E2EProf: Automated End-to-End Performance Management for Enterprise Systems

Sandip Agarwala, Fernando Alegre, Karsten Schwan, Jegannathan Mehalingham[†]

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, fernando, schwan}@cc.gatech.edu

Delta Technology Inc.[†]
Delta Air Lines
Atlanta, GA 30354
Jegannathan.Mehalingham@delta.com

Corresponding Author : Sandip Agarwala (sandip@cc.gatech.edu)

Contact Address : 332213 Georgia Tech Station
Atlanta, GA 30332
USA

Submission Category : Regular Papers describing original research

Submission track : PDS

E2EProf: Automated End-to-End Performance Management for Enterprise Systems

Sandip Agarwala, Fernando Alegre, Karsten Schwan, Jegannathan Mehalingham[†]

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, fernando, schwan}@cc.gatech.edu

Delta Technology Inc.[†]
Delta Air Lines
Atlanta, GA 30354
Jegannathan.Mehalingham@delta.com

Abstract

Distributed systems are becoming increasingly complex, caused by the prevalent use of web services, multi-tier architectures, and grid computing, where dynamic sets of components interact with each other across distributed and heterogeneous computing infrastructures. For these applications to be able to predictably and efficiently deliver services to end users, it is therefore, critical to understand and control their runtime behavior. In a datacenter environment, for instance, understanding the end-to-end dynamic behavior of certain IT subsystems, from the time requests are made to when responses are generated and finally, received, is a key prerequisite for improving application response, to provide required levels of performance, or to meet service level agreements (SLAs).

The E2EProf *toolkit enables the efficient and non-intrusive capture and analysis of end-to-end program behavior for complex enterprise applications. E2EProf permits an enterprise to recognize and analyze performance problems when they occur – online, to take corrective actions as soon as possible and whereever necessary along the paths currently taken by user requests – end-to-end, and to do so without the need to instrument applications – non-intrusively. Online analysis exploits a novel signal analysis algorithm, termed* pathmap*, which dynamically detects the causal paths taken by client requests through application and backend servers and annotates these paths with end-to-end latencies and with the contributions to these latencies from different path components. Thus, with pathmap, it is possible to dynamically identify the bottlenecks present in selected servers or services and to detect the abnormal or unusual performance behaviors indicative of potential problems or overloads. Pathmap and the E2EProf toolkit successfully detect causal request paths and associated performance bottlenecks in the RUBiS ebay-like multi-tier web application and in one of the datacenter of our industry partner, Delta Air Lines.*

Keywords: End-to-End performance diagnosis, online time series analysis

1 Introduction

Modern distributed systems are becoming increasingly complex, in part because of the prevalent use of web services, multi-tier architectures, and grid computing, where dynamic sets of machines interact via dynamically selected application components. A key problem in this domain is to understand the runtime behavior of these highly distributed, networked applications and systems, in order to better manage system assets or application response and/or to reduce undesired effects. In fact, sometimes, the processing of a single request can generate intricate interactions between different components across many machines, making it hard even for experts to understand system behaviors. A concrete example are the '*poison messages*' experienced in the IT infrastructure run by one of our industry partners [19]. Rapid problem detection, diagnosis [10], and resolution in cases like these are critical, since the potential business impact of problematic behaviors (e.g., inordinate request delays, request losses, or service outages), can be substantial. A recent study found, for example, that for a typical enterprise, the average cost of downtime either due to outright outage or due to service degradation is about US$125,000 per hour [1].

Online behavior understanding is also important under normal operating conditions. A case in point is runtime management to meet application-specific Service Level Agreements (SLAs), by classifying requests and then ensuring different service levels for different request classes, or by managing systems to meet certain utility goals [18, 26]. Additional examples are management tasks like job scheduling [15] or resource allocation [23]. For instance, a front-end web request scheduler making online scheduling and dispatching decisions in a multi-tier web service [5] re-

[1] IDC #31513, July 2004

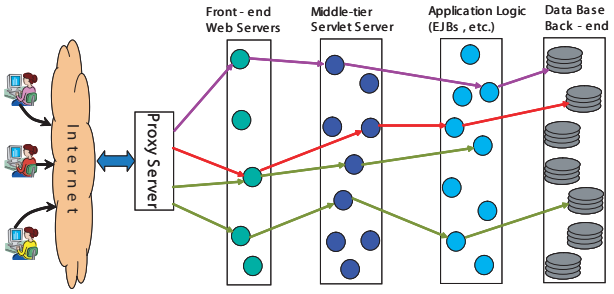

Figure 1: Example ServicePath in a multi-tier web service

quires continuous updates about the execution of the client's requests at the backend servers.

This paper presents the *E2EProf* toolkit for online performance understanding. E2EProf can be used to diagnose the performance problems that arise from complex interactions across multiple subsystems and machines. First, its methods for *end-to-end* performance understanding can capture the entire life-cycles of requests as they are being processed by an enterprise application's many hardware and software components. Second, E2EProf analysis enables *online* problem diagnosis, because of its optimzations and compact trace representations. Third, since E2EProf uses *non-intrusive* kernel-level network tracing for application monitoring, it can operate across the large diversity of applications routinely used in the enterprise domain, without the need to assume the existence of common, clean, and perhaps most importantly, without requiring uptodate monitoring instrumentation. Fourth, E2EProf operates without requiring access to source code, since it is not likely readily available for all of the applications being evaluated and managed by an organization. In fact, even if sources were accessible, the lack of proper documentation often makes it a daunting task to analyze these extensive codes.

The *E2Eprof* toolkit uses an incremental approach to performing *end-to-end* analyses of request behaviors. Specifically, it encapsulates different request interactions across distributed program components with different '*service paths*', where each such path describes a set of dynamic dependencies across distributed components formed because of the services they provide and the requests they service. Figure 1 shows the multiple service paths used by three different types of clients in a multi-tier web service, for example, where paths are differentiated by the kinds of requests being submitted.

Online service path encapsulation is done by correlating the timestamps of the messages exchanged between interacting components. E2EProf's cross-correlation analyses can capture application-relevant performance metrics, such as the end-to-end latencies experienced by requests, and they can determine the contributions of specific application-level services and network communications to such latencies. The choice of requests, components, and service paths to be analyzed can be changed at any time, without the need to recompile, re-link, or re-edit programs.

