Dynamic Adaptation of Temporal Event Correlation Rules

Rean Griffith*, Gail Kaiser*, Joseph L. Hellerstein** and Yixin Diao**

*Columbia University Department of Computer Science
New York, New York
{kaiser, rg2023}@cs.columbia.edu

**IBM Thomas J. Watson Research Center
Hawthorne, New York
{hellers, dias}@us.ibm.com

Abstract

*Temporal event correlation is essential to realizing self-managing distributed systems. Autonomic controllers often require that events be correlated across multiple components using rule patterns with timer-based transitions, e.g., to detect denial of service attacks and to warn of staging problems with business critical applications. This short paper discusses automatic adjustment of timer values for event correlation rules, in particular compensating for the variability of event propagation delays due to factors such as contention for network and server resources. We describe a corresponding **Management Station** architecture and present experimental studies on a testbed system that suggest that this approach can produce results at least as good as an optimal fixed setting of timer values.*

1. Introduction

Temporal event correlation across event streams from multiple sources will be a key component of self-managing distributed systems. For example, a denial of service attack might be detected by correlating failed logins on multiple machines in the same enclave within a very short period of time, and problems with multi-server applications can sometimes be detected by analyzing anomalous variations in interval times between processing stages that occur on different servers. In such applications, it may be necessary to *dynamically adjust* the time-interval values used in the correlation pattern rules, to minimize both false alarms and missing (undetected) alarms. One reason is that preset timer values are, in many cases, inherently 'fuzzy' heuristic thresholds – in which case automatic dynamic adjustment could potentially be organized according to a reinforcement learning approach, another level of feedback control loop. Another reason for dynamic adjustment is the variability in event propagation delays due to contention for network and server resources and other performance-related factors, which is the main focus of this short paper. We have developed an architecture and an adaptive control algorithm that dynamically compensate for variations in propagation delays, without reliance on a synchronized global clock (although essentially the same approach could also useful for compensating for 'fuzziness' in rule timer specifications even when events carry authoritative timestamps). Our empirical studies suggest that our prototype implementation produces results at least as good as theoretically optimal but fixed settings of timer values.

We assume here that the events to be correlated are specified as part of if-then rules interpreted by an engine in a **Management Station**. The if-part of these rules consists of a temporal event pattern and the then-part prescribes an action to be taken, such as invoking an automated effector [5] or a concerted workflow of such effectors [4], updating a management console display, or paging an administrator. Our focus here is on the if-part. The events considered in the if-part of a given rule

may come from a combination of different (sub)systems or components within the managed target system, and are generally bound to variables in the if-part that are then used to parameterize the then-part. Consider the simplified examples below in which the question marks indicate variables. (In our implementation, rules are written in a more cryptic XML-based notation [6, 2]; other correlation specification models would work as well, such as SMART's 'codebook' approach [8].)

- Rule 1: If there is no `Heartbeat` event from system ?S1 at location ?L1 within 1 minute of another `Heartbeat` event from system ?S2 $\neq$?S1 at location ?L1 and there is no `Heartbeat` event from system S3 at location ?L2 $\neq$?L1, then alert the Network Manager for location ?L1.
- Rule 2: If there is a `CompletedPhase1` event from application ?A1 and there is no `CompletedPhase2` event from application ?A1 within 5 seconds of the first event, then alert the Application Manager for application ?A1.

Rule 1 distinguishes network problems from application problems using a pattern consisting of two events from different machines at the same location. Rule 2 checks on the health of a critical business application whose processing steps are executed on different servers. In both rules, there is a **timer value** that constrains the maximum elapsed time between receiving the first and second events in the pattern (although more complex temporal patterns involving more than two events, from a combination of the same and different sources, would often be used). For Rule 1, the timer value might be determined from the experience of system administrators with the timing of related events. For Rule 2, the timer value similarly indicates the anticipated time between processing steps. Note the timer value is specified *a priori* as part of the rule, and is generally a heuristic rather than a precise 'hard' threshold. Note also that the then-part is defined to occur when the second event does not occur within the time-bound (more generally, there could be different actions for satisfaction within a timer value vs. when the timer expires).

Performing temporal event correlation as in Rules R1 and R2 requires that events be consistently timestamped. It is thus desirable that all prospective event sources participate in a global synchronization scheme. But that may not always be feasible: The clock at the event source may be unreliable, e.g., when that source is a customer-managed server. Even when the event source is running the right synchronization software, the system may have a partial failure that affects clock synchronization, or a clock used in a security protocol might be compromised early in an attack. Recall that the purpose of temporal event correlation is to help detect failures and attacks, so we cannot assume they will not occur.

