Non-Intrusive Transaction Monitoring Using System Logs

Bikram Sengupta(*), Nilanjan Banerjee(*), Animashree Anandkumar(†), Chatschik Bisdikian(‡)

(*)IBM India Research Lab
New Delhi 110070, India
{bsengupt, nilanjba}@in.ibm.com

(†)ECE Department, Cornell University
Ithaca, NY 14853, USA
aa332@cornell.edu

(‡)IBM T. J. Watson Research Center
Hawthorne NY 10532, USA
bisdik@us.ibm.com

Abstract— We consider the problem of online monitoring of transaction instances in enterprise environments, based on footprints left by the instances in system logs and using a state-based reference model of the transaction. Unlike existing approaches, we do not rely on any platform-specific knowledge, neither do we assume footprints to carry correlating identifiers, as injected through instrumentation. We outline a solution for tracking transaction instances at individual and aggregate levels, present preliminary results on theoretical analysis of monitoring precision and conclude with directions of ongoing and future research.

Keywords-transactions; monitoring; log files;

I. INTRODUCTION

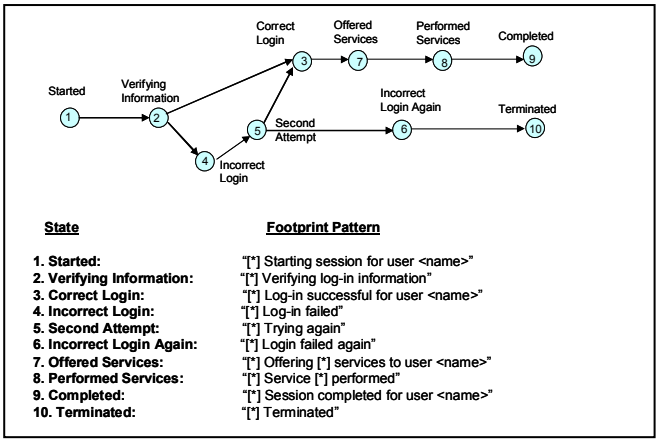
Consider a travel company that starts off by offering attractive bargains on domestic air tickets to customers through its website. It has a small IT team managing a few custom-written applications running on a handful of servers. As its customer base grows, the company decides to expand its offerings and starts selling international air tickets as well, and adds several new applications to their IT suite. With increasing market share over the years, it acquires a couple of competitors with their IT infrastructure, and positions itself to manage travel end-to-end, by not only arranging air tickets, but ground transportation and hotel accommodation as well. All this while, the once small and manageable IT infrastructure continues to grow and eventually becomes a complex mix of numerous legacy applications patched together with newly added software components, running on several hundred servers and databases, and integrated with a host of third-party products. The IT staff has increasingly less visibility into the infrastructure they are supposed to keep up and running, and they struggle with customer complaints of failed or time-consuming transactions. Existing transaction monitoring products are considered but these are either platform specific and able to handle only a subset of the infrastructure, or require large-scale instrumentation, which the IT staff finds to be an unacceptable proposition in their environment due to licensing issues. Ultimately they resort to manual collation of information spread across a variety of infrastructure elements and applications, which is a time-consuming and labor-intensive task, and essentially reactive in nature.

Such scenarios are common in the complex enterprise environments of today, and consequently, ample motivation exists for a *lightweight, non-intrusive* and *general-purpose* management solution that is easy to deploy and can track transactions as they execute in an enterprise environment, end-to-end, while imposing minimal requirements on the underlying applications and infrastructure. Such a solution should ideally (a) help discover and model transaction flow in the deployed environment and (b) monitor the environment to automatically draw conclusions regarding the status of transactions. Existing tools usually support these activities through instrumentation and/or platform-specific knowledge. However, we take a platform-agnostic approach and focus on environments where the only information available are *footprints* left by transaction instances in system logs, which may be custom-written and not necessarily generated by any specific middleware. Moreover, we do not assume these footprints to carry transaction identifiers (as injected through instrumentation) which would have allowed a footprint to be mapped to a specific transaction instance. While our larger research effort includes (semi-) automated discovery of transaction models from historical logs, in this paper we focus on the problem of online monitoring of transaction instances by observing footprints that are created in various system logs, using a state-based model of the transaction. In particular, our contributions are two-fold: (i) we propose a monitoring solution outline for tracking ongoing transaction instances at individual and aggregate levels and (ii) we present preliminary results of theoretical analysis that explores the precision bounds with which footprints coming from the same transaction instance may be correlated.