While the idea of path-based analysis been used by other researchers to discover faults and performance problems in distributed systems [1, 8, 4, 25, 24], E2EProf makes the following unique contributions:

- Its *service path* abstraction can be used to encapsulate the causal paths of different requests (or services), capture end-to-end request delays, and the components of those delays due to each individual software component.
- Its time-series analysis algorithm, termed *pathmap*, discovers the causal request paths from network packet traces non-intrusively, which means pathmap neither requires access to application source code, nor modifications to deployed application services.
- Its ability to understand the performance of complex distributed applications is demonstrated by carrying out detailed online performance analyses for the RUBiS multi-tier auctioning web application.
- The low latency, efficient analyses performed by E2EProf permit it to be used for online management, using a black-box scheduling algorithm to manage Service Level Agreements (SLA) in RUBiS.
- E2EProf has gone beyond in-lab concept demonstrations, by using its pathmap algorithm to evaluate the performance of an enterprise application deployed in one of our industry partner's datacenters, the 'Revenue Pipeline' used in Delta Air Line's Atlanta datacenter.

E2EProf is the outcome of a multi-year effort to develop efficient mechanisms and methods for runtime performance understanding. E2EProf's online analysis permits it to capture and deal with the dynamic behaviors of complex enterprise applications. A specific target class of applications addressed by E2EProf are the *Operational Information Systems*(OIS) [13] used by large organizations for controlling day-to-day operations, an example being the OIS run by one of our industrial partners, Delta Air Lines. In order to function properly, these systems must operate and adapt to changes within well-defined constraints derived from their SLAs and dependent on the business values or utilities associated with their various services. If a SLA is violated, system administrators usually analyze large complex logs in order to isolate faulty components. E2EProf can be used to automate performance diagnosis, thereby reducing such maintenance costs.

In the remainder of this paper, we describe the service path abstraction and various components of E2EProf toolkit. The next section surveys the related work. Section 3 describes the pathmap algorithm and analyzes it in detail. Experimental evaluation is presented in Section 4 together with some realistic test cases of performance diagnosis and management. Conclusions appear in Section 5

2 Related Work

The large number of tools available for distributed system performance diagnosis may be categorized based on three broad features: online/offline, level of intrusiveness, and quality of analysis.

Single web server system performance has been studied extensively. EtE [12] and Certes [22] measure client-perceived response time at the server side. The former does offline analysis of the packets sent and received at the server side, while the latter does online analysis by observing the states of TCP connections.

Tracing tools for single systems like the Linux Trace Toolkit [27] and Dtrace [6] provide mechanisms for logging events by inserting instrumentation code. Compiler-level instrumentation is commonly used to understand program behaviors (e.g. gprof.) However, source code may not always be available, and the sizes and complexities of sources are disincentives for software engineers engaged in post-development instrumentation or evaluation. Even binary instrumentation requires some level of understanding of application details.

Path-level analysis of distributed systems tracks the causal relationship between different components and has recently been an area of active research. ETE [14] uses application-specific instrumentation to measure the latencies between component interactions and relates them to end-to-end response times to detect performance problems. Pinpoint [9] detects system components where requests fail, by tagging (and propagating) a globally unique request ID with each request. Magpie [4], on the other hand, requires no global ID, and it can capture not only the causal paths, but also monitor the resource consumption of each request. Industry standards like ARM [3] (Application Response Measurement) used by HP's Openview, IBM's Tivoli, and BEA's Weblogic require middleware-level instrumentation to measure end-to-end application performance. In contrast, E2Eprof does not require any modification to applications and therefore, can also be used with legacy components. However, unlike Magpie, it does not measure general resource usage.

The work by Aguilera *et al.* [1] is most closely related to E2Eprof. They propose two algorithms to determine causally dependent paths and the associated delays from the message-level traces in a distributed system. While their *nesting* algorithm assumes 'RPC-style' (call-returns) communication, their *convolution* algorithm is more general and does not assume a particular messaging protocol. Our pathmap algorithm is similar to the *convolution* algorithm, in that both uses time series analysis and can handle non-RPC-style messages. While the convolution algorithm is primarily intended for offline analysis, pathmap uses compact trace representations and a series of optimizations, which jointly, make it suitable for online performance diagnosis.

3 Service Paths

3.1 Basic Abstractions, Methods, and Assumptions

In modern enterprise systems, different client requests may belong to one or more *service class(es)*, which are defined on the basis of simple request types, clients IDs, or more generally, SLAs. These requests may take different paths through the enterprise software, invoking different and multiple software components before responses are generated. We term the ensemble of paths taken by client requests in different service classes as '*Service Paths*'.

Service paths form the basis of E2EProf's online end-to-end performance analyses, because they characterize the end-to-end properties sought by the enterprise and capture the complex dependencies that exist across the different software components involved in service provision. For each path, E2EProf's analyses can describe not only the path's end-to-end latency but also the latencies incurred across different path edges. Therefore, service path analysis can pinpoint the bottleneck components in a request path, and it can be used for provisioning, capacity planning, enforcing SLAs, performance prediction, etc.

The *pathmap* algorithm uses time-series analysis to discover the service paths of different service classes, making the following assumptions:

- Each client's requests belong to a unique *service class*, which is known to the front end (i.e., the first nodes in the distributed system that receives the request). Pathmap assumes that requests belonging to the same service class have similar resource requirements, and that they tend to take the same paths through the distributed system.
- A request path can either be unidirectional (as in streaming media applications) or bidirectional as in the request-response conduits used in multi-tier web services. In the latter case, responses traverse the same set of nodes as the corresponding requests, but in reverse order.
- Pathmap assumes that the distributed application and system are operating in steady state during the analysis 'time window', where deviations are due to internal anomalies or external drastic changes in system usage. Such anomalies occur when a node malfunctions, when a network link goes down, or when a buggy application overloads the system, for example. A sample abnormal external change may be a malicious attack or a sudden increase in user interaction (e.g., the *Slashdot effect*.)
- At small time scales, there may be large variability in the processing of individual requests, but in steady state, the system is assumed to be adequately provisioned so that the queuing and processing delays at each of its

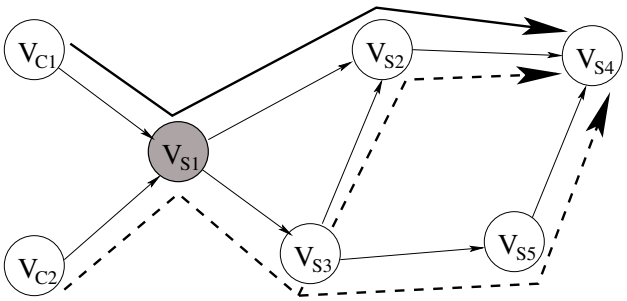

Figure 2: Example Service Graph: V_{c1} and V_{c2} are the client nodes and V_{sn} are service nodes.

nodes don't significantly change the distribution of the intermediate responses (generated as a result of partial processing of the requests at the intermediate nodes in the path), as compared to the arrival distribution at the front-end. Pathmap can, however, accommodate changes in rate across nodes (e.g., an EJB server issuing multiple data base queries for a single client requests).