Thus the Management Station should timestamp events with their arrival times at that station (we assume that the station itself is operating properly and has not been compromised). These timestamps will include the delays to propagate each event from its source. Propagation delays may happen to be identical for all events in a pattern, but it is more likely that there will be some *propagation skew*, a term referring to the differences between propagation delays in the events participating in a temporal event pattern. Our experiments reveal propagation skews as large as 50% of the timer values used in sample rules.

It is possible to compensate for propagation skew by automatically adjusting timer values based on measurements taken from appropriate sensors. However, since the propagation skew varies from one instance of a pattern to another, we can only estimate the skew for any particular pattern at any given time. This creates a conundrum. If we over-compensate for propagation skew by using too large a timer value, true problems may go undetected. On the other hand, if we use too small a timer value, there may be many alarms generated for situations that do not merit action, such as a change in workload or re-allocation of resources to the application. False alarms are undesirable since they divert the operations staff from true problems. However, since the timer values are generally imprecise anyway, some number of what turn out to be false and undetected alarms must be expected. One could also use an analogous approach to periodically tweak the timer values in concert with reinforcement learning, aimed towards minimizing a weighted function of false and undetected alarms. We do not discuss the reinforcement learning model further here.

This paper makes the following contributions:

1. Description of the propagation skew problem for temporal event correlation in distributed systems, including measurements of actual propagation skew for a testbed system; and
2. An architecture that includes Calibration Probes, Probe Monitors and a Controller that collaborate to adjust timer values to compensate for propagation skew.

A variety of adaptive control algorithms could be used to adjust the timer values. Due to length limits for this short paper, we point the reader to [7], which presents our particular algorithm in full detail and also assesses our algorithm in terms of the probability of a 'correct' result in the case where the pre-specified timer values should be treated as 'hard' deadlines.

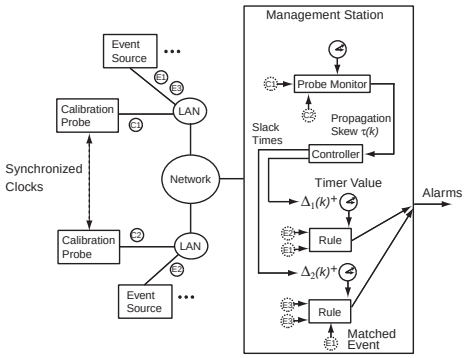

Figure 1.

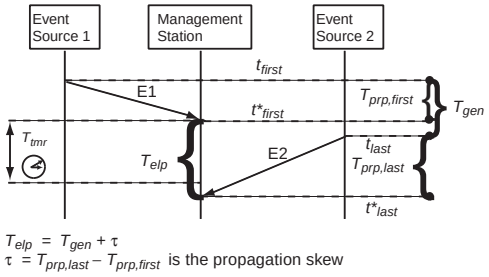

Figure 2.

2. Architecture

Figure 1 depicts how we extend a conventional Management Station architecture to compensate for propagation skew. There are four main components: Reasonably simple (and hopefully thus failure-proof and tamper-proof) instrumentation – Calibration Probes – that create events so that there are known (consistently synchronized) pattern generation times special Calibration Patterns; a Probe Monitor that measures the propagation skew of the events generated by this instrumentation, and compares with the corresponding known arrival times at the Management Station; an autonomic control algorithm (see, e.g., [7] for computing *slack times* that compensate for propagation skews; and a mechanism in the Management Station for extending partially instantiated patterns in the event correlation engine with these slack times, i.e., one way to adjust timer values is to simply sum the timer value and the slack time.

Figure 2 illustrates the procedure for matching a temporal rule consisting of two events, E_1 and E_2. E_1 is generated by Event Source 1 at time t_{first}, and E_2 is generated by Event Source 2 at time t_{last}. Thus, the *pattern generation time* is $T_{gen} = t_{last} - t_{first}$. Administrators generally write rules for temporal correlation based on expected pattern generation times. Consider a timer value T_{tmr} chosen so that an alarm is generated if $T_{gen} > T_{tmr}$. Since the Management Station may not know T_{gen} authoritatively, it uses T_{elp} instead. From Figure 2, $T_{elp} = t^*_{last} - t^*_{first} = T_{gen} + \tau$, where τ is the propagation skew.

