Eve: A Measurement-Centric Emulation Environment for Adaptive Internet Servers

Hani Jamjoom
IBM Watson Research
jamjoom@us.ibm.com

Chang-Hao Tsai
University of Michigan
chtsai@eecs.umich.edu

Kang G. Shin
University of Michigan
kgshin@eecs.umich.edu

Sharad Singhal
HP Labs
sharad@hpl.hp.com

Abstract

Emulation plays a central role in the performance evaluation, capacity planning, and workload characterization of servers and data centers. Emulation tools usually require developers to focus on mimicking application behavior as well as to deal with system-level details of managing the emulation. With the continuing increase in computing capacity and complexity, capturing the interactions between different parts of an emulation (e.g., clients' reactions to server reconfiguration) increases the complexity and overhead of emulation design. Furthermore, since the amount of measurement data can easily be huge, efficient data management is becoming a key requirement to the proper scalability of any emulation tool. In this paper, we propose *Eve*, an efficient emulation environment that provides rapid development of distributed and adaptive emulators. By incorporating *in-path data processing* and *custom triggers* into a distributed shared variable (DSV) core, Eve provides full and customizable control of how and when measurement data is moved from the source to the DSV, where the data is stored. Both functions simplify data management and minimize the overhead of frequent updates, thus enhancing the created emulator's scalability. They also simplify feedback monitoring and control when creating adaptive emulators. The capabilities of Eve are shown to allow emulation designers to focus on application behavior rather than on system-level details.

***Keywords*—Service Emulation, Distributed Emulation, Adaptive Services**

1 Introduction

The importance of emulation to system evaluation has long been recognized for accurate and repeatable experiments. However, the advent of high-capacity servers, offering rich and dynamically-created content, has elevated the complexity of proper emulation to a much higher-level. Especially in emerging data centers and consolidated server environments, emulation plays a key role in the provisioning and configuration of servers by mimicking realistic workloads, on both the client and the server sides. On the client side, emulation is used to re-create client interactions with a given service, while, on the server side, it must offer enhanced features that deal with persistent states, feedback monitoring, dynamic adaptation and reconfiguration. In this paper, we introduce *Eve*, an efficient emulation environment that provides seamless integration of signaling and data management: both are used for dynamic adaptation and management of complex clients/services.

Signaling and data management in Eve are realized by the *Distributed Shared Variable* (DSV) layer. The DSV provides a location-independent view of the underlying data and enforces data sharing at the object level, where objects represent initialization parameters, measurement data, etc. Eve integrates *in-path data manipulation* to allow for customizable conflict resolution and consistency enforcement. Eve also integrates *custom triggers* that initiate data exchange when guard conditions are met, which are essential for implementing feedback adaptation and monitoring. As we will show, both features substantially reduce the overheads (in terms of development time and running costs) that are associated with monitoring and data collection.

To illustrate its capabilities and usefulness, we use Eve to implement an a *feedback-triggered adaptive service emulator* (FASE) that is specifically targeted for testing dynamic reconfiguration of services in a consolidated server environment. This setting imposes three key criteria that cannot be easily addressed with existing tools. First, clients must maintain a persistent state and arrive at a sustained rate, i.e., do not just generate a series of uncorrelated requests. As shown in [12], client persistence can dramatically change the underlying workload. Second, the emulated services can be migrated as the load on the underlying server varies. This clearly requires feedback monitoring and control. Finally, as services

are added or removed, adaptive changes in the emulated client behavior are necessary. For instance, if a Content Distribution Network (CDN) is used to offload a portion of a server's content during very high load, emulated clients must adapt to the change (namely, issue fewer requests to static objects since these requests are no longer visible to the server). FASE has three parts: (1) a client-side emulator that relies on Eve's distributed user-threads to simultaneously stress-test multiple servers while accurately mimicking client behavior (with persistent state), (2) a server-side emulator that relies on Eve's rich set of resource abstractions to allow for the emulation of a wide variety of services, and (3) a controller that relies on the integrated signaling and data management to monitor and adapt the behaviors of clients and services.

This paper is organized as follows. We describe the general architecture of Eve in Section 2, detailing its underlying components. Section 3 discusses how Eve's flexible architecture is used to emulate end-clients and running services in FASE. In Section 4, we evaluate the effectiveness and efficiency of Eve. The paper ends with a discussion of related-work and conclusions in Sections 5 and 6, respectively.