3.2 System Representation

Formally, a distributed application or system may be described as a directed graph G(V,E), where the vertices in the graph represent application components and the edges represent their logical communication links. The *service graphs* considered in this paper are comprised of nodes that may be processes, threads, or machines, communicating via edges that map to network links [2].

Each service graph has two type of nodes: client nodes(V_C) and service nodes (V_S). Requests originate in client nodes, where we assume that the requests issued by each particular client node belong to the same service class. A physical client issues multiple classes of requests will be modelled as multiple client nodes, one per request class. Service nodes house software components that operate on requests. They are labelled by their IP addresses or by a combination of their IP addresses and process IDs, depending on whether there is one or more service node per physical machine node (e.g., an application server and database server being located on the same physical machine).

Edges denote logical communication link between service nodes. These logical connections are characterized by source and destination address pairs. They may be transient, which will usually be the case in front-end servers, or persistent, which is typical for middle and back-end servers. Furthermore, a single connection may consist of aggregated traffic from separate clients, and it may therefore, exhibit multiple traffic patterns. Figure 2 depicts a sample service graph. For this graph, the goal of the *pathmap* algorithm is to compute the paths of requests from each client node through the service graph, along with the delays incurred in traversing the edges and nodes in those paths.

[2]Although we consider only network communication links, the E2EProf approach can also be extended to IPC mechanisms like *pipes* and *message queues*.

3.3 Pathmap Algorithm

The pathmap algorithm relies on the E2EProf tracing subsystem, which uses standard operating system facilities to collect message traces from each service node. Traces are not collected from client nodes, since those are usually beyond the reach of enterprises. The key idea of the pathmap algorithm is to convert these message traces to per-edge time series signals and then compute the cross-correlations of these signals. Specifically, if a signal f contains a copy of the signal g, then their cross-correlation signal ($f \star g$) has a distinguishable spike at position d, where d is equal to the time that the copy of g in f has shifted from g. This kind of correlation analysis is commonly used in digital signal processing to compute the level of similarity between two signals.

First introduced by Aguilera *et al.* [1] in a similar context, pathmap uses cross-correlation analysis to discover the most probable request paths in a distributed system. Consider the request path ($V_{C1} \rightarrow V_{S1} \rightarrow V_{S2} \rightarrow V_{S4}$) shown in Figure 2. Let $T^{x}_{x \rightarrow y}$ be the time series signal of the messages from x to y collected at the node x, and $T^{y}_{x \rightarrow y}$ be the time series signal for the same set of messages collected at node y. The cross-correlation plot of $T^{s1}_{c1 \rightarrow s1}$ and $T^{s1}_{s1 \rightarrow s2}$ (denoted by $corr(T^{s1}_{c1 \rightarrow s1}, T^{s1}_{s1 \rightarrow s2})$) has a spike at position d, where d is the time that V_{s1} takes to process V_{c1}'s request. This implies that there is a causal relationship between messages on edge $V_{c1} \rightarrow V_{s1}$ and messages on edge $V_{s1} \rightarrow V_{s2}$. Similarly, the cross correlation plot $corr(T^{s1}_{c1 \rightarrow s1}, T^{s2}_{s2 \rightarrow s4})$ also has a spike, and its position is the sum of the communication latencies at the two edges ($V_{C1} \rightarrow V_{S1}$ and $V_{S1} \rightarrow V_{S2}$) and of the computation latencies at the two vertices (V_{s1} and V_{s2}). The presence of the spike also indicates a causal relationship between messages on edge $V_{c1} \rightarrow V_{s1}$ and messages on edge $V_{s2} \rightarrow V_{s4}$. The cross-correlation plot $corr(T^{s1}_{c1 \rightarrow s1}, T^{s1}_{s1 \rightarrow s3})$, however, has no distinguishable spike as no requests from V_{c1} pass through V_{s3}.

The above example illustrates how correlation can be used to establish causality between different edges. Given this background, Algorithm 1, outlines the actual pathmap algorithm. It takes as input the time-series data streams computed from the message timestamps collected at different service nodes. The most recent *sliding window* of size W is maintained for each of these streams. After every time interval ΔW, the '*ServiceRoot*' function is invoked to update the service graphs for all *clients* belonging to different service classes. For the analysis to be statistically significant, the size of W is chosen such that it contains large number of requests. The algorithm starts tracking the path at the front-end service nodes, which become the roots of

Algorithm 1 Pathmap

Let W = Length of sliding window
Let ΔW = Service Graph refresh interval
Input: Online time series data streams from service nodes
function ServiceRoot()
for all Service node S_i that are at the front-end **do**
  for all Client nodes V_c connected to S_i **do**
    Service Graph G_c = {}
    Add S_i in Graph G_c
    Add an edge $E_c(V_c \rightarrow S_i)$
    ComputePath(G_c, $T^{S_i}_{V_c \rightarrow S_i}$, S_i)
  end for
end for

function ComputePath(G_c, T_c, S_i)
Mark S_i as visited
Let S_d = List of destination nodes S_i is connected to
for all d_s in S_d **do**
  $corr$ = ComputeCrossCorrelation(T_c, $T^{d_s}_{S_i \rightarrow d_s}$)
  P = List of spike's position in $corr$
  if P is not empty **then**
    if vertex d_s not in G_c **then**
      Add vertex d_s in G_c
    end if
    Add an edge $E_s(S_i \rightarrow d_s)$ and label it with P
    if d_s not visited **then**
      ComputePath(G_c, T_c, d_s)
    end if
  end if
end for

service graphs. In addition, it adds an edge between the client node and the root vertex and then calls *ComputePath* to calculate rest of the graph.

ComputePath's parameters are a partial service graph G_c, a time-series signal (T_c) of the incoming requests of the service class (say C) at the front-end for which the service graph G_c is being determined, and the service node (S_i) to be processed next. Its job is to find the next set of service nodes used by the request class represented by the time-series T_c. This is done by the process of correlation described above. Basically, T_c is cross-correlated with the time-series signal from the nodes(d_s) adjacent to S_i. If the correlation is high (as indicated by the presence of the spikes), then there exists a path from S_i to d_s taken by the requests belonging to service class C. This is recorded by adding vertex d_s into the graph G_c (if such a vertex does not yet exist) and by adding an edge from S_i to d_s. The edge is labelled with the *delay*(s) as denoted by the spikes' position in the cross-correlation test. This delay is the sum of the time taken by the request to arrive at node S_i, the processing delay at node S_i, and the communication delay in the path from S_i to d_s. The computing delay at node S_i is the difference of the delays corresponding to its incoming and outgoing edges. The existence of more than one spike indicates that the request may have taken different paths to S_i (e.g., $S_1 \rightarrow S_2 \rightarrow S_i \rightarrow S_4$ and $S_1 \rightarrow S_3 \rightarrow S_i \rightarrow S_4$). Once the path to d_s is established, the algorithm proceeds further by performing a recursive depth-first search and exploring other edges in the service graph.

