Techniques and their Application to Service-Based Environments

Heidar Pirzadeh and Abdelwahab Hamou-Lhadj

Department of Electrical and Computer Engineering

Concordia University

1455 de Maisonneuve West, Montréal, Québec, Canada

{s_pirzad, abdelw}@ece.concordia.ca

Abstract

Software systems are perhaps today's most complex engineering systems due to the ever evolving technologies they employ. Understanding how these systems are built and why they are built this way calls for advanced tools and techniques that go beyond mere analysis of the source code as it is the case of most existing software comprehension and modernization approaches. In this paper, we argue that the complex behavior embedded in most distributed, multi-tier, and service-based software systems can benefit significantly from applying dynamic analysis approaches such as the ones based on monitoring and tracing techniques. The main advantage of these techniques is that they constitute a natural fit with the distributed paradigm that is the main mechanism of such applications. In this paper, we present several monitoring and tracing techniques, and compare them based on their advantages and disadvantages. We then discuss how these techniques can be used to understand serviced-based applications, with the ultimate objective being to uncover the key challenges that need to be addressed.

Keywords: Monitoring and tracing techniques, software modernization and comprehension, service-based applications, distributed systems.

1. Introduction

One of the key challenges in modernizing exiting software systems is the ability to understand what the system does and why it does it this way. In an ideal situation, a software engineer relies on up-to-date documentation to make changes to the system in a way that preserves the system's functionality. However, maintaining sufficiently good documentation has been shown to be difficult to achieve in practice. This is due to several factors including time to market constraints, the cost of maintaining documentation not justifying the benefits, the documentation being poorly designed in the first place, etc. To reduce the impact of this problem, there exist several software modernization techniques that aim to investigate techniques and build tools to help software engineers understand a poorly documented software system.

These techniques depend significantly on the ability to collect information about the system under study. Data collection techniques can be grouped into two categories, namely, static analysis and dynamic analysis. Static analysis uses the source code as its main artifact to uncover the system's main components and their relationships. Performing static analysis has the benefit that all the program's execution paths could be potentially covered. However, it can only reveal the static aspects of the system, and it is very limited in providing insights into the behavioral characteristic of a program design. This knowledge of the behavior of a system can be critical for the analysis of distributed applications such as service-based systems due to the high interactions they involve.

Dynamic analysis, which is the focus of this paper, is the study of how the system behaves by analyzing its execution traces. Unlike static analysis, dynamic analysis has the advantage of allowing the software analyst to focus only on parts of the program that needs to be analyzed by studying the interactions among the involved components [4]. In addition, as noted by Ball [4], dynamic analysis can be very useful for applications that require the understanding of the system's behavior by relating the system inputs to its outputs. There exist two types of dynamic analysis: Online (ante-mortem) analysis and Offline (post-mortem) analysis. Online analysis is the analysis of the behavior of an active system while it is running. This type of dynamic analysis comes handy when the system under analysis is not going to terminate its task any time soon (e.g., servers).

However, online dynamic analysis is limited when the performance is important since it usually imposes a large overhead on the application being analyzed. In offline dynamic analysis, on the other hand, the analysis time is different from the time when event traces are collected. That is, the event traces are collected during the execution of the system, while the analysis is usually performed when the execution is finished. In this way, offline analysis can reduce overhead on the application. However, one of the major problems of offline dynamic analysis is the huge amount of information that has to be processed and stored as we will describe in the next section [41].

Service-based systems are distributed systems where several services communicate with one another to implement a particular functionality. These services can be composed of other services depending on the application that uses them. While some services can be known in advance, others may be unknown and are composed on the fly (dynamic composability). In addition, the internal implementation of many services that compose an application might be hidden to this application. This ability to compose services on the fly, combined with the limited access to service implementation, can pose significant challenges to using dynamic analysis to analyze such applications.

The objective of this paper is to discuss monitoring and tracing techniques and the challenges they represent when applied to service-based systems. We also propose some solutions that overcome these challenges. We believe that understanding these challenges and working towards addressing them is a key step in developing effective techniques for tracing and monitoring service-based applications.