The rest of the paper is organized as follows: In Section II, we introduce our notion of transaction models, and briefly discuss how such models may be obtained. Section III presents an overview of our monitoring solution, while theoretical analysis preliminaries are introduced in Section IV. Section V cites related work, while Section VI discusses directions for ongoing and future work.

II. TRANSACTION MODELS

Transaction models may be obtained through log analysis techniques and domain-expert involvement. In our approach, historical log records carrying footprints of the transaction of interest are first analyzed to detect structurally similar log records. Such log records are then mapped to a common abstract form, which we call a *footprint pattern*.



State	Footprint Pattern
1. Started:	"[*] Starting session for user <name>"
2. Verifying Information:	"[*] Verifying log-in information"
3. Correct Login:	"[*] Log-in successful for user <name>"
4. Incorrect Login:	"[*] Log-in failed"
5. Second Attempt:	"[*] Trying again"
6. Incorrect Login Again:	"[*] Login failed again"
7. Offered Services:	"[*] Offering [*] services to user <name>"
8. Performed Services:	"[*] Service [*] performed"
9. Completed:	"[*] Session completed for user <name>"
10. Terminated:	"[*] Terminated"

Fig1. Example ATM Transaction Model

Next, sequences of patterns that seem to occur frequently are discovered, and these lead to sequences of "states", where a state corresponds to a footprint pattern. Finally, the domain experts inspect these state machines, add/delete states as needed, label states with the activity names they correspond to, logically compose separate sub-models, and incorporate additional information in the footprint patterns when available (e.g. indicating that there is a variable or token of interest in the pattern). This yields the final transaction model that is used during monitoring. While the details of our model discovery approach are beyond the scope of this paper, we adapt techniques from a rich body of related work cited in Section V.

A. Example

To illustrate the notions of states and footprints, we present a simple example of a transaction model. The model, shown in Fig.1, represents the possible interactions of a user with an Automated Teller Machine (ATM). State 1 ("Started") corresponds to the initiation of a new session for a user. First, the login credentials supplied by the user are verified (State 2), and in case of correct login (State 3), a set of banking services are offered to the user (State 7), certain selected services are performed (State 8), and the session completes successfully (State 9). In case of an incorrect login (State 4), the user is permitted a second login attempt (State 5). If this is successful, then the interaction follows states 7 through 9 as before. However, if the second login attempt is again unsuccessful (State 6), then the session is terminated (State 10).

Fig. 1 also shows example footprint patterns corresponding to each state. The footprint pattern for a state may contain wildcard characters "[*]" denoting arbitrary strings, timestamps etc. and may also carry placeholders for tokens. For example, <name> in the footprint pattern of state 1 represents the name of the user as recognized from the ATM card. The presence of such tokens cannot usually be detected through automated log file analysis, but may be indicated by domain experts. Moreover, footprints may not carry any such token at all e.g. the footprint patterns of states 2, 4, 5 etc. are devoid of tokens.

III. MONITORING TRANSACTIONS

We will now discuss challenges in monitoring transaction instances based on their footprints. Next, we will present a solution outline for monitoring ongoing transaction instances at individual and aggregate levels, using their footprints and a model of the transaction.