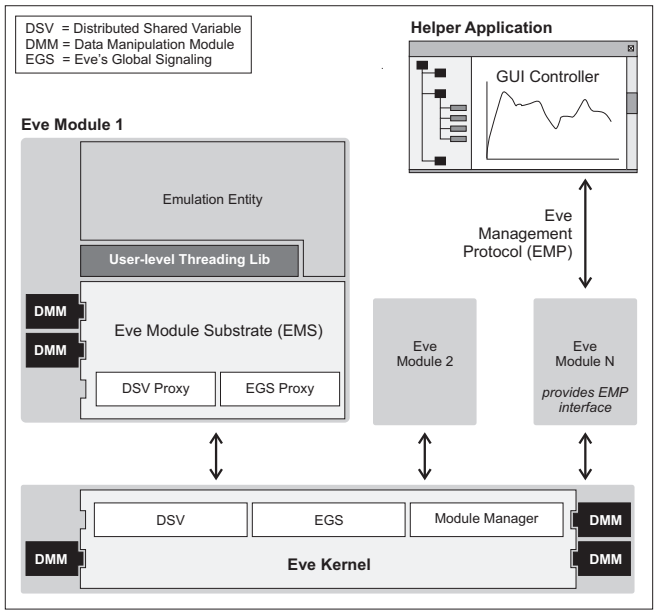

Figure 1: High-level architecture of Eve. Modules connect to Eve Kernel using a thin substrate. Both modules and Eve Kernel can customize (using plug-in DMMs) how data is managed.

2 Architecture of Eve

Figure 1 shows a high-level architecture of Eve, consisting of three basic components: a kernel, modules, and helper applications. Eve Kernel manages modules and facilitates efficient data storage. Eve modules are the basic building blocks; they can be easily connected—via *Eve Module Substrate* (EMS)—and are relatively autonomous. However, data exchange and management is done automatically by the underlying layers. Helper applications are used to further enhance the functionality of Eve, e.g., a configuration and control graphical user interface (GUI). Internally, these components combine several key features that provide Eve with its flexibility and power.

Each participating machine in Eve must run an instance of the Eve Kernel, whose primary function is to manage the running modules (illustrated by the *Module Manager* in the figure). The *Module Manager* is used for initialization, appropriately configuring the communication channels (to facilitate signaling and data exchange), and performing post-experiment cleanup. An Eve Kernel may also host the DSV and EGS (Eve's Global Signaling) functionalities to provide the necessary data and signaling exchange capabilities.

2.1 Integrated Signaling in User-Level Threads

Clearly, supporting concurrency is crucial; there are several models to choose from, each comes with its trade-offs. Motivated by [19], we used the well-tested `GNU Pth` user-level threading library [10] in our implementation and provided the appropriate wrapper functions that allow straightforward migration to emerging threading libraries like Capriccio [20]. User-level threading ensures scalability within the boundaries of a single machine. However, *inter-machine* scalability remains an important issue that is not completely addressed by the available user-level libraries. Addressing this issue requires the ability to manage as well as control distributed threads.

One key extension to the user-level threading library is *Eve's Global Signaling* (EGS). EGS is conceptually similar to existing message passing techniques. However, EGS focuses on conditional delivery of the associated signals. Essentially, each signal can be associated with an arbitrary guard condition (which can be externally defined) that dictates when a signal should be delivered. As we will show later, having these guard conditions (called *triggers*) can greatly reduce the overheads of implementing adaptive control in emulators. When designing EGS, we wanted to minimize the overhead of setting and raising a signal. This was achieved by designing a lightweight EGS component to act as a proxy in the EMS of each module. When a signal is set, it is first registered with the EGS proxy, which adds the corresponding thread to the list of interested ones. The proxy then registers itself in Eve Kernel as having interest in receiving that signal. To invoke a signal, a request is relayed through Eve Kernel to the appropriate proxies.

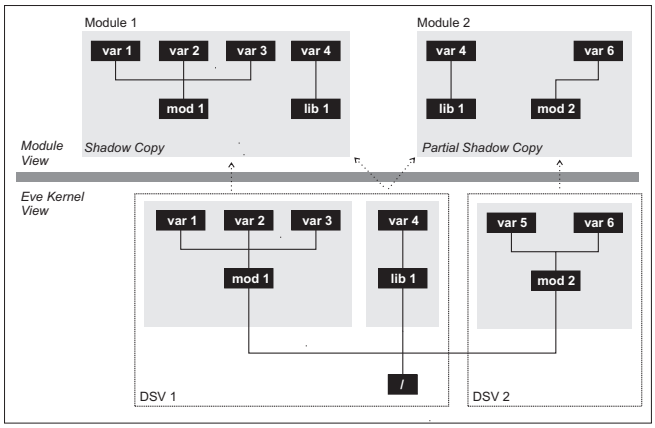

Figure 2: Variable organization inside the DSV. Libraries in the DSV are place holders (akin to abstract classes in OOP) and are used for convenience.