Organization of the paper: The paper is divided into two main parts. The first part (Section 2) discusses existing monitoring and tracing techniques, their advantages and disadvantages. These techniques are also compared according to several properties. The section part (Section 3) focuses on service-based applications and the challenges (and proposed solutions) that need to be addressed when applying monitoring and tracing to such application. We conclude the paper in Section 4.

2. Execution Monitoring and Tracing

The term *monitoring* is defined by Oxford Dictionary as “keep under observation, especially so as to regulate, record, or control” [42]. Using the same definition, *execution monitoring* can also be seen as the task of monitoring execution of a software system to understand, report, and analyze what it does and why it does it this way.

Execution monitoring is used for various purposes including profiling, testing and debugging, performance evaluation, program optimization, software failure detection, malicious code detection, diagnosis, and design recovery [11]. It relies on tools, known as execution monitors, to collect information about a program’s execution [34]. This information, more specifically insights gained through execution monitoring, cannot be obtained by simply studying the source code as suggested by static analysis. This is due to the increasing complexity of the behavior embedded in most today’s distributed systems.

2.1. Categories of Execution Monitoring

Execution monitoring can be grouped into two categories, which differ in the way information is collected: *Sampling* (time-driven monitoring) and *Tracing* (event driven monitoring) [22].

Sampling techniques, also known as time-driven monitoring techniques, focus on observing the state of the monitored system in a timely manner. In sampling, information is collected at the request of the monitor. In addition, the information collection in sampling may be asynchronous with the occurrence of events internal to the program execution. Therefore, sampling can be practical when delay in reaction to an event is not important. For example, some statistical information about the execution may be collected at certain time intervals but this information would be insufficient for behavior analysis.

Tracing techniques, which tend to be event-driven monitoring techniques, report all occurrences of certain events within a certain time interval. In this way, tracing is synchronous with the occurrence of events. Therefore, tracing can be readily used to analyze the detailed behavior of a software system due to the fact that one can trace just about any event that occurs in the system [31]. It should be noted, however, that tracing is recommended when the occurrences of the events of interest are detectable - The events of interest are known in advance [37]. Tracing techniques are not limited only to software entities but can also be applied to hardware (e.g., tracing the performance of a CPU, etc.). One of the main issues with tracing techniques is the large volume of data generated in traces of even small systems. Large size of traces makes them hard to store, transfer, process, and analyze.

To overcome this problem, abstraction techniques can be used, which are concerned with the ability to reduce the size of the trace by removing (or hiding) events that are not of interest to the user depending on the purpose of the analysis. This could be done either automatically or semi-automatically. Examples of abstraction techniques include pattern matching techniques [10], the removal of utilities [43], etc. These techniques can be applied while the trace in being generated (ante-mortem analysis) or after the trace is generated and saved (post-mortem fashion).

2.2. What Can Be Traced?

Many different aspects of running programs can be monitored and traced. In fact, one can trace just about any aspect of the system that is deemed helpful to accomplish the task at hand. These different aspects include function and procedure calls and returns, variable values, loops and branches, inputs and outputs, inter-process communication, executed statements, system state transitions, external interrupts, system calls, as well as data structures such as process control blocks.

Generating fully detailed traces can consume a great deal of CPU time, resulting in high overhead. Therefore, tracing in software should be selective based on the purpose of monitoring. For example, if the objective of monitoring is to detect behavior or prevent security breaches then monitoring can be applied to observe the internal structure of the malicious software such as runtime code envelopes, data encryption/decryption engine, memory layout, etc. Therefore, it would be helpful to generate a trace by real time break points on memory reads/writes, port reads/writes, and interrupts. Softice [2], WinDBG [35], CWSandbox [40], and Cobra [39] are among the tools and techniques that fall into this category.

In addition, one can apply a more coarse-grained level monitoring technique to capture the behavior of a malicious code at a high level. For this level, a series of routines and/or instructions can be traced. Then, the trace will be analyzed to make sure that no subsequence in the trace has a malicious behavior (e.g., No “read file” followed by “connect to Internet”). Some of the known approaches in this area include SASI [14], Naccio [15], and Polymer [5].

