Transaction Dependency Graph Construction using Signal Injection

Shuyi Chen[†] Kaustubh R. Joshi[†] Matti A. Hiltunen[‡]
William H. Sanders[†] Richard D. Schlichting[‡]

[†]Information Trust Institute
Coordinated Science Lab. & Electrical and
Computer Engineering Dept.
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{schen38,joshi1,whs}@crhc.uiuc.edu

[‡]AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{hiltunen,rick}@research.att.com

***Abstract*— Understanding the runtime behavior and dependencies between components in complex transaction-based enterprise systems enables the system administrators to identify performance bottlenecks, allocate resources, and detect failures. This paper introduces a novel method for extracting dependency information between system components at runtime by using delay injection on individual links and Fast Fourier Transforms. Our proposed method introduces minimal disturbance in the system and its execution time is independent of the system workload. Thus, it can be used at runtime in production systems. Furthermore, it avoids false positives introduced by other methods. We present preliminary experimental results that demonstrate that our approach is able to identify dependencies, avoid false positives, while ensuring low perturbation to the target system.**

I. Introduction

Large transaction-based distributed systems are becoming increasingly complex, and understanding their runtime behavior is crucial for system administration and resource planning. *Transaction dependency graphs* (TDGs) are a technique for representing dependencies between components in a distributed system. An edge from node A to node B in a TDG indicates that a message sent from component A to component B will be received and processed by B before the transaction at A can return to the originator. In this paper, we propose a new method for determining TDGs for transaction-based systems. The proposed approach uses delay injection on communication connections between individual components to perturb the system slightly at runtime. The system response times are then analyzed using Fast Fourier Transform (FFT) to identify dependencies.

Our approach has significant advantages compared to the prior work. In contrast to [2], [3], and [4], the execution time of our analysis algorithm is independent of the application payload (application messages) and controllable by the administrator. Thus, the algorithm can be deployed in running production systems. Furthermore, our approach avoids false dependency edges that are possible in the causal graphs generated by those techniques. In contrast to [1] and [5] that inject faults into the system and cannot feasibly be used on production systems, our method injects only a small amount of delay in network packets. The delay is a small fraction of the original response time and is usually less than the standard deviation of the noise present in the original response time, thus allowing our technique to be used even when a system is in operation.

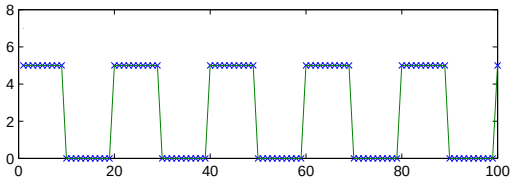


Fig. 1. Delay injection pattern and sampling

Our method discovers the existence of a dependency between two components by injecting a small amount of delay on packets exchanged between these two components (specifically, between socket endpoints) at a fixed frequency. The delay is injected using a square wave. To determine whether a link is a dependency edge for a certain transaction TS, we setup a special monitor client that issues periodic requests to transaction TS at frequency f and measures their response times. By the *Nyquist* theorem, f must be at least twice the delay injection frequency at the link. Finally, we take the collected response times as a time series and use FFT to transform it from the time domain to the frequency domain. We use the distribution in the frequency domain to determine if a link is a dependency edge for transaction TS. If a spike is observed at the frequency where we injected the delay, we conclude that the link is a dependency edge. Otherwise, the link will not be in the graph. Because the choice of frequency is a free variable, our approach can choose frequencies which do not have significant components in the original response time waveform. Figure 1 shows how we inject delay and sample response time. In the figure, the solid line is the delay injection pattern. A delay of 1 ms is injected into all messages

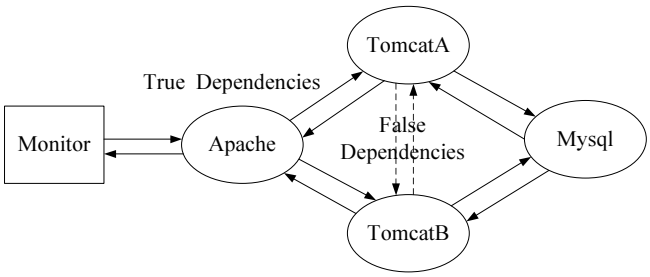
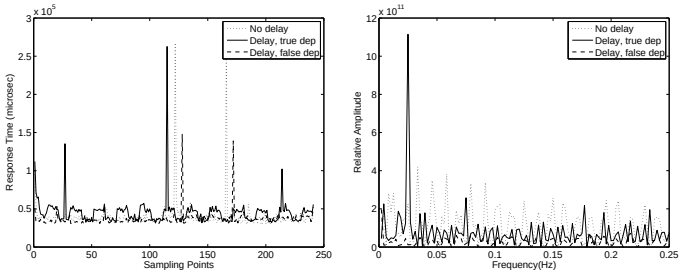