2.2 Integrated Data Management and Control

Eve's power lies in its flexibility and customizability when managing measurement data and enforcing memory coherence. As with any distributed shared memory system, the DSV separates data access from data management. Data access is provided by two standard calls to access shared variables. They are `eve_in` and `eve_out`, which are similar to Linda's `in` and `out` methods [5].

Similar to Munin [3], the DSV enforces sharing at the object level. All variables (or objects) have a master copy that is stored in the DSV on Eve Kernel (referred to as the *DSV server*). When the number of objects is large, multiple DSV servers can be configured to spread the master copies across multiple machines and help distribute the load (Figure 2). Objects in the DSV are organized in a two-level directory, where the first-level directory corresponds to modules or libraries and the second-level to variables. Each shared variable is identified by its full path to avoid name conflicts across different modules (e.g., `/mod1/var1` in Figure 2). That said, the DSV assumes a strong trust model (as expected in a typical emulation environment), where modules can access each other's variables.

To improve the performance of the DSV, objects are locally cached at the module level; cached objects are called *shadow copies*. The DSV uses two key principles to provide seamless access to distributed data without enforcing a prescribed model of memory coherence. The first is the ability to perform in-path data manipulation, allowing measurement data to be pre- or post-processed at the source (a module) or sink (Eve Kernel), respectively. Here, post-processing enforces customized conflict resolution. The second is the ability to associate arbitrary triggers (not just data updates) with each variable to perform data synchronization or adaptation.

In-path data manipulation allows the emulation designer to write (or reuse) custom functions that pre-process the data at the module side and then post-process them at Eve Kernel. These custom functions control how and when data is updated. When an emulation designer is interested in overriding the default behavior of the DSV (which is release consistency), s/he must define the new behavior in a *data manipulation modules (DMM)*, which may reside at either the client module or at Eve Kernel, depending on which functionality is to be overwritten (Figure 1). Each DMM must export three functions: one that overrides `eve_in`, another that overrides `eve_out`, and finally a trigger handler that defines how the shadow variables should be updated to their corresponding master version in Eve Kernel. This final function responds to triggers and is used for feedback, synchronization, and periodic updates.

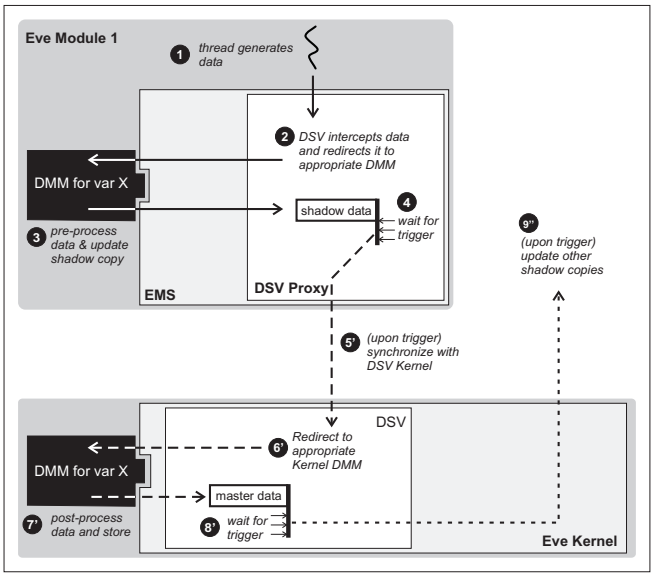

Figure 3: In-path data manipulation. The `eve_in` and `eve_out` requests are redirected to the corresponding DMM to perform pre- or post-processing. Triggers are registered to indicate when an update is performed.

Much like other parts of Eve, a DMM is configured at runtime in the configuration file. There, the DMM defines one or more variables to listen on and also defines which module to hook into. Other DMM-specific parameters can also be defined (such as an update interval, if any). Whenever the corresponding variable in the DSV is accessed (for reading or writing), the corresponding function in the interested DMM—instead of the default `eve_in` or `eve_out`—is called. The replica of this behavior is performed on Eve Kernel.