Traces of routine calls have also been shown to be useful for solving practical maintenance problems such as uncovering faulty behaviors [38], adding new features to an existing system [24], or detecting system inefficiencies [9].

2.3. Monitoring Techniques

There exist different options for designing and implementing execution monitors. Although the design and implementation of monitors usually depend on information sources and access methods, currently, designers must choose between hardware monitoring, software monitoring, or hybrid monitoring techniques. In this section, we review hardware and software monitoring techniques. *Hybrid monitoring* is the combination of software monitoring and hardware monitoring approaches [8].

2.3.1. Hardware Monitors

Hardware monitors are used to monitor the internal system signals and their various patterns on the buses with hardware probes. This information could help an analyst understand the traffic on the memory bus, measuring systems performance, and tuning.

One advantage of using hardware monitoring is that first it is not intrusive, meaning that the monitor does not cause an overhead to the system under monitor and, second, it does not change the behavior of the system under study [7, 25, 28]. However, it requires using additional hardware devices such as snooping devices for the monitoring task. Hardware monitoring is usually a good choice when the objective of monitoring is to analyze the performance of the system.

Logic analyzers, for example, are devices that record processors and their bus activities to present a detailed view of the system’s execution. They can be used to evaluate performance of a system by timing its different segments. They usually include an oscilloscope for displaying various digital states. Another example would be to monitor dataflow between different processors across a network using communication monitors. Thus, they can be used to obtain statistics of the bus loading or collision rates. These devices do not interfere with the communications and cause no timing issues.

The disadvantage of hardware monitoring is the cost associated with the monitoring devices. In addition, hardware monitors tend to be machine dependent or at least processor dependent. Another major drawback of the hardware monitoring approach is that with the advanced in hardware technologies fewer physical probe points will be offered on newer systems (i.e., on-chip functionalities). This results in difficulties in collecting enough information to determine the behavior of the system and renders hardware monitoring less practical [16].

2.3.2. Software monitors

When we need to monitor a complex system, hardware monitors may be inapplicable and costly [18]. Another way to monitor systems is to apply software monitoring by inserting code inside the system under study that outputs information about the system execution. Selecting a software monitor depends on the type of data that needs to be collected such as function and procedure calls and returns, variable assignments, loops and branches, etc. A software monitor can be an add-on program or can be implemented by instrumentation of existing systems or the execution environment in which the system runs.

One simple way to implement software monitors is to add a new program to the system, which is responsible for monitoring the running of the existing system. In this case, the monitor sees the running program as a whole and can be used to investigate the effects of program execution. Therefore, this type of software monitors can only be used for sampling purposes as they are unaware of the system's internal events. The advantage of using add-on monitors is that they do not modify the monitored program or the environment in which the monitored program runs. Therefore, when the monitoring task is finished the monitor could be simply removed without causing any disintegration to the monitored program and its environment. However, a big shortcoming of add-on monitors is that they cannot be used for event-driven monitoring, i.e., the ability to generate traces that contain events of interest.

Instrumentation techniques overcome the shortcomings of add-on monitors by allowing the designer to insert probes in places of interests in the source code. Instrumentation can be performed manually [12, 30] where the software engineer inserts monitoring code (e.g., security checks, printouts, etc.) into the program. The obvious drawback with manual instrumentation is that it is usually labor intensive (should be done for each monitored program), and can be error-prone. In addition, there is a need to update or remove the monitoring code when an instrumented program is modified or the monitoring is finished.

Another instrumentation technique is known as online instrumentation [30, 27], where the modification of the program code is done automatically by inserting probes (usually printout statements) before or during the execution of the program. In this method, the code is modified to cause a temporary alteration to the program execution path. This alteration is usually done by inserting “jump” instructions before a predefined set of instructions. In this way, the control will be temporarily transferred to another code that performs the necessary monitoring actions. When these actions are performed, the control will be transferred back to the program by another jump. The advantage of using automatic instrumentation is that the code modifications could be done on any number of programs automatically. In the same way, the original programs can be recovered automatically from the modified versions. One difficulty with this method is the ability to select areas of interest that need to be instrumented. Furthermore, since a same set of predefined monitoring actions would be performed for an instruction regardless of the code, like most other instrumentation approaches, automatic instrumentation suffers from high overhead.