Spikes in the cross-correlation series are detected by finding *points* that are local maximas and exceed a threshold ($mean + 3 \times Std.Dev.$). In traces with some noise, there may exist spikes that are very close to each other. To address this issue, we define a resolution threshold window that chooses only the tallest spike in a particular window.

3.4 Computing Cross-Correlation

The most expensive step in the pathmap algorithm is computing the cross-correlation. The basic formulation of the discrete cross-correlation shown in Eq. 1 can be computed in $O(n^2)$ time.

$$Corr_d(x,y) = \frac{\sum_{i=0}^{n-1}(x_i - \overline{x})(y_{(i+d)} - \overline{y})}{\sqrt{\sum_{i=0}^{n-1}(x_i - \overline{x})^2}\sqrt{\sum_{i=0}^{n-1}(y_{(i+d)} - \overline{y})^2}} \quad (1)$$

$where,$ d = 0,1,...,(n-2),(n-1)

The *cross-correlation theorem* (Eq. 2) provides an efficient alternative to compute cross-correlation. The *Fourier transform* can be computed using FFT(*Fast Fourier Transform*), which reduces the time to calulate cross-correlation from $O(n^2)$ to $O(n \log n)$.

$$x \star y = Corr(x,y) = \mathcal{F}^{-1}\left[\mathcal{F}[x]\mathcal{F}[y]^*\right] \quad (2)$$

$where,$ $\mathcal{F}$ denotes Fourier transform, and
z^* denotes the complex conjugate of z.

Although FFT-based computation is more efficient and is the *de facto* standard in computing the cross-correlation of two arbitary signals, it has certain limitations. First, it is not incremental. However, when processing online streams of timestamped data, it is desirable for analysis to be done incrementally, rather than recomputing the correlation from scratch. In the Algorithm 1, we would ideally like the *ComputeCrossCorrelation* function to update correlation information based on the new time series data (of length ΔW) that has been appended most recently, rather than recalculating it for the complete *sliding window*. Second, Eqn. 2 computes cross-correlation for the full range of delay corresponding to the input time series. That is, if the length of the *sliding window* is 10 minutes, the length of the cross-correlation series is also 10 minutes. However, there are scenarios when correlation needs to be evaluated for short delays only.

For our analysis, we choose the direct cross-correlation method (Eqn. 1), because it can be adapted easily for incremental computation of correlation metrics, in addition to

other optimizations. The first optimization is based on the fact that most transactions in a distributed system are just a small fraction of the *sliding window*. Since our goal is to find the service transaction delays and not the full range of cross-correlation series, by assuming an upper bound (say T_u) on the transaction delay, the time complexity of computing cross-correlation directly (i.e., without FFT) is drastically reduced from $O\left([\frac{W}{\tau}]^2\right)$ to $O\left(\frac{T_u}{\tau} \cdot \frac{W}{\tau}\right)$. τ is the time quanta or the smallest delay of interest. In comparison, the time complexity of FFT-based cross-correlation (Eqn. 2) is $O\left(\frac{W}{\tau} \log \frac{W}{\tau}\right)$, which is less than the $O\left(\frac{T_u}{\tau} \cdot \frac{W}{\tau}\right)$ even for small values of T_u. Fortunately, direct cross-correlation is incremental (as discussed earlier), and therefore, it can be computed over only the newly appended trace of size ΔW. This reduces the time complexity of direct cross-correlation further, to $O\left(\frac{T_u}{\tau} \cdot \frac{\Delta W}{\tau}\right)$.

A third important optimization is based on the fact that the network packet traffic in the Internet and in most enterprise systems is inherently bursty. This burstiness can be due to system or user behavior [11, 2], or it can be due to the lower level network protocol (e.g., TCP) behavior and network queueing [17]. In addition, a single transaction may be composed of multiple packets sent back-to-back. Bursty behavior results in dense network packet traffic intermixed with 'long' quiet zones. Our optimization takes advantage of this fact by simply omitting to compute correlation in the 'quiet' region, without compromising the accuracy of the result. This is done by computing the *time series* in such a way that the entries with value 0 (i.e., zero packets seen at the time corresponding to that entry) are discarded. As a result, the length of the *time series* trace is reduced by a large margin (more than 10 times for some of our enterprise traces). This not only decreases the computation time of the direct cross-correlation, but also increases the efficiency (both in time and space) of collecting the trace at each service node, as we shall see in the next section. In summary, assuming that the average factor of *time series* reduction is 'k', the time complexity of direct cross-correlation drops to $O\left(\frac{T_u}{\tau} \cdot \frac{(\Delta W)/k}{\tau}\right)$.

3.5 Computing Time Series

The message traces collected at service nodes are converted to time-series data using a *density function* $d(i)$, which represents the '*density*' of the packets at time instant $i \cdot \tau$ (or *ith* time quanta). The density function estimation is based on two parameters: time quanta (τ) and the size of *rectangular sampling window* (ω), an integral multiple of τ.

$d^x_{x \to y}(i)$ = square root of number of messages at service node x transmitted to y in time interval $\left[i \cdot \tau - \frac{\omega}{2},\ i \cdot \tau + \frac{\omega}{2}\right]$

Figure 3 shows a pictorial representation of time series computation. The message arrivals are shown as small rectangular boxes. Both W (size of sampling window) and ΔW (refresh interval) are also integer multiples of τ. Note the entry $d_i = (t_i, n_i)$ in the time-series computation in Figure 3. No packet was received during the ith sampling window, and therefore, as discussed in the previous section, d_i is not recorded in the time-series. The size of time quanta τ determines the resolution of the analysis. For a given sliding window size (W), a small τ results in longer time-series ($\frac{W}{\tau}$) and a proportional increase in the cost of servicepath analysis. Its value, therefore, should not be arbitrary small, but equal to the shortest service delay of interest. The purpose of the rectangular sampling window is to reduce the effect of variance in delay and suppress infrequent paths that occur due to the noise in the trace. A very small ω may produce many spikes during cross-correlation analysis resulting in false delays/paths. On the other hand, a large value of ω may over-generalize the result (collapsing two spike into one, for example). For the systems we have analyzed, $\omega = 50 \cdot \tau$ gave the best set of results.