A. Tracking Transaction Progress Via Footprints

Given a model of the transaction we can use it to track the progress of transaction instances by observing their footprints. More specifically, we can match log records as they appear in different logs with the expected footprint patterns of different states in the transaction model. In case of sequential execution of transaction instances, this is simple to do; a new log record must have been written by the current transaction instance, and the state, whose footprint pattern matches the log record, is the current location of the instance. However, transaction monitoring becomes more challenging when several transaction instances execute concurrently, with interleaved log record footprints. It has been our experience that such footprints - particularly in legacy systems, or where instrumentation is not feasible - frequently do not carry any transaction identifier; at most, they may carry a token representing an item of interest e.g. line-of-business name, order number etc. However, these tokens may not be unique to one transaction instance, may not appear in every footprint of an instance, or may get transformed as the transaction instance flows from one application to another. For these reasons, when multiple transaction instances execute, we may not know which specific instance has produced a particular footprint. In such cases, our solution identifies the *candidate transaction instances* that may have produced a log record footprint and tries to narrow down the "possible" states of these instances to a localized part of the model. For large and complex transaction models, this carries significant benefits in terms of understanding transaction flow, performance, and debugging.

There are situations however, when the overall distribution of instances over the different states of the transaction model is of more interest that the state of a particular instance. In such cases, we may compute bounds on the number of instances in the different states. Such information may drive dynamic resource allocation and load balancing strategies.

B. Basic Approach and Supporting Architecture

Let us first consider a sequence of log records corresponding to the ATM model in Fig.1 to informally explain our proposed approach. Suppose we have a record "Starting session for user John", then a new instance, say $T1$, must have started, with token *name=John*. Next, if a similar record comes in for user Peter, then we can conclude that another instance, say $T2$, has started. Both instances are now in state 1. If log record "Verifying login information" corresponding to state 2 arrives next, both $T1$ and $T2$ are *candidates* for it; thus the possible states of $T1$ (as also $T2$) are states 1 and 2. If another instance of "Verifying login information" arrives, then two instances of this record have been received so far, and there are two candidate instances. Since each instance can reach state 2 only once (i.e. state 2 is not part of a cycle), both $T1$ and $T2$ must have reached state 2, and state 1, which precedes state 2, is removed as a possible state. Next, if a footprint "Log-in successful for user Peter" corresponding to state 3 is received,

the user name indicates that this log record was produced by $T2$. Hence, the possible state of $T2$ is updated to state 3. The overall idea is to use a combination of execution history (e.g. number of footprints received, and candidate instances for the same) and available token information to identify the possible states of ongoing transaction instances.

To monitor transactions at aggregate level, we can assign a pair of counters $[s_{min}, s_{max}]$ to each state s in the model, where s_{min} and s_{max} denote the minimum and maximum number of transaction instances currently in state s, respectively, and are both initialized to 0. Suppose at some point during execution, we have the following counts; state 1: [4, 4], state 2: [2, 2], state 3: [5, 5] and state 5: [3, 3]. If we now get a log record "Verifying log-in information" corresponding to state 2, it indicates that (a) a transaction instance has moved to state 2, so its count is updated to [3,3] and (b) since state 2 has only one immediate predecessor in state 1, the instance must have moved from state 1 to state 2, so the count of state 1 is reduced to [3,3]. On the other hand, if we next get a log record corresponding to state 3, we see that it has two immediate predecessors in state 2 and state 5, both of which have some instances. In this case, we cannot tell if the log record has been written by an instance moving from state 2 to state 3, or by an instance going from state 5 to state 3. So, while we increase the counts of state 3 to [6, 6], we can only reduce the lower bounds of both state 2 and state 5; thus the new bounds are state 2: [2,3] and state 5: [2, 3].

Our proposed system architecture for supporting transaction monitoring at individual and aggregate levels, as illustrated above, is shown in Fig. 2. The main components are (a) a set of probes and a probe manager (b) the monitoring engine and (c) a set of agents.

A probe is associated with each log file where a transaction footprint may be recorded. It reads records as they appear in the log, and forwards the same to a probe manager. Assuming that each log record carries a timestamp, the probe manager sorts records received from various probes according to the record timestamps, and forwards a sorted sequence of log records to the monitoring engine.

The monitoring engine maintains a list of ongoing transaction instances, and the states they expect next (the *next expected states*). When the engine receives a log record that corresponds to an initial state of the transaction model, it records a new transaction instance whose next expected states are those that immediately follow the initial state, and creates an agent that is henceforth responsible for tracking the progress of this instance. The engine also binds tokens, if present in footprint patterns, to their concrete values as obtained from matching log records. If the engine receives a log record that corresponds to some other (non-initial) state of the model, then it scans the list of ongoing instances to determine which candidate instances may have produced it, by considering the next expected states of these instances, and their token values. The agents corresponding to the candidate instances (called candidate agents) are forwarded the log record, along with information about all the candidate instances for this record.