Unlike online instrumentation where code modification is done just before or during runtime, compiler-based instrumentation techniques add instrumentation to the code at compile time [17, 21, 29]. That is, they generate a monitored program code which can later on be executed. One key advantage of compiler-based instrumentation is that, unlike online instrumentation, it does not change the source code. In addition, it automatically instruments any program written in the target programming language of the compiler. One main disadvantage of compiler-based instrumentation is that the instrumented code that they generate is much larger than the original code.

Finally, one can also instrument the execution environment such as the Java Virtual Machine (JVM) [6, 26]. The benefit is that the monitoring code is inserted once and can apply to any program executed on this particular execution environment. One can also instrument the operating system [32, 3] to collect statistics during the normal execution of a system. For this, it requires the installation of extra instructions into the kernel to record desired information about the operation of the kernel and the services it provides. This type of instrumentation can keep count of the number of times processes execute, the turnaround time for each process, and the average loading time of the processor. Operating system instrumentation can also make it possible to monitor system state transitions, external interrupts, system calls, as well as data structures such as process control blocks. Thus, this type of instrumentation can be a good choice for time-driven monitoring. Some of the obstacles in OS instrumentation are that the kernel codes, if available, are difficult to debug and sensitive to modification (any error can cause system crash).

2.4. Comparison

In this section, first, we discuss important properties of the execution monitoring techniques. Then, we compare execution monitoring techniques based on these properties. This comparison may be used when one is going to decide among different monitors implementation choices.

It is of a high importance for any execution monitor not to compromise the *security* of the system in which the execution monitoring is performed. One possible metric to measure the security of a system is the size of its trusted computing base.

The trusted computing base (TCB) of a computer system is the set of components in which the occurrence of bugs might put the security properties of the entire system in danger [1]. Rationally, in a system, the smaller the TCB is, the less probable the compromise in security would be. Thus, the best way to assure that a system is secure is to keep its TCB small and simple.

According to the definition, the operation system and the hardware of a computing system are parts of its TCB. In the same way, any new hardware monitor would add to the TCB size. As it is shown in Figure 1, the growth in the size of the TCB is inevitable for all major types of execution monitors regardless of their implementation. Though the size of the expansion of TCB can be different from one type of monitor to another, monitors at the code level (i.e., manual instrumentation, online instrumentation, and instrumentation by compiler) affect the TCB the most as they tend to generate large amount of monitoring codes. The situation is even worse for manual instrumentation as it is done in an ad-hoc manner which makes the whole approach less trustworthy. The instrumentation in language level usually results in smaller monitoring code and results in fewer compromises in security of the system. As mentioned earlier, instrumentation of OS is a difficult and sensitive task as any modification may lead the system to crash.