The process of time-series computation is further optimized using run-length encoding (RLE). Upon close examination of the time-series of actual enterprise traces, we found that there are many repeatable sequences, which provide substantial room for compression. RLE is particularly appropriate for this purpose, because it can be computed online, with negligible compression and decompression overheads. This not only reduces the network transmission overhead (when the time-series data is streamed to the remote node for analysis), but it also decreases the cost of cross-correlation analysis because the correlation of overlapping sequences in the series (Eqn 1) can be computed in a single step. The resultant time-series becomes a 3-tuple series (t, c, n) (one tuple for each *run*), where t is the timestamp of the first density function entry in the *run*, c is the length of the *run* and n is the value of density function.

3.6 Trace Collection

One of the requirements of service path analysis is that no application components should be modified or restarted. Also, the system should experience as little perturbation as possible. Our analysis requires timestamps and (source, destination) identification of the inter-component messages. These messages may be collected at various levels: at the application level (e.g., apache web server's access logs), at the middleware level (e.g., J2EE-level tracing [8]) or at the system and network level. The problem with tracing transactions at the application- or middleware-level is that there is not a single and widely deployed standard. *Application Response Measurement* (ARM) [3] is one such standard for monitoring transactions end-to-end in enterprise systems. The ARM standard was proposed in 1996 by a consortium of companies, but it still has limited acceptance.

Passive network tracing provides a convenient way of listening to the interactions between different *service nodes*,

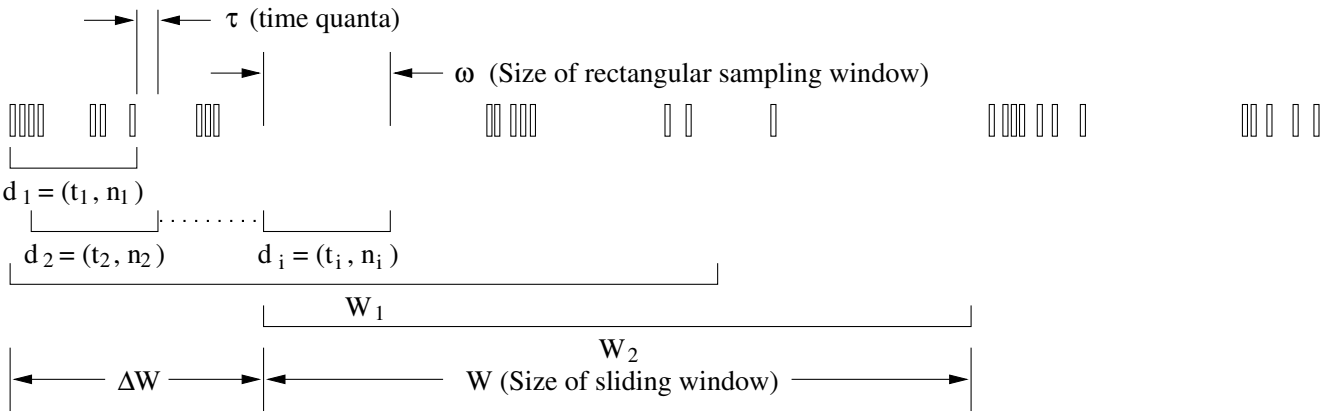

Figure 3: Time series computation

without the need to modify any system components. Network packet traces may be collected from ethernet switch with *port mirroring* support or directly from service nodes by running *tcpdump*. The traces obtained can be streamed to some central location for analysis. Although, this looks like a simple and attractive approach, it limits the scalability of our overall servicepath analysis. This is because the analysis node has to first compute the time series and then the service paths. Offloading the time-series computation to the service nodes decreases the work on central node. Also, the time-series can be calculated directly from the network activity at the service nodes instead of first logging the raw packet traces (using tcpdump) and then converting it to time-series signals. Towards this end, we implemented a linux kernel module called *tracer*, which uses the '*netfilter*' hooks to listen to the packets in the network stack and streams *REL*-encoded time series data.

3.7 Complexity Analysis

The overall time complexity of our pathmap algorithm is $O\left(E \cdot [\frac{W}{\tau}]^2\right)$, where E is the total number of edges in the service graph, W is the sliding window size and τ is the *time quanta*. After applying all optimizations discussed in previous sub-sections, the time complexity is reduced to:

$$O\left(E \cdot \frac{T_u}{\tau} \cdot \frac{(\Delta W)/(k \cdot r)}{\tau}\right),$$

where T_u is the maximum possible transaction delay and ΔW is the service graph update interval. k is the optimization factor achieved by skipping quiet intervals in the packet traces and r is RLE compression factor. Assuming $W = m \cdot \Delta W$, the above can be rewritten as:

$$c_1 \cdot \left[\frac{1}{k \cdot r \cdot m} \cdot \frac{T_u}{\tau} \cdot E \cdot \frac{W}{\tau}\right],$$

where c_1 is a constant. On the other hand, the complexity of FFT-based cross-correlation (Eqn. 2) is $c_2 \cdot \left[E \cdot \frac{W}{\tau} \log \frac{W}{\tau}\right]$, where c_2 is a constant and is much larger than c_1. Comparing the two equations, it is easy to see that our optimized direct cross-correlation approach is much more time efficient than FFT-based computation.

The pathmap algorithm receives a total $2 \cdot E$ number of time-series signal streams from the service nodes, two from the two nodes connected by an edge. It stores the cross-correlation vectors (of size $\frac{T_u}{\tau}$) and a history of time-series (of the size of sliding window $\frac{W}{\tau}$) for each of these edges. The total space complexity, therefore, turns out to be $O\left(2 \cdot E \cdot (c' \cdot \frac{T_u}{\tau} + c'' \cdot \frac{W/(k \cdot r)}{\tau})\right)$.

The pathmap algorithm can easily be made more scalable by parallely computing the service graph of each client nodes (i.e., parrallelizing the inner loop of *ServiceRoot*). The results reported in this paper use a single central analyser.

3.8 Other Considerations

We have implicitly assumed that the clocks of all service nodes are time-synchronized. Pathmap can tolerate small clock skews (i.e., equal to few times of the time quanta τ) when determining service paths, but will exhibit some inaccuracy (equal to the amount of skew) whem computing service delays. Fortunately, most of today's machines are synchronized using NTP, which has an RMS errors of less than 0.1 ms on LANs and of less than 5 ms on Internet (except during rare disruptions) [20]. If the skew is large, cross-correlation results will not be accurate. We can, however, estimate time skew between two service nodes (say x and y) by cross-correlating the time series $T^x_{x \to y}$ and $T^y_{x \to y}$ streamed from x and y respectively. The resultant cross-correlation series will have a spike at position 'd', where d is equal to the sum of the time by which x lags behind y and the network delay. The latter can be computed easily by one of the various passive network measurement techniques [16].