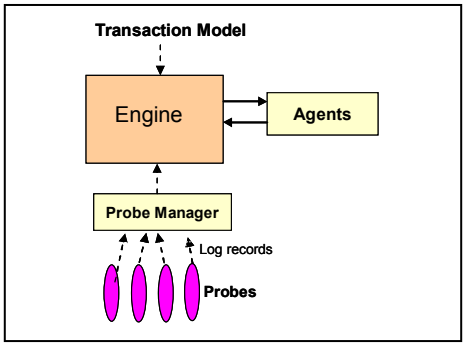


Fig2. Monitoring System Architecture

An agent keeps track of the current possible states of its associated transaction instance. If an agent receives a log record corresponding to state s, for which it is the only candidate, then its instance must be in state s. Otherwise, s is recorded as a *possible* state, and for each possible state, an agent maintains a count of the number N_1 of footprints of that state witnessed by it, and the cumulative set of candidate instances A that may have produced these footprints. When $N_1=A$ for a state s that is not part of a cycle, the agent concludes that the instance has definitely reached state s, and accordingly updates the set of possible states, by removing states that precede s, or are unreachable from s. The next expected states (containing any state that follows a current possible state) are then determined, and updated at the engine. Subsequently, the engine reads the next record, and the cycle repeats. The engine also monitors aggregate count of instances at each state of the model. On reception of a log record corresponding to state s, the engine increments s_{min} and s_{max}, and decrements the bounds of the parent(s) of s as illustrated in the example above.

IV. PRECISION MODELING PRELIMINARIES

In this section, we highlight some of the theoretical challenges related to this study. Specifically, given a set of interleaved footprints generated by several ongoing transaction instances, we would like to investigate the theoretical limits on the precision with which we can correlate footprints coming from the same instance.

Fig. 3 shows a simple system comprising of two states, a *start* state and a *finish* state. We assume that log records contain information on when transaction instances start being processed (arrivals at times $y_0(1), y_0(2),\ldots$ to state S_0) and when they finish (arrivals at times $y_1(1), y_1(2),\ldots$ to state S_1). However, due to absence of transaction identifiers, we have no knowledge of which of the instances have departed or in what order. We say that an *n-match* occurs whenever we pick n consecutive arrivals and n departures (not necessarily consecutive) and the latter "happen" to be the departures of the n arrivals picked. We consider n-matches that correspond to *busy periods* of the system, i.e., when the first of the arrivals

arrive to an empty system and the n-th departure leaves the system empty again. It should be obvious that there is no need to attempt matches across busy periods. The system in Fig 3, and the preliminary results shown next, serve as the basis for analysis of more elaborate system we currently study.

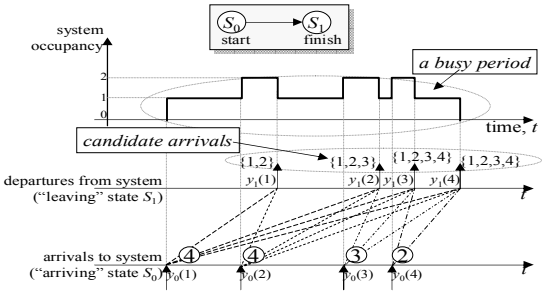


Fig3. Arrivals and departures in a 2-state system

If instances were processed in FIFO or LIFO order, then matching would have been trivial. However, when the only information available is the *size* of a busy period, i.e., the number n of instances processed during the busy period, then any departure could be due to any of the arrivals in the busy period. Thus, the number N_m of possible n-matches equals the number of permutations of n objects ${}_nP_n=n!$. If, in addition, timestamps are also available, then causality arguments can be used to reduce the number of departure choices (or permutations), e.g., in Fig. 3 the fourth arrival (at time $y_0(4)$) can only be associated with the third and fourth departures (at times $y_1(3)$, $y_1(4)$, respectively); the circled numbers at each the arrival instance in the figure represents the number of candidate departures for the arrival. If we were to write the biadjacency (0-1) matrix $A=\{a_{ij}=1$, iff departure j occurs after arrival $i\}$, then the number of possible n-matches N_m "reduces" to:

$$1 \le N_m = \sum_{\pi_n} \prod_{i=1}^{n} a(i, \pi_n(i)) \le n!, \qquad (1)$$

where $\pi_n(i)$ is the i-th element of a permutation of the set {1, 2, …, n}. The equality in the RHS of (1) can be attained when *all* departures in the busy period occur after *all* arrivals to the busy period have occurred. The equality on the LHS is trivially attained when the size of the busy n equals 1.When stochastic information is also known in terms of the distribution $f_T(t)$ of the residence time of an instance in state S_0, then a most likely match can be selected. Due to space limitations, we do not elaborate further on this here, however, it can be shown, as expected, that (1) plays an important role in this case too and aids in deriving performance bounds.

V. RELATED WORK

Log Analysis and Model Discovery: [1] presents techniques for discovering similar records in a log and clustering them to simplify log complexity. The idea of process mining in the context of workflow management was introduced in [2]. The ProM framework [3] integrates a set of tools that support different mining techniques for model discovery. A survey of such techniques may be found in [4].

Distributed System Monitoring: *White box* methods for monitoring generally depend on instrumentation techniques e.g. [5], [6]. Some commercial offerings having transaction monitoring capabilities are tied to specific technologies like SOA [10] and make use of product/domain-specific information and ARM instrumentation [9] to track call status. In contrast, our solution is platform-agnostic and non-intrusive. There are *black-box* techniques e.g. [7], [8] which require no instrumentation; however, they generally do not focus on monitoring individual call flows end-to-end, but rather on the discovery of component-level dependency models.

VI. ONGOING AND FUTURE WORK

We have implemented the transaction monitoring solution proposed in this paper, and have carried out some empirical studies, details of which could not be provided due to lack of space. An extended version of the theoretical study in section IV can be found in [11]. Our current work focuses on run-time transaction model validation. In future, we would like to incorporate fault-tolerance capabilities into the solution.

Acknowledgements: We wish to thank Paul Klein, Shoel Perelman, Dakshi Agrawal and Dinesh Verma for their support, Paul Hurley for his complementary work (with Chatschik) on transaction model discovery, and Arun Kumar for fruitful discussions on transaction monitoring.

REFERENCES

[1] Logfile Clustering Tool. http://kodu.neti.ee/~risto/slct/

[2] R. Agrawal, D. Gunopulos and F. Leymann. Mining Process Models from Workflow Logs. Int'l Conf. on Extending Database Technology. LNCS vol. 1377, 1998.

[3] PROM. http://is.tm.tue.nl/~cgunther/dev/prom/

[4] W.M.P van der Aalst, B.F. van Dongen et al. Workflow Mining: A Survey of Issues and Approaches. Data and Knowledge Engineering, 47(2):237-267, 2003

[5] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: problem determination in large, dynamic internet services", in *Proc. of DSN* 2002, pp. 595-604.

[6] M. Schmid, M. Thoss et al. A Generic Application-Oriented Performance Instrumentation for Multi-Tier Environments. Integrated Network Management, 2007: 304-313

[7] M. Agarwal, M. Gupta, G. Kar, A. Neogi and A. Sailer. Mining Activity Data for Dynamic Dependency Discovery in e-Business Systems. IEEE eTransactions on Network and Service Management Journal (eTNSM), Fall 2004

[8] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, A. Muthitacharon, "Performance debugging for distributed systems of black boxes", in *Proc. of 19th ACM SOSP* 2003, pp. 74-89.

[9] ARM. http://www.opengroup.org/tech/management/arm/

[10] ITCAM for SOA. http://www-306.ibm.com/software/tivoli/products/composite-application-mgr-soa/

[11] A. Anandkumar, C. Bisdikian, and D. Agrawal, "Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints," *IBM Research Report RC24422*, Nov. 2007.