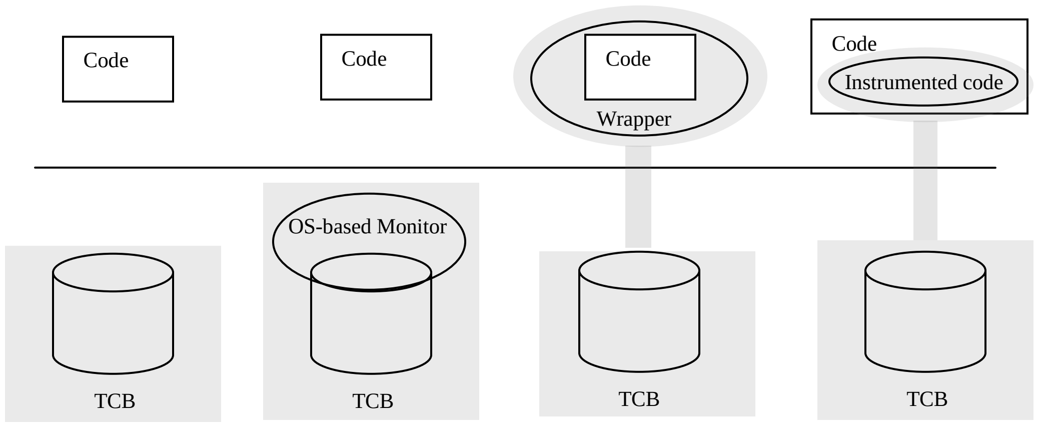

Figure 1. The TCB of execution monitors

We always prefer execution monitors that have low effects on the *performance* of the system. The overhead caused by execution monitors can be used as a performance scale. Hardware monitors usually do not cause any overhead to the system under monitor as they use extra hardware devices. The closer a monitor is to the code level the more compromises in performance we have. This is true for all event-driven monitoring (i.e., manual instrumentation, online instrumentation, instrumentation by compiler, interpreter instrumentation, and OS instrumentation). However, time driven monitoring causes less overhead to the system in comparison with event-driven monitors in the same level.

Intrusiveness is the scale of alteration that monitors may cause to the execution environment of the system [13]. Therefore, one can say a monitor is highly intrusive if it causes direct or indirect alteration to system behavior, timing, and resource consumption. Thus, non-intrusiveness may be aligned to the performance of monitors. The monitors usually need enough access privilege to be able to make a change to the system behavior. One can categorize monitors into two classes based on extensiveness in terms of privileges, namely user-mode monitors and kernel-mode monitors. The kernel-mode monitors (monitoring by OS instrumentation) have more privileges and potentially can make more intrusion. At the same time, if we define intrusiveness as alteration visible to the monitored system, the kernel-mode monitors are less visible and detectable.

When we want to choose between monitor implementation choices, the *cost* of a monitor can always play an important role. The monitors are more expensive at hardware level as they need new and maybe custom made hardware devices. As we move toward code level monitors, the cost of monitors decreases. It should be mentioned that, here, we do not consider the time as a cost factor since it may vary depending on the implementation (though, manual instrumentation tends to be time-consuming most of the times).

We define *flexibility* as combination of modifiability, extendibility, portability, and easiness of monitor removal. Obviously, the hardware monitors are the least flexible monitors as they cannot be easily modified or extended (this usually needs a totally new device) and they are highly machine dependent. The machine dependency decreases as we move towards code level monitoring. The same rule holds for modifiability and extendibility. By contrast, monitor removal is easier when monitors have high integrity and separation from the system under monitor. Thus, hardware monitors and add-on monitors have high easiness of monitor removal. The existence of an automatic monitor removal also increases the flexibility of monitors.

Broadness can be defined as ability of monitors to perform different types of monitoring. Therefore, a monitor has high broadness if it can perform, both, time driven and event-driven monitoring.

It should be noted that to make a good decision between different implementation choices one must weigh between the costs of an implementation choice against the benefits of that implementation choice. A summary of the discussion is shown in Table 1.

Table 1. Comparison between different implementation choices for execution monitoring

	Security	Cost	Flexibility	Intrusiveness	Performance	Broadness
Add-on Monitor	Low	Medium	High	Medium	Medium	Low
Manual instrumentation	Low	Low	Low	High	Low	Medium
Online instrumentation	Low	Medium	Low	High	Low	Medium
Instrumenting compiler	Low	Medium	Medium	High	Low	Medium
Interpreter instrumentation	Medium	Medium	Medium	Medium	Low	Medium
OS instrumentation	Low	High	Low	Medium	Medium	High
Hardware monitor	Medium	High	Low	Low	High	Low

3. Monitoring Challenges for Multimedia Services

Multimedia services and services in general can be composed of smaller services. This enables a service consumer to be service provider at the same time, since a composed service, itself, can be used as a service by other developers in a service composition. This ability to compose services creates challenges to efficient monitoring of services since the internal structure of composite services may be hidden. For example, suppose that we want to compose a new service *CS* as shown in Figure 2. For this, we, as a service requester, can search and locate our required service using a service broker *SB*. The service broker provides a list of services and their descriptions [33].