Fig. 2. Experiment Setup



(a) Original response time series (b) Power spectrum

Fig. 3. Results of delay injection in RUBiS (time and frequency domain)

between two targeted components for a period of 10 seconds, and then disabled for 10 seconds in a square wave pattern (of 0.05Hz). We send test requests from a monitor client to the system every 1 second (frequency of 1Hz), and calculate the response time. The sampling points are shown as x-marks in the figure.

II. Implementation

We inject delay on links by implementing an interposition library using the **LD_PRELOAD** variable in Linux. Our library interposes the C library's system call wrapper to log all the application socket calls. Dependency extraction is divided into two phases. In the first phase, the interposition library on each machine logs all the outgoing links from the machine. The links extracted are the set of *possible* dependencies. In the second phase, for each input transaction type *TS*, we loop through the links in the possible dependencies set. For each link, we issue *TS* from the monitor at some frequency f for a period of time, and log the response times. In the meantime, we use the interposition library to inject a square-wave delay on that link, and use the power spectrum of the measured response time to determine whether the link should be added to the TDG or not.

III. Experiments

We carry out experiments on a Java servlet based E-Bay like auction system application called **RUBiS** using the configuration shown in Figure 2. The Apache server does load balancing between the Tomcat servers, which also share session state via replication. During the first phase of our algorithm, we discovered all the possible communication links shown by the arrows in the figure. However, the links between the Tomcat servers are false dependencies that represent session state sharing that Tomcat performs asynchronously for every request. In the delay injection phase, we show results for only the Bid transaction. The 10ms, 0.025Hz delay we inject is small compared to the mean transaction response time of 40ms. The monitor sends the bidding request at a frequency of 0.5Hz. The algorithm issues 240 sampling requests for each link.

We present results for 1) no delay injection, 2) delay injection on a true dependency link from **TomcatA** to **Mysql**, and 3) delay injection on a false positive edge from **TomcatA** to **TomcatB**. Figure 3(a) shows the observed response time for all three cases. The difference between the mean response time with and without delay injection is around 5ms, which only accounts for 11% of the original response time. Figure 3(b), shows the power spectrum of the response time for all three cases. In the graph, we can observe a spike at frequency 0.025Hz, which is our injection frequency on the true dependency edge. In contrast, the spectrums corresponding to no delay injection and delay injection on a false positive edge are statistically similar. The results show that our approach is able to identify the true dependencies in the systems and avoid any false dependencies while injecting only minimal perturbation into the running system.

IV. Conclusion and Future Work

In this abstract, we propose a new method for extracting the TDG for distributed transaction-based computing systems. Our method uses an interposition library to collect information and inject packet delays. We use Fast Fourier Transforms to identify dependencies. Our preliminary results are computed on an unloaded system with very little noise in the response time. Our next step is to study a system with real workload. We will explore different filtering techniques to smooth out noise and answer the question of how much delay we need to inject in order to extract a dependency edge reliably. Also, incremental construction of the TDG will be explored, with the prior TDG and current link information used as inputs to the next run of the algorithm. Only new links, deleted links, and links that have changed over time are selected as candidates for the algorithm. Finally, we will also exploit the TDGs to detect failures, and direct recovery, load balancing, and process migration in the system.

References

[1] A. Brown et al. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *The Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[2] M.K. Aguilera et al. Performance debugging for distributed systems of black boxes. In *Symposium on Operating Systems Principles*, 2003.

[3] P.Reynolds et al. Wap5: black-box performance debugging for wide-area systems. In *International World Wide Web Conference*, 2006.

[4] S. Agarwala et al. E2eprof: Automated end-to-end performance management for enterprise systems. In *International Conference on Dependable Systems and Networks*, 2007.

[5] S. Bagchi et al. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *Decision Support for Operations & Maintenance*, 2001.