The goal of our control algorithm is to automatically select a slack time to add to T_{tmr} such that it considers τ thereby reducing the number of false alarms. The control algorithm operates as follows:

```
Select an initial guess/estimate for the slack time
Use initial slack time estimate until we have collected N observations of τ
Sort last N observations of τ
Pick median
if median > 0 then update slack time
```

3. Experimental Results

Our testbed follows the architecture depicted in Figure 1. The Management Station employs Columbia University's previously developed temporal event correlation engine, called the Event Distiller [6, 2]. The event transport is University of Colorado's Siena publish/subscribe bus [1]. Three components are deployed in our test-bed: A **Calibration Event Generator** produces pairs of *calibration events* E_1 and E_2, separated by a known pattern-generation time e.g. the E_2 is generated 2000 msecs after E_1. These pairs of events are also known as *calibration frames*. Events E_1 and E_2 contain four important fields: FPResolution – the time (in msecs) that should elapse between the generation of E_1 and E_2. FPSeqNum – a sequence number for a calibration frame. Both E_1 and E_2 in a calibration frame will share the same sequence number. FPStartSeq – a flag, set to one in E_1 indicating the beginning of a calibration frame. In E_2 it is set to zero. FPGenGap – only applicable for the end event, E_2, of a calibration frame, it records the actual time (in msecs) elapsed since the generation of the start event, E_1. It is expected that this value would be close to the FPResolution time depending on the current load of the machine where the Calibration Event Generator runs. Figure 3 shows a pair of calibration events, E_1 and E_2, each calibration event is represented

E_1 = { $FPGenGap = "0" FPResolution = "2000" FPSeqNum = "1" FPStartSeq = "1" FPTest = "FPTest"$ }
E_2 = { $FPGenGap = "2041" FPResolution = "2000" FPSeqNum = "1" FPStartSeq = "0" FPTest = "FPTest"$ }

Figure 3. Calibration Frame

as a Siena Notification [1] of size ∼80 bytes.

The **Event Distiller** (Management Station) receives calibration frames and records an arrival time stamp on each event comprising the calibration frame. The difference in the arrival times of E_2 and E_1 is compared to the FPResolution, adjusted based on FPGenGap if necessary, and used to estimate the propagation skew/delays in receiving pairs of calibration events. As an example, a calibration frame with FPResolution = 2000 msecs and FPGenGap = 2005 msecs indicates that the end event, E_2, of a calibration frame was delayed by 5 msecs. On the receiver end, we adjust the difference in arrival times by 5 msecs to compensate for the delay in event generation. Whereas it is possible for events generated $T_{gen} + \epsilon$ to have a difference in arrival times of T_{gen}, due to delays/congestion in the network or packet processing delays on the receiver end, we take the conservative approach of adjusting arrival times at the receiver end primarily to mitigate any scheduling/load issues at the sender that may have delayed event generation.

Finally, a **Siena Event Router** is responsible for managing client subscriptions, receiving published events and routing them (based on their contents) to interested subscribers. The Event Distiller, is an example an interested subscriber of calibration events.

We study four configurations of these three components running on a mix of Windows XP and Linux platforms. A total of four machines in our CS network are used; Kathmandu.clic, Lisbon.clic, Amman.clic and Liberty.psl. Kathmandu, Lisbon and Amman each have a single 3.2 GHz Intel Pentium 4 processor, 1 GB RAM running a 2.6.9-22.0.2.EL Linux kernel. Liberty is a 3 GHz Pentium 4 with 1 GB RAM running Windows XP SP2.

Configuration A is a mixed-platform 3-machine configuration. The Calibration Event Generator runs on the Linux host, Kathmandu.clic, the Siena Event Router runs on the Linux host, Lisbon.clic, while the Event Distiller runs on the Windows XP host, Liberty.psl. **Configuration B** is a homogeneous-platform 3-machine configuration. The Calibration Event Generator, Siena Event Router and Event Distiller run on Linux hosts Kathmandu.clic, Lisbon.clic and Amman.clic respectively. Configurations C and D are 2-machine configurations where the Siena Router and Event Distiller are collocated on the same host. Collocation of the Event Distiller and the Siena Event Router is intended to mimic situations where there is contention for machine resources such as CPU, memory and/or network resources at the management station. **Configuration C** is a mixed-platform 2-machine configuration where the Calibration Event Generator runs on the Linux host, Kathmandu.clic, and the Siena Event Router and Event Distiller both run on the Windows XP host, Liberty.psl. **Configuration D** is a homogeneous-platform 2-machine configuration where the Calibration Event Generation runs on the Linux host, Kathmandu.clic, and the Siena Event Router and Event Distiller both run on the Linux host, Lisbon.clic. For each configuration all machines were located on the same campus LAN and exhibited ping times on the order of <1ms for 32 bytes of data between machines.

In our experiments we observed large variations in the propagation skews measured in configuration C as compared to those measured in configurations A, B and D, Figure 4. It seems counter-intuitive that propagation skews would be larger for a 2-machine configuration than for a 3-machine configuration. Our initial conjecture was that using the difference in arrival times of calibration events at the Event Distiller captures more than network delays. This time difference may also be influenced by contention for shared resources such as the CPU and network I/O stack, which the collocated Siena Event Router and Event Distiller compete for.