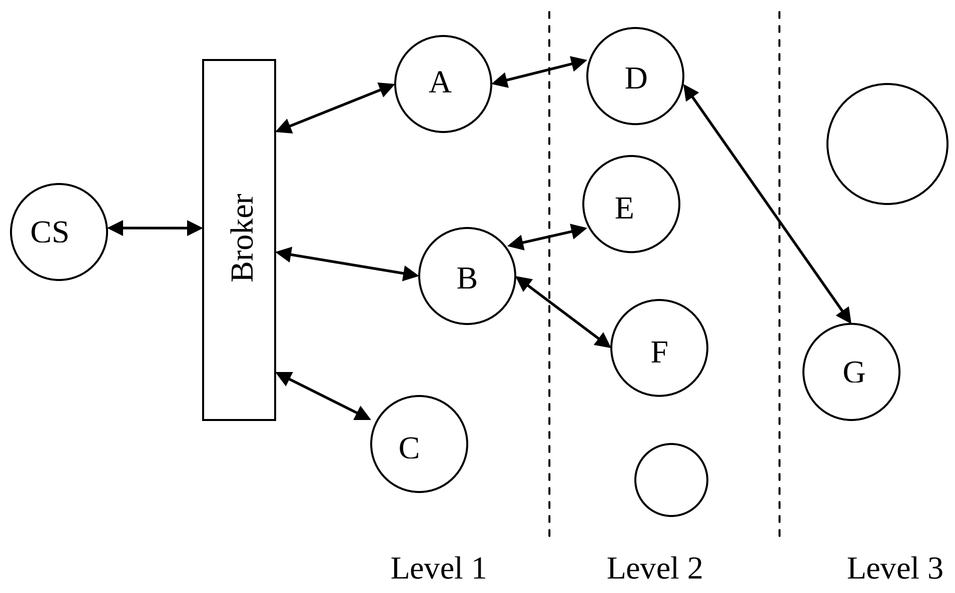

Figure 2. Sample composition of service *CS*

We read the descriptions and choose the services A, B, and C which we think that might suit us better. We do not necessarily know who composed these services because we only interface with the broker BS, and the broker BS itself might only see the suppliers it contracts with directly. A and B are composite services, whereas C has no further subcontract. However, we do not know this because the providers of A and B need to protect their assets following the principle of encapsulation. Jeffery [23] studied the common problems posed by execution monitoring and concluded that there are three main challenges to software monitoring: The first challenging issue is the huge volume of data generated through execution monitoring that should be collected, processed, and presented. The second issue is the intrusion that may be caused due to activities of monitors. That is, monitors may change the execution environment of the system (e.g., by consuming system resources or changing their normal status). The last challenge is to make available an extensive access, often required by monitors, to different aspects of the system being monitored. In this section, we discuss potential difficulties in monitoring service-based systems. Although service-based systems are different from traditional systems (e.g., closed systems managed by single organizations), a number of these difficulties can fit in the three mentioned categories of challenges. In addition, we address some of the new challenges that might be faced due to the distributed nature of service-based systems.

3.1. Behavioral Change

In standalone programs, the output of the program depends only on the inputs to the program. This is not always valid in most service-based systems. In these systems, depending on the race condition among services and synchronization sequences used by services, the output may vary from one system run to another. For this reason, delay in transferring information, stopping or slowing down components caused by software event driven monitoring may change the behavior of the entire system. Communication delays also make it difficult to determine a system’s state at any given time.

3.2. Information Access

One challenge in monitoring web services is to collect enough information. Enough information can be collected in one or more executions. Service providers usually charge consumers per execution. Therefore, this cost must be weighed against the benefit of the information gained.

Event driven monitoring of service-based systems can result in large traces. Abstraction techniques can be used to reduce the size of the traces. However, current trace abstraction techniques [19, 10, 20, 36] depend on statistical information gained through static analysis of the source code of the system. In service-based systems the access to the source code of services may be partial, if at all possible. One way to ease the problem is to gain approximate statistical information using dynamic analysis instead of static analysis. However, this could be possible only if we do not have problems with collecting adequate information.

Also, even if we can inspect the data flows between our composed service *CS* and the services in Level 1, the release of information by providers in other levels of the network could not be guaranteed.