In-path data manipulation alone does not provide on-demand updates. The DSV addresses this issue by introducing the ability of customizing when updates occur in both directions: from the module to Eve Kernel and vice versa. Triggers are used to specify when an update is pushed to the mod-

ule or Eve Kernel. Triggers can represent different events, not just a change in the shadowed variable. Triggers can be specified on both the module and Eve Kernel sides. Also, a trigger in Eve Kernel can be used to periodically request (i.e., pull) an update from the modules, rather than wait for the update to be pushed.

Figure 3 highlights how in-path data manipulation and trigger updates are performed. One important argument regarding the design of these mechanisms is whether manual programming of such optimization inside each emulation entity is sufficient, eliminating the underlying complexities in the lower layers. Here, the basic question is whether complexity should be implemented at the EMS level or at the Emulation Entity level. We justify our design decisions with three reasons. First, because many data collection (and tracking) techniques share the same goal, only one version of the DMM need to be implemented. This allows better reusability of optimized code (i.e., reusing the same DMM) as well as simpler design of emulation entities. Second, the architecture of Eve allows the DMM to be plugged at Eve Kernel as well, providing even greater flexibility than only modifying the Emulation Entity. Finally, as far as performance is concerned, since we have implemented the function as part of the EMS (which uses user-level threads), the difference between manual and automatic data management is minimal.

3 Implementing a Feedback-Triggered Adaptive Service Emulator (FASE) with Eve

In this section, we motivate and describe the use of Eve for creating a feedback-triggered adaptive service emulator (FASE). Similar to traditional service emulators (e.g., SPECWeb [8]), FASE has a client-side and server-side parts. However, FASE goes a step further by capturing the potential dynamics between client and service behavior, which enables the evaluation of complex server configurations, such as testing the effects of service migration, client persistence, and content adaptation. Figure 4 gives a high-level architecture overview of FASE. FASE focuses on three important elements: (1) emulated clients can change their behavior based on their observation of server performance and content availability, (2) emulated services can be dynamically moved or configured to mimic the addition, removal, or reconfiguration of services in a multi-tiered environment, and (3) both client and service behaviors are managed via an independent controller module, which is free to implement any management or control algorithm. The remainder of this section describes the implementation of both the client- and server-sides of FASE and also a potential controller that dynamically manipulates both clients and server behaviors.

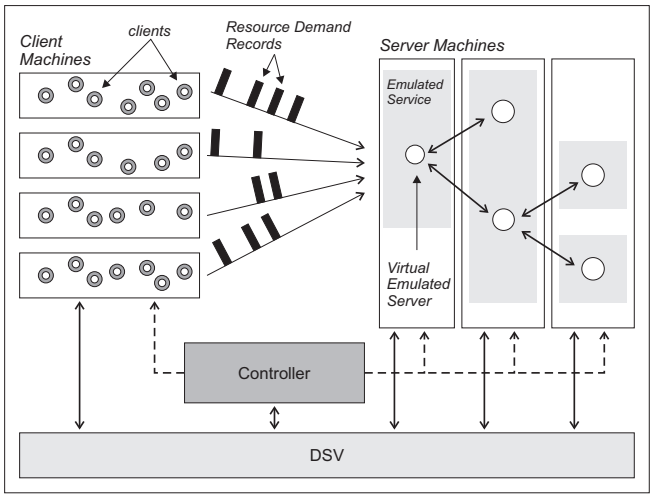

Figure 4: Architecture overview of FASE

3.1 Distributed Client Emulation in FASE

The client-side of FASE (FASE-client) is responsible for modeling real clients and generating representative requests. The basic model is based on those proposed in [1, 7, 12], where clients' accesses are divided into active and inactive periods. Our FASE-client implements the following four key features:

CF1: The clients' arrival rate is sustained regardless of the server's response. Each client is represented by a new session and maintains a persistent state; a session consists of a series of visits (active and inactive periods).

CF2: Each request consists of a Resource Demand Record (RDR), which describes the resource consumption on one or more *emulated* services. RDRs, described in Section 3.2, are used to maximize the reconfiguration flexibility of FASE. Note that clients use pre-recorded RDR trace files to decide the next request to send. To play back actual HTTP logs, regular HTTP trace files can be easily used. Probabilistic models (similar to the one used by Surge [1]) are also implemented.

CF3: Clients create a *behavior adaptation* function, which is tied to a trigger, allowing the controller to change the client behavior in response to server reconfiguration. This is needed whenever a server updates its content or introduces new services, which may change the arrival and access distributions of main and embedded objects. In Section 3.3 we detail how the controller performs this task.