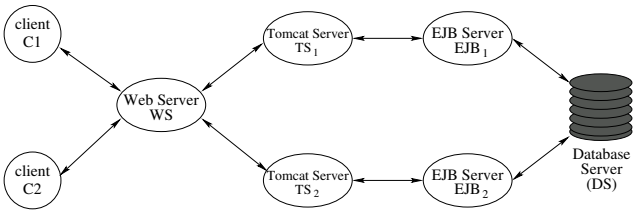

Figure 4: Multi-tier RUBiS application setup

4 Evaluation

The E2Eprof toolkit has been implemented in C and tested extensively on Linux-based platform for both artificial traces and actual enterprise applications. For lack of space, we will present results from just two enterprise-scale multi-tier applications. The first is an open source multi-tier online auction benchmark, called *RUBiS*, from Rice University [7], and the second is the *Revenue Pipeline* application used by Delta Air Lines. We evaluate the overhead and accuracy of E2Eprof and demonstrate how it can be used for online performance debugging in these applications.

4.1 Multi-tier Application: RUBiS

RUBiS implements the core functionalities of an auction site like selling, browsing, and bidding. RUBiS is available in three different flavors: PHP, Java HTTP Servlets and Enterprise Java Beans (EJB). We use the EJB's stateless session beans implementation with the configuration shown in Figure 4. The *Tracer* kernel module runs on all six server nodes and streams time series data to a remote analyzer (not shown in the figure). The two client nodes run *httperf* [21] to generate requests belonging to two service classes (i.e., *bidding* and *comment*). The httperf workload generator in the client nodes emulates 30 clients by initiating 30 client sessions each. Web service requests generated by these client sessions have a *Poisson* arrival distribution. We experiment with two different path configurations:

- *Affinity-based*: the web server forwards all bidding requests to Tomcat server 1 (TS_1) and all comment requests to Tomcat server 2 (TS_2). The path of the bid request becomes $C_1 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$. Similarly, the path of the comment request is $C_2 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$.
- *Round-Robin*: the web server dispatches requests to the two tomcat servers in a round-robin fashion. Here, the bid requests take two different paths: $C_1 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$ and $C_1 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$. Similarly, comment requests has two paths: $C_2 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$ and $C_2 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$.

For RUBiS experiments, the pathmap algorithm parameters are configured as follows: Sliding Window (W) = 3 minutes, refresh interval (ΔW) = 1 minute, time quanta (τ) = 1ms and sampling window size (ω) = 50ms. The upper bound on transaction delay (T_u) is set to 1 minute. These values are chosen based on the guidelines discussed in section 3.4.

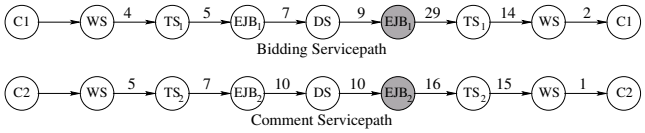

Figure 5: Service Graph for affinity-based server selection. (All delays in milliseconds)

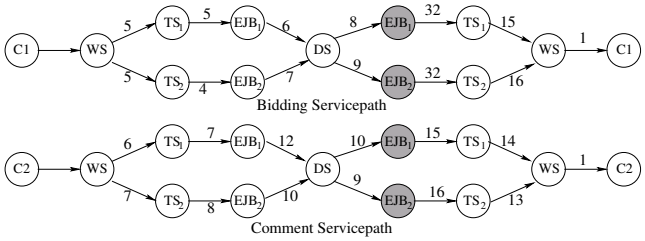

Figure 6: Service Graph for round-robin server selection (All delays in milliseconds)

4.1.1 Service Path Detection

Figure 5 shows the service graph for affinity-based server selection. Here, E2Eprof correctly discover the paths of the two type of client requests. The vertices indicate the different servers, which are hosted on different physical machines. The label on the edge indicates the sum of the computation delay at the source node and of the communication delay from source to destination node. The paths of two types of requests are structurally similar, except for the difference in the service nodes they traverse and the delays incurred. The major sources of delay are automatically detected by E2Eprof and marked in grey (i.e., the EJB servers in the figure). Note the duplicate vertex label in the service path. This is due to the return path taken by the response. For clarity, we avoid using cycles in the figure.

Figure 6 shows the service graph for round-robin server selection approach. The two paths taken by each type of requests are shown, and the major source of delay are marked in grey.

In order to verify the correctness of our results, we add code to RUBiS' servlets and EJB components to keep track of transaction latency at different servers, by piggybagging performance delay information in requests and responses. The resulting performance data coupled with the access logs from the web server and the response time observed at the clients are compared against the service path results generated by E2Eprof. The difference of the processing delays computed at each server is within 10%. The latency observed at the client is about 16% more than that obtained from E2Eprof.

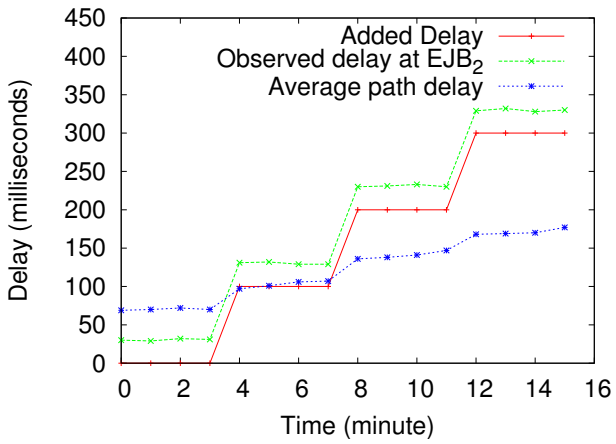

Figure 7: Performance change detection

4.1.2 Change Detection

One of the goals of online service path analysis is to detect changes in path performance. We are interested not only in cumulative end-to-end delays, but also in fluctuations in *per-edge* performance. This is useful for isolating bottlenecks, re-routing request traffic, debug anomalies, etc. In order to demonstrate this capability of E2Eprof, we vary the performance of one of the EJB servers (EJB_2) in the round-robin server selection setup, by artificially introducing some amount of delay in the bid request processing and increasing it after every 3 minutes. The length of the sliding window (W) is set to 1 minute. The other parameters of the pathmap algorithm are the same as in the previous experiments. Figure 7 shows the actual delay introduced and the bid request processing delay at EJB_2 captured by E2Eprof. The algorithm correctly tracks the change in performance. The difference between the observed and added delay is due to the fact that the former includes the actual time spent by EJB_2 in processing the requests in addition to the artificial delay introduced in the experiment. The delay patterns of other edges remain unchanged. The figure also shows the average processing delay observed at the front-end web server. Since more than half of the requests take the low latency path (via EJB_1), the average delay does not change by the same amount. In cases like these, E2Eprof can help diagnose bottlenecks faster, because it can separately track the performance of each service node.