3.3. Timing

Services can interact in different styles including synchronous and asynchronous messaging, remote procedure calls, etc. In synchronous messaging, the service consumer has to wait for the service operation respond. In an asynchronous mode, the service consumer does not wait for a response from the service operation, although, when the processing of the service operation is complete a callback from the service to the consumer is used. In, both, synchronous and asynchronous messaging, the delay in response is important (e.g., there is a deadline for receiving a response). Monitoring, as mentioned earlier, can slow down the monitored system, component, or service. Therefore, it may result in out-of-date responses.

3.4. Synchronization of Event Traces

Chronological order of events in trace is usually of a high importance to detect major phases of a system. In a single-processor system, the event can be stamped using the system clock. This can be difficult to achieve for service-based systems since each service has its own clock. For this reason, it is hard to determine if an event with a later timestamp took place after an event with an earlier time stamp. One not so easy way to overcome this problem is to update time stamps when merging traces. For this, the events at each individual service must be time stamped when the trace is generated. Then, traces from all services should be merged according to a common reference to order them.

3.5. Distributed Monitoring

Unlike single-processor systems, where only one event of interest may occur at a time, many events can occur at the same time in a service-based system. Therefore, in order to be able to monitor such systems each service should be monitored. Each service, then, should be monitored as an independent system with one or more processors.

3.6. Trace Transmission

When an event trace of an individual service is being generated it can be stored and then sent to a center for merging traces. The main issue with storing and then sending traces is that they require huge storage space in the computing system which is implementing that individual service. Another problem which might arise is that using the same network to send event traces may affect the behavior of the service-based system due to an increase in network traffic. One way to solve the storage problem is to send an event trace in an online fashion while the trace is being generated. One possible side effect to this approach is that it may increase the complexity of merging traces. To overcome the network problem one may suggest monitors to use a network other than the one used by the service. A different network is an extra cost that must be weighed against its benefit.

4. Conclusion

There is strong interest in the field of execution monitoring and tracing due at the increasing complexity of today’s software systems that tend to be distributed, multi-tier, and service-based. Monitoring can help in various software engineering activities including profiling, testing and debugging, performance evaluation, program optimization, detection of software failure, malicious code detection, diagnosis, and design recovery.

In this paper, we have discussed the enabling technologies of monitoring and tracing techniques with a focus on time and event driven monitoring techniques. We also presented the important properties of execution monitors, their advantages and disadvantages, along with a detailed comparison.

With rapid growth of using multimedia services and distributed systems in the age of Internet, we argued that new research lines would be concerned with the monitoring of service-based systems. To support this, we reviewed and reported on key issues in this area that must be addressed in order to enable efficient analysis of such systems using monitoring and tracing techniques.

5. References

[1] Department of defense standard, department of defense trusted computer system evaluation criteria, 1985.

[2] Debugging blue screens. Technical paper, Compuware Corporation, September 1999.

[3] S. Agarwala and K. Schwan. SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring. In 26th International Conference on Distributed Computing Systems (ICDCS), Lisboa, Portugal, July, 2006.

[4] T. Ball. The Concept of Dynamic Analysis. LECTURE NOTES IN COMPUTER SCIENCE, pages 216–234, 1999.

[5] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 305–314. ACM New York, NY, USA, 2005.

[6] H. D. Bocker, G. Fischer, and H. Nieper. The enhancement of understanding through visual representations. SIGCHI Bull., 17(4):44–50, 1986.

[7] P. Calingaert. System performance evaluation: survey and appraisal. Communications of the ACM, 10(1):12–18, 1967.

[8] J. Calvez and O. Pasquier. Performance Monitoring and Assessment of Embedded HW/SW Systems. Design Automation for Embedded Systems, 3(1):5–22, 1998.

[9] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. Software Visualization, pages 151–162, 2002.

[10] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization.

[11] N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. Software Engineering, IEEE Transactions on, 30(12):859–872, 2004.

[12] M. Ducass´e and J. Noy´e. Tracing Prolog programs by source instrumentation is efficient enough. The Journal of Logic Programming, 43(2):157–172, 2000.

[13] Aral, Z. and Gertner, I. Non-intrusive and Interactive Profiling in Parasight. In Proceedings of the ACM/SIGPLAN PPEALS 1988, pages 21–30, September 1988.

[14] U. Erlingsson and F. Schneider. SASI enforcement of security policies: a retrospective. In Proceedings of the 1999 workshop on New security paradigms, pages 87–95. ACM New York, NY, USA, 1999.