CF4: Statistics regarding server throughput, in terms of request delay, is *automatically* collected. This is simplified by the DSV, where measurements are updated by all participating FASE-clients. However, whenever a variable is updated, a DMM intercepts the update request and performs in-path data

manipulation. As we will show in Section 4, this dramatically reduces the number of required writes.

3.2 Distributed Service Emulation in FASE

In many data center setups, servers are dynamically allocated and assigned to running services to improve resource utilization and management. Several algorithms have been proposed to manage data centers [6]. However, it is not trivial to evaluate these efforts. To simplify the evaluation and verification of newly-adopted management techniques, we implemented the server-side part of FASE (FASE-service) to provide repeatable and scalable emulation of arbitrary resource demands on distributed servers while simplifying the measurement process and service migration or reconfiguration. In FASE-service, there are two important design challenges. The first is to specify how resource requirements should be defined. The goal is to provide a rich syntax that can emulate services at different granularities. The second is to allow for the dynamic configuration and administration of these emulated services to fully observe how an on-line resource management algorithm would behave. To address these challenges, we have implemented FASE-service with three key features:

SF1: FASE-service is composed of two elements: *emulated services* (ESs) and *virtual emulated services* (VESs). An ES is implemented as a module in Eve. So, multiple ESs can share a single server, and similarly, multiple VESs can share a single ES. A VES is, thus, logically equivalent to Virtual Hosts in Apache. The organization of VESs within ESs and across servers can be used to emulate different configurations of applications and services.

SF2: Work is triggered by the arrival of a request, called a *Resource Demand Request* (RDR). An RDR is a variable-length structure which contains a fix-sized header and a list of *Resource Demand Instructions* (RDIs). An RDI is used to describe resource consumption on a VES and the relationship between different VESs (Figure 5). So, the behavior of the system can be controlled by the RDR generator, which simplifies testing different configurations. Two types of instructions are implemented: *resource consumption* and *flow* instructions. Each instruction contains three fields: (1) a VES ID, (2) a resource type, and (3) a data field.

Currently four resource types and nine resource consumption instructions are defined in Eve: CPU, memory (allocate, access, free), network, and disk I/O (create, read, write, delete). Each of them takes a 32-bit payload to indicate (in general) the amount of resource to consume. Details of these instructions are omitted due to space limitation.

Because dependencies between VESs can be arbitrary, we have introduced two flow instructions to control the request's flow in the system and capture resource dependencies and parallelism of typical services: `RETURN` and `FORK`. In Figure 5, for example, when C0 (which is a VES) needs two other application components C1 and C2 to complete a request, several possible relationships can exist between these three components. To solve this ambiguity, a `RETURN` instruction is introduced to mark the end of the work in one component. When the VES executes this instruction, it sends the next instruction pointer back to its caller. Otherwise, the request is sent to next components and the server waits for the next instruction pointer to be returned. To place workload on more than one component simultaneously in a specific request, a `FORK` instruction is introduced. When the server executes the `FORK` instruction, it creates two (user-level) threads to concurrently execute the two subsequences, each can contain any number of RDIs. However, at the end of each subsequence, the control returns to its parent, which was waiting for the two subsequences to complete before processing additional RDIs.

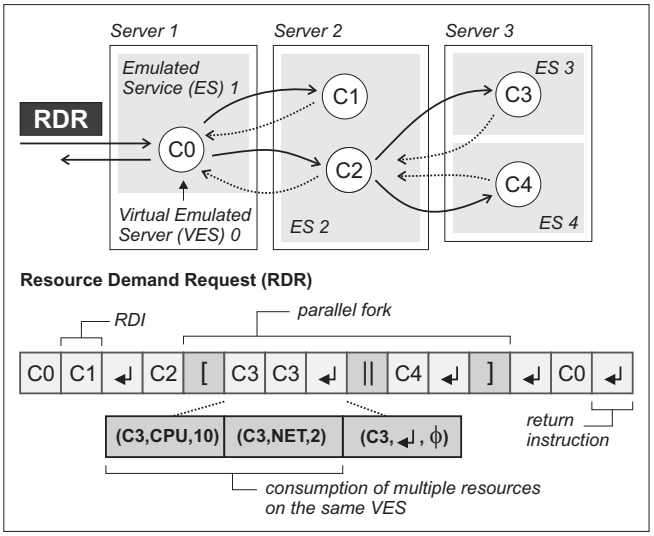

Figure 5: Example of possible node dependencies and the corresponding Resource Demand Request

SF3: The mapping between VESs and ESs is performed online. Thus, after an instruction (RDI) is executed, the current VES is also responsible for routing the corresponding request (RDR) to the next VES, which is defined in the next instruction's VES ID. The routing process involves two stages: map lookup and request delivery. It is in the map lookup stage that we used of Eve's well-managed data dissemination infrastructure. VES-to-ES mappings are stored in the DSV, where Eve provides caching of these mappings and allows map updates by other modules (e.g., controller or GUI). By re-mapping a VES to a different ES, it is effectively migrated, and both the order and location of execution can be dynami-

cally tested.