We considered it improbable that the relatively large propagation skew values observed in configuration C could be attributed to an extremely inefficient mechanism within Siena for delivering events to a local subscriber, especially when Siena is intended to be a scalable, wide-area event notification service [1]. Based on a side-by-side comparison of the 2-machine configurations, C and D, where the Siena Event Router and Event Distiller are collocated on a Windows XP machine and a Linux machine respectively we conclude that variations in propagation skew are more pronounced under Windows XP than under Linux and our measure of propagation skew is also influenced by resource contention/the current workload on the machine running the Event Distiller.

Figure 4 reports data from configurations A through D. In configurations A, B and D, the propagation skews are tightly clustered around 0, although there are a few large spikes. The second plot in the top row is the cumulative distribution function (CDF), which reinforces the view that values are tightly clustered. Also plotted are the autocorrelations between propagation skews. Note that all autocorrelations lie within the dashed lines, indicating that they are not statistically significant according to the Bartlett Test [3]. In 2-machine configuration C, propagation skews are much more variable and considerably larger.

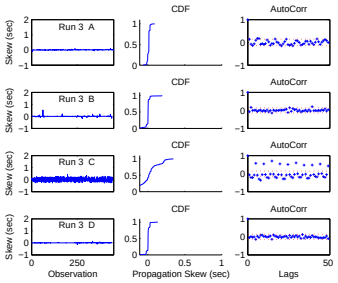

Figure 4.

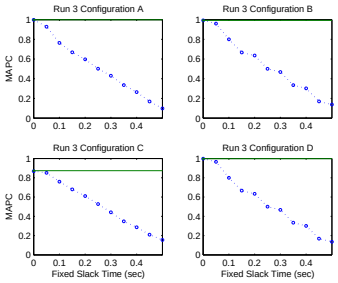

Figure 5.

We also see substantial autocorrelations, possibly due to periodic activities and/or resource contention.

Figure 5 assesses the effectiveness of using fixed slack times. The horizontal axis is the slack time Δ and the vertical axis is Minimum Average Probability of a Correct result ($MAPC$). Large $MAPC$ values are achieved with a fixed slack time near 0 in configurations A, B and D. However, for the 2-machine configuration C, $MAPC$ is maximized at larger fixed slack times. This can be explained by looking at the distribution of propagation delays. The solid line in Figure 5 plots the $MAPC$ values achieved by our adaptive control algorithm ([7]). In all cases, the algorithm selects slack times very close to the value of fixed slack time that maximizes $MAPC$. This is impressive in two respects: First, we did not have to parameterize or train the controller, i.e., slack times are selected in a self-managing manner. Second, we achieve near-optimal results in the 2-machine configurations, even though the data may have significant autocorrelations.

4. Conclusions

Mechanisms towards self-management of distributed systems often require that events be correlated from multiple (sub)systems using temporal patterns. This paper briefly sketches our approach to autonomically adjusting the timer values for instantiated temporal rules in accordance with observed propagation skew. This short paper only presents a small portion of our work on this subject. See [7] for further details concerned specifically with the case where event source clocks cannot be trusted. Forthcoming papers will address the probably more common case of globally synchronized clocks, but where the timer values manually specified in rules remain heuristic and thus themselves uncertain – and in need of dynamic adjustment.

Acknowledgements

Kaiser's Programming Systems Lab is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063.

References

[1] Antonio Carzaniga, David S. Rosenblum and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computing Systems*, 19(3):332–383, 2001.

[2] Gail E. Kaiser, Janak Parekh, Philip Gross and Giuseppe Valetto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. In *Active Middleware Services*, pages 22–31, 2003.

[3] George E.P. Box and Gwilym M. Jenkins. *Time Series Analysis Forecasting and Control*. Prentice Hall, 1976.

[4] Giuseppe Valetto and Gail Kaiser. Using Process Technology to Control and Coordinate Software Adaptation. In *25th International Conference on Software Engineering*, May 2003.

[5] Giuseppe Valetto, Gail Kaiser and Dan Phung. A Uniform Programming Abstraction for Effecting Autonomic Adaptations onto Software Systems. June 2005.

[6] Janak Parekh, Gail Kaiser, Philip Gross and Giuseppe Valetto. Retrofitting Autonomic Capabilities onto Legacy Systems. *Journal of Cluster Computing*, 2006. In press.

[7] Rean Griffith, Joseph L. Hellerstein, Yixin Diao and Gail Kaiser. Dynamic Adaptation of Rules for Temporal Event Correlation in Distributed Systems. Technical Report CUCS-003-05, Columbia University Department of Computer Science, January 2005.

[8] S.A. Yemini, S. Kliger, E. Mozes, Y. Yemini and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, 1996.