4.2 Automated Path Selection

The front-end web server, among other things, has to perform request scheduling and dispatching, the purpose of which is to ensure load balancing and provide quality of service. Often, different workloads are associated with certain performance goals (e.g., minimum throughput or best response time) and may have certain SLAs associated with them. For example, a *bidding* request in an online auction site like RUBiS has real-time deadlines, while a *comment* posted by a user has a less stringent deadline. Under normal circumstances, the round-robin server selection scheme works 'fairly' well. However, when the application servers experience performance problems, the simple round-robin scheme may not be able to meet SLA requirements.

Table 1: Average latency with different path selection method

	Bidding	Comment
Round-Robin (No perturbation)	72 ms	64 ms
Round-Robin (with perturbation)	121 ms	109 ms
E2EProf (with perturbation)	97 ms	139 ms

In order to improve upon round robin scheduling, we design a setup similar to the previous experiments, with two different classes of workload (bidding and comment), but introducing artificial delay experienced by the two EJB servers, which changes once per minute. These delays are randomly chosen, ranging from 0 to 100 milliseconds. The aim is to reduce the latency of the bidding requests. Furthermore, the server selection algorithm in the web server is modified to route bidding requests to the lower latency path and comment requests to the other based on path latency information obtained from E2EProf. Table 1 shows the average latency of bidding and comment requests measured during a 10 minutes period. After the perturbation is introduced, the average latencies of both types of requests increase with round-robin path selection. In comparison, the E2EProf-based scheduling method decreases the processing delay of bidding requests by directing them to the lower latency paths and penalizing comment requests.

The above is a straightforward example of automated performance management with E2EProf's path-based analysis. Clearly, the E2EProf-based path selection method performs better because it uses more information than the round-robin method, the latter being a black-box approach. We show these results simply to demonstrate E2EProf's utility for online and automated system management, in addition to its already proven use by system administrators to diagnose performance problems in complex enterprise applications. A concrete example of the latter is described in the next section.

4.3 Delta's Revenue Pipeline Application

The "*Revenue Pipeline System*" is a subsystem of Delta's OIS (Operational Information System) that keeps track of operational revenue from worldwide flight operations. It is composed of multiple black-box components (including legacy components) purchased from many different software vendors. About 40K events per hour arrive in one of 25 queues in the front-end control system and are then forwarded to the back-end servers, as shown in Figure 8. Each event/request has strict SLAs. If an SLA is violated, system administrators have to analyze complex logs in order to isolate the faulty components. This process is quite time-consuming, in part because of complex dependencies across multiple black-box components.

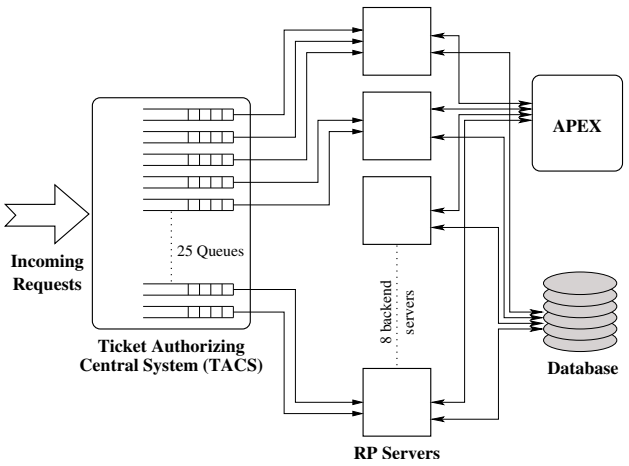

Figure 8: Delta Airlines' Revenue Pipeline Application

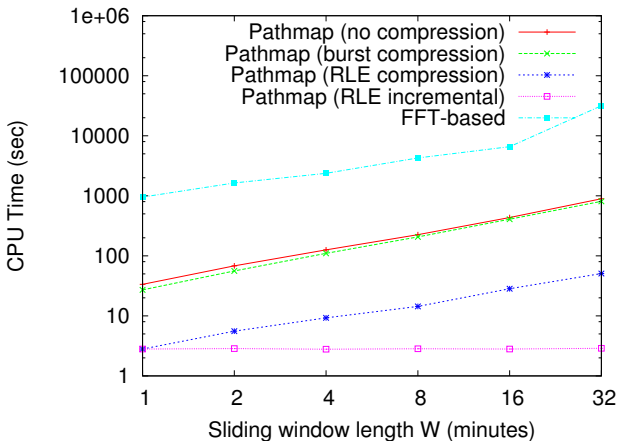

Figure 9: Execution time of service path analysis

E2EProf is used to analyse a week long trace collected from this subsystem. This trace consists of *access logs* from different servers and contains timestamps, server IDs, and request IDs for every application-level transactional event processed by the system (as opposed to the network-level packet events analysed in earlier experiments).

Several limitations of the existing pathmap algorithm are exposed by this use case. First, this subsystem's queuing delays can be large (much larger than the actual processing time). This changes the arrival pattern of the requests at different stages of request processing. Second, there can be wide variations in request traffic. For example, a batch process consisting of all of Delta Air Lines' paper tickets processed all over the world in the last 24 hours is submitted at 4 AM EST, due to which the queue length goes as high as 4000. These facts break the 'steady state' assumption made by the algorithm. Thus, although the pathmap algorithm is able to compute the service path correctly, the computed delays are far from accurate. In response, we have to carefully set the sliding window length (1 hour), the time quanta (1 second) and the sample window (50 seconds), thereby eliminating the error due to traffic variation. The analysis error due to the large queue length could not be eliminated.

Despite inaccurate delay computation, the service paths computed above are still useful in detecting causal dependencies across different components. For instance, E2EProf was able to successfully diagnose a slow database server connection that resulted in large response time for a moderate workload.