The above instructions are only building blocks that must be augmented with stochastic models to fully capture the expected behavior of applications. These stochastic models are integrated at the source, which sends the RDRs. In fact, FASE-client has the ability to integrate arbitrary probabilistic models in both request generation (i.e., the distribution of request arrivals at the system) and RDR generation (i.e., the distribution of resource requirements in each VES). Advanced service profiling to extract appropriate resource consumption models is the subject of our ongoing research.

3.3 Emulating Service Migration and Control in FASE

As with our client- and service-sides, Eve's flexibility enabled seamless integration of measurement data and feedback control to emulate service migration. As a proof of concept, we have created a controller (FASE-controller) that reads both request delay from FASE-clients and service utilization from the VESs. Since both data are stored in the DSV, it is straightforward to obtain them. Whenever the client response time exceeds the threshold for an acceptable delay, our simple service-reconfiguration algorithm is executed. The algorithm moves the most utilized VES into an empty ES, if any. Otherwise, depending on the type of the VES (e.g., mimicking static content), the algorithm may remove it (implying its move to an external CDN) and signals an adaptation in the clients' arrival behavior. To migrate the service, the controller simply changes the VES-to-ES mapping, which is stored in the DSV and determines the VES-to-ES binding. Note, however, that this simple service-migration algorithm is used to illustrate the usefulness of Eve, rather than the effectiveness of the algorithm.

4 Evaluation

Eve is implemented in C on the Linux kernel version 2.4.20, but its cross-platform porting is straightforward. In Section 3, we showed how Eve can be used to implement a feedback-triggered service emulator. Due to space limitations, we only focus on evaluating two aspects of FASE: (1) the effectiveness of using Eve to adapt client behavior and emulate service migration, and (2) the potential benefits of having customized DMMs.

We used a testbed of up to 4 identical machines (Intel Pentium 4 PC with 2.66 GHz and 1 GBytes of RDRAM) connected through a FastEthernet switch. In some tests, two additional server machines were used (Intel Pentium 4 PC with 1.7 GHz and 512 MBytes RDRAM). The first set of machines

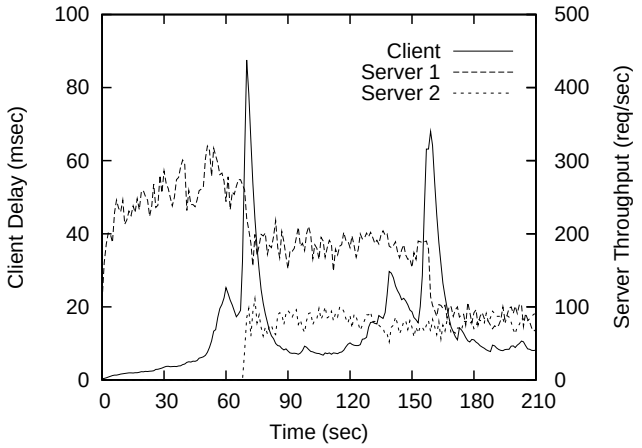

Figure 6: Adaptation example in FASE.

run FASE-client and issue RDR requests to two server machines running FASE-service, each running an instance of a VES to mimic Web-like services. The two VESs, however, had different restrictions. VES-1 can be migrated both to an internal server and to an external CDN. This is akin to an HTTP server handling static content, such as embedded objects. On the other hand, VES-2 can by replicated and migrated only to an internal server. This is akin to an HTTP server providing dynamic content. FASE-clients were configured to send a mix of requests to both servers. The request arrival rate is also gradually increased to mimic time-of-the-day effects.

Figure 6 shows the change in client delay and throughput of the running servers. At the beginning both VES-1 and VES-2 run on the same server. However, whenever the average client- perceived delay starts to increase, the adaptation algorithm is invoked. Because requests for VES-2 are configured to require more resources, the adaptation algorithm replicates VES-2 to a second available server and remaps VES-2 on two servers, which cause the load to automatically be load-balanced. This happens at t=69 sec in Figure 6, after which, we observe a drop in the throughput of server 1 as server 2 starts to handle some of the incoming load. As client load continues to increase, the adaptation algorithm is triggered once again at t=157 sec, which deletes VES-1 to mimic migrating it to an external CDN. FASE-clients are also triggered to stop sending requests to VES-1 as these request would be handled by an external server.