[15] D. Evans and A. Twyman. Flexible policy-directed code safety. In Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on, pages 32–45, 1999.

[16] A. Gallo and R. P. Wilder. Performance measurement of data communications systems with emphasis on open system interconnections (osi). In ISCA ’81: Proceedings of the 8th annual symposium on Computer Architecture, pages 149–161, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[17] A. Geist, T. Heath, B. Peyton, and P.Worley. A User’s Guide to PICL: A Portable Instrumentation Communication Library. Technical report, TR TM-11616, Oak Ridge National Lab. 1990.

[18] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. IEEE Trans. Softw. Eng., 16(2):197–211, 1990.

[19] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In Proc. 14th Int. Conf. on Program Comprehension (ICPC), pages 181–190.

[20] A. Hamou-Lhadj and T. Lethbridge. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on, pages 559–568, 2005.

[21] R. Henry, K. Whaley, and B. Forstall. The University ofWashington illustrating compiler. ACM SIGPLAN Notices, 25(6):223–233, 1990.

[22] R. Jain. The art of computer systems performance analysis. Wiley, 1991.

[23] C. Jeffery. Monitoring and Visualizing Program Execution: an Exploratory Approach. 1993.

[24] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In WCRE ’97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE ’97), page 56, Washington, DC, USA, 1997. IEEE Computer Society.

[25] A. Karush. Two approaches for measuring the performance of time-sharing systems. In Proceedings of the second symposium on Operating systems principles, pages 159–166. ACM Press New York, NY, USA, 1969.

[26] A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors. SIGPLAN Not., 26(6):338–352, 1991.

[27] M. Linton. The Evolution of Dbx. In Proceedings of the Summer USENIX Conference, pages 211–220, 1990.

[28] H. Lucas, Jr. Performance evaluation and monitoring. ACM Comput. Surv., 3(3):79–91, 1971.

[29] A. D. Malony. Multiprocessor instrumentation: approaches for cedar. pages 1–33, 1989.
[30] B. Mohr. Performance evaluation of parallel programs in parallel and distributed systems. In Proc. CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing, Zurich, Lecture Notes in Computer Science 457, pages 176–187. Springer, 1990.
[31] R. Oechsle. Automatic visualization with object and sequence diagrams using the java debug interface (jdi). In Proceedings of the Software Visualization: International Seminar, pages 176–190. Dagstuhl Castle, Germany, Springer-Verlag Heidelberg, 2001.
[32] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. SIGOPS Oper. Syst. Rev., 19(5):15–24, 1985.
[33] W. T. Tsai, "Service-Oriented System Engineering: A New Paradigm," IEEE International Workshop on Service-Oriented System Engineering (SOSE), Beijing October 2005, pp. 3 - 8.
[34] B. Plattner and J. Nievergelt. Special Feature: Monitoring Program Execution: A Survey. Computer, 14(11):76–93, 1981.
[35] J. Robbins. Debugging windows based applications using windbg. Miscrosoft Systems Journal, 1999.
[36] A. Rountev and B. Connell. Object naming analysis for reverse-engineered sequence diagrams. In International Conference on Software Engineering: Proceedings of the 27th international conference on Software engineering, volume 15, pages 254–263, 2005.
[37] R. Snodgrass. A relational approach to monitoring complex systems. ACM Transactions on Computer Systems (TOCS), 6(2):157–195, 1988.
[38] T. Systa. Dynamic Reverse Engineering of Java Software. LECTURE NOTES IN COMPUTER SCIENCE, pages 174–174, 1999.
[39] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In IEEE Symposium on Security and Privacy, 2006.
[40] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE SECURITY & PRIVACY, pages 32–39, 2007.

[41] A. Zaidman. Scalability Solutions for Program Comprehension through Dynamic Analysis. In Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, 2006.
[42] John Simpson and Edmund Weiner, editors. Oxford English Dictionary Additions Series, volume 2. Clarendon Press, 1993.
[43] Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T. (2005). Recovering behavioral design models from execution traces. In Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pages 112–121. IEEE Computer Society.