4.4 Micro-Benchmarks

Micro-benchmarks are used to examine the costs of E2Eprof analysis for RUBiS traces. The results of overhead analysis for the Delta Air Lines traces are similar to those shown here, and we omit them for lack of space. We evaluate the cost of E2Eprof analysis with the different optimizations discussed in earlier sections and compare it with the FFT-based analysis. Figure 9 shows the time required to compute the service graphs shown in Figure 6 for different sliding window sizes (W). Other parameters of the pathmap algorithm are the same as in earlier experiments with RUBiS: τ = 1ms, ω = 50ms, T_u= 1 minute. The plot labelled '*no compression*' just assumes an upper bound on transactional delay with no other optimizations. The '*burst compression*' plot only considers non-zero time series entries. '*RLE compression*' uses run-length encoded time series data. '*FFT-based*' plot uses *FFTW* package (www.fftw.org) to compute the cross -correlations. FFTW is one of the fastest implementation of FFT.

From the results, it is clear that the RLE-based pathmap algorithm outperforms other methods by orders of magnitude. The cost of pathmap analysis increases linearly with W. For a sliding window of length 32 minutes, the RLE-based algorithm takes just 50 seconds. In reality, a 32 minute window may be too large for enterprise applications, as they need to react to changes within a few seconds to a few minutes. FFT-based analysis does not have linear cost and thus, takes an order of magnitude more time than pathmap to compute the same service graphs. Note that the cost of '*incremental*' pathmap analysis is almost constant for refresh interval (ΔW) set to 1 minute. This makes pathmap suitable for online analysis. The *burst compression* technique does not show much improvement over normal pathmap for RUBiS traces, but it decreases the length of time series (and therefore space overhead) significantly, as shown next.

Trace size: Figure 10 shows the compression achieved by different pathmap's optimizations for the time-series data of the connection between one of the tomcat servers and the web server. The plot labelled '*total packets*' shows the number of packets captured from which these time-series was computed. The time series length increases linearly with window size W, and the plot labelled '*no compression*' is the upper bound ($\frac{W}{\tau}$) on the time series length for a given W and τ. Once again, RLE compression achieves the best results and decreases the length of time

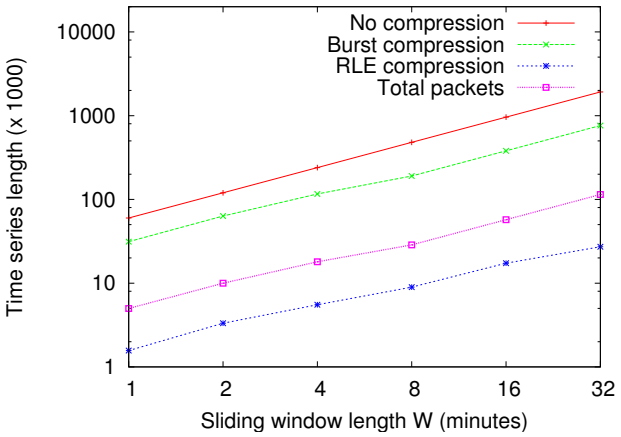

Figure 10: Time series compression

series by an order of magnitude as compared to other optimizations. It is also much smaller than the raw timestamped data (indicated by the total number of packets). Although there are better techniques to compress packet traces, the advantage of using RLE compression is that it also reduces the time complexity of the pathmap algorithm.

5 Conclusions and Future Work

The complexity of distributed systems have been increasing rapidly. To address this complexity, our research has developed a toolkit for online, end-to-end performance diagnosis of distributed systems, called E2EProf. The toolkit uses a modified form of time-series analysis (commonly used in Digital Signal Processing or DSP), to detect the paths taken by requests and delays incurred due to different path components. Since the toolkit does not require applications to be modified, it can also handle legacy components. Experimental evaluations show that E2EProf can detect performance bottlenecks in realistic enterprise applications, while at the same time, reducing the analysis time by an order of magnitude compared to similar techniques presented in the literature.

Our near term future work will explore other areas and applications to which the techniques presented in this paper can be applied. These include network overlays and publish-subscribe systems. Further, we have recently been able to start a collaboration with another group at Delta Air Lines that manages the Delta.com infrastructure, which is much more complex than the revenue pipeline system. Analyzing these new traces will provide us with new insights into the challenges posed by complex enterprise applications. We are also building visualization interfaces that would highlight interesting performance behaviors of service paths.

In the long term, we plan to deploy E2EProf as a basic service, 'pluggable' into any distributed system. When applications or services subscribe to its interfaces, they henceforth, will receive real-time information about their service paths and systems 'health' in general.

References

[1] M. K. Aguilera et al. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] M. Arlitt et al. A workload characterization study of the 1998 World Cup site. *IEEE Network*, 14(3):30–37, 2000.

[3] Systems Management: Application Response Measurement (ARM). `http://www.opengroup.org/products/publications/catalog/c807.htm`.

[4] P. T. Barham et al. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[5] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, September 1999.

[6] B. Cantrill et al. Dynamic Instrumentation of Production Systems. In *USENIX ATC*, 2004.

[7] E. Cecchet et al. Performance and Scalability of EJB Applications. In *OOPSLA*, 2002.

[8] M. Y. Chen et al. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[9] M. Y. Chen et al. Path-based failure and evolution management. In *NSDI*, 2004.

[10] I. Cohen et al. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004.

[11] M. Crovella et al. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *TON*, 5(6):835–846, '97.

[12] Y. Fu et al. Ete: Passive end-to-end internet service performance monitoring. In *USENIX ATC*, 2002.

[13] A. Gavrilovska et al. A Practical Approach for Zero Downtime in an Operational Info System. In *ICDCS*, 2002.

[14] J. L. Hellerstein et al. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *ICDCS*, 1999.

[15] J. L. Hellerstein et al., editors. *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

[16] S. Jaiswal et al. Inferring TCP connection characteristics through passive measurements. In *Infocom*, 2004.

[17] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales? In *Sigmetrics*, 2005.

[18] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41 – 50, January 2003.

[19] M. Mansour et al. I_RMI: Performance Isolation in Service Oriented Architectures. In *ACM Middleware*, 2005.

[20] D. L. Mills. The network computer as precision timekeeper. In *PTTI*, 1996.

[21] D. Mosberger et al. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, 1998.

[22] D. P. Olshefski, J. Nieh, and D. Agrawal. Inferring client response time at the web server. In *Sigmetrics*, 2002.

[23] R. Rajkumar et al. A resource allocation model for QoS management. In *IEEE RTSS*, 1997.

[24] P. Reynolds et al. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

[25] E. Thereska et al. Stardust: tracking activity in a distributed storage system. In *Sigmetrics*, 2006.

[26] J. Wilkes, J. Mogul, and J. Suermondt. Utilification. In *SIGOPS European Workshop*, 2004.

[27] K. Yaghmour et al. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *ATC '00*.