While space limitation prevented us from presenting the actual code, emulating a relatively complex scenario that requires both on-line monitoring and reconfiguration was greatly simplified by Eve. Furthermore, because the ability to perform in-path data manipulation, we were able to dramatically reduce data exchange to minimize its overhead and

interference on the experiments. To see effects of in-path data manipulation, we tracked the number of kernel accesses for updating the client-perceived delay while enabling different kinds of DMMs. Along with a conflict resolver DMM on the DSV, two types of client DMMs were implemented to manage the delay value. The first implements a time-based DMM, which writes back the change in the delay at a configurable rate. The second is a sample-based DMM, which writes back the change in the delay after a configurable number of samples is collected. Figure 7 shows the number and frequency of DSV accesses for the delay. Here, the actual delay values are not important; what is important is how often the DSV is accessed to update this value. Four configurations are shown: (1) when no DMMs are used (thus requiring locking, reading the content, updating the value, and unlocking of variables), (2) Eve Kernel-side DMM is used to perform customized conflict resolution (only adds the change in the value, which eliminates the need for the global lock/unlock and read operations), (3) both Eve Kernel- and Module-side DMMs are used with a sample-based trigger (updates are triggered every 10 samples), and (4) both Eve Kernel- and Module-side DMMs are used with a time-based trigger (updates are triggered every 5 seconds).

The first part of Figure 7 clearly shows the benefits of having in-path data manipulation. Obviously, by changing the trigger sample/period, the actual overhead will change. In the second part of Figure 7, the ordinate plots the actual sum of measurement samples. We note that the values for the different configurations do not match because they represent different experiments with many non-deterministic components (i.e., even with the same random seed, the server/network delay changes these values). What is important, however, is that the plot shows the change in frequency as different configurations are used; with each update, the value of the variable becomes consistent. We note here in cases 2, 3, and 4, the actual code for FASE-client did not change, but the configuration file is used to enabled or disabled the DMMs.

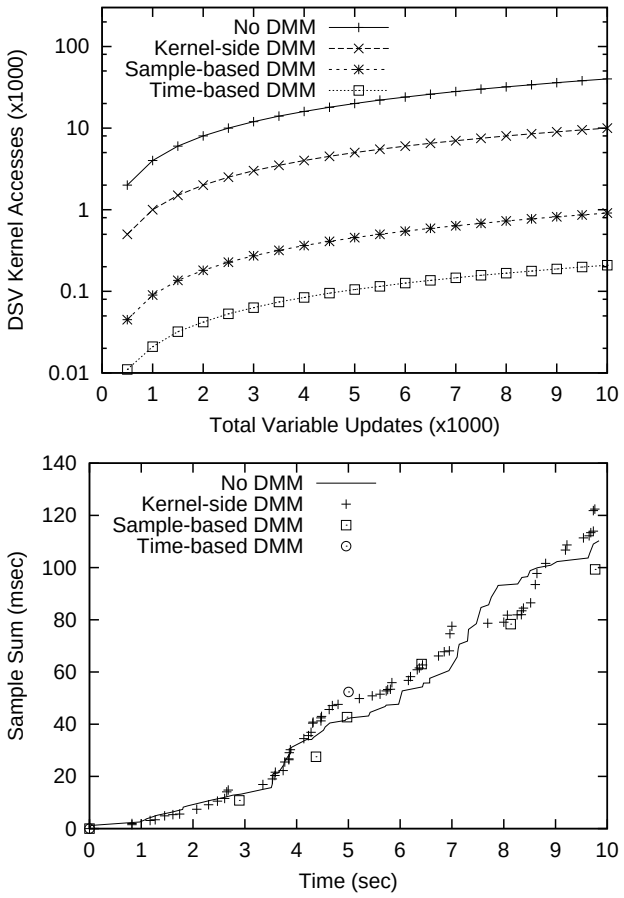

Figure 7: Benefits of in-path data manipulation: (right) number of DSV accesses (left) frequency of DSV accesses

5 Related Work

Eve uses many well-established concepts in OS and distributed systems. In many ways, our implementation of EGS resemble existing message passing mechanisms such as UNet [2], MPI [13], and PVM [11], to name a few. In EGS, however, we implemented a lightweight mechanism that is tailored to user-level threads and provides external control of message delivery. Because of the large number of potential signals, EGS would benefit from a multicast-based design, similar to [18].

The heart of Eve is its efficient data-manipulation layer. The ideas there build on existing distributed shared memory systems [3–5, 9], which have also focused on various aspects of keeping memory consistent. Our work provides fully customizable data handling similar to Astrolabe [17]. However, it also allows the implementation of reusable consistency and conflict resolution models.

We have used Eve to implement a feedback-triggered service emulator. There are several popular tools that are used for similar purposes (e.g., SPECWeb [8], Surge [1], httperf [15], LoadRunner [16]). What distinguishes Eve from the other tools is its ability to incorporate feedback control and adaptation with little development overhead. Eve is also able to achieve a high level of flexibility and customizability (similar to [14]). Tools like SPECWeb and Surge follow a black-box architecture and are created as a monolithic application. Any modification may require intimate knowledge of the tool's internal behavior. Some tools like httperf [15] do provide hooks for extension, but they tend to be limited to certain functionalities. Eve also allows custom data manipulation and centralized control, which are key for stress-testing today's powerful servers.

6 Conclusions

Emulation tools must not only be efficient, but also adaptive, extensible, accurate, and scalable. In this paper we presented *Eve*, an emulation environment that met these five design objectives. We used Eve to implement FASE, a feedback-triggered adaptive service emulator. FASE mimics the possible interactions between clients and services. While we focused on using FASE to appropriately provision server resources, the flexibility of Eve allows us to extend FASE to be used for root causes analysis. FASE-clients can be used for playing back test transactions to identify potential problems during unusually high loads, such as flash crowds or denial-of-service attacks. Malicious clients, for instance, can be constructed as plug-in modules and used to analyze the behavior of the tested services.

References

[1] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proceedings of Performance'98/ACM Sigmetrics'98*, May 1998, pp. 151–160.

[2] A. Basu, V. Buch, W. Vogels, and T. von Eicken, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[3] J. K. Bennett, J. K. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," in *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 168–176.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," in *Proceedings IEEE COMPCON Conference*. IEEE, 1993, pp. 528–537.

[5] N. Carriero, D. Gelernter, and J. Leichter, "Distributed Data Structures in Linda," in *Proc. ACM Symposium on Principles of Programming languages*, 1986, pp. 236–242.

[6] A. Chandra, W. Gong, and P. Shenoy, "Dynamic Resource Allocation for Shared Data Centers Using Online Measurements," in *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003)*, Monterey, CA, June 2003.

[7] L. Cherkasova and P. Phaal, "Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites," in *Proceedings of Seventh IWQoS*. IEEE/IFIP event, May 1999.

[8] S. D. Committee, "SPECweb," Tech. Rep., April 1996, http://www.specbench.org/osg/web/.

[9] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro, "Lightweight Logging for Lazy Release Consistent Distributed Shared Memory," in *Proceedings of the Second Symposium on Operating Systems Design and Implementations (OSDI'98)*, Seattle, Washington, October 1996.

[10] R. S. Engelschall, "GNU Pth - The GNU Portable Threads," http://www.gnu.org/software/pth.

[11] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "PVM and MPI: a Comparison of Features," *Calculateurs Paralleles*, vol. 8, no. 2, 1996.

[12] H. Jamjoom and K. G. Shin, "Persistent Dropping: An Efficient Control of Traffic Aggregates," in *Proceedings of the ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003, pp. 287–298.

[13] Message Passing Inerface Forum, "MPI-2: Extensions to the Message-Passing Interface," http://www-unix.mcs.anl.gov/mpi/.

[14] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," in *Proceedings on Symposium on Operating Systems Principles*, December 1999, pp. 217–231.

[15] D. Mosberger and T. Jin, "Httperf — A Tool for Measuring Web Server Performance," HP Research Labs, Tech. Rep.

[16] M. I. W. Paper, "Load Testing to Predict Web Performance," Mercury Interactive Corporation, Tech. Rep., www.mercuryinteractive.com.

[17] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 164–206, May 2003.

[18] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "SCRIBE: The Design of a Large-scale Event Notification Infrastructure," in *Proceedings of the Third International Workshop on Networked Group Communication*, London, UK, November 2001.

[19] R. von Behren, J. Condit, and E. Brewer, "Why Events Are a Bad Idea (for High-Concurrency Servers)," in *Proceedings of the 2003 HotOS Workshop*, May 2003.

[20] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.