Eve: A Scalable Network Client Emulator

Hani T. Jamjoom* Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{*jamjoom, kgshin*}*@eecs.umich.edu*

Abstract

Client emulation tools play a central role in the performance evaluation, capacity planning, and workload characterization of servers. However, traditional emulation tools, because of their limited scalability and extensibility, fail to keep up with servers as they increase in complexity and performance. In this paper, we propose *Eve*, a scalable client emulation tool that is capable of stress-testing powerful servers. *Eve* relies on an open and modular architecture that provides a simple and extensible programming environment. By incorporating I/O call handing into its user-thread library, *Eve* is capable of simultaneously emulating thousands of clients. Furthermore, *Eve* uses a distributed shared variable (DSV) core to facilitate communication between different clients, thus enhancing scalability to multiple machines.

1 Introduction

The importance of emulating real user demands to evaluate the performance of servers has long been recognized. However, the advent of high-capacity e-commerce servers has elevated the complexity of performance analysis to a much higher level. Not only have client models become more complicated, but also proper analysis involves an accurate emulation of thousands, if not tens of thousands, of simultaneous clients. Consequently, inaccurate measurements can cause improper provisioning of a server's capacity, which can be disastrous, especially for mission or business-critical applications. It is therefore important to develop an efficient tool for emulating client behaviors.

Figure 1 shows the architecture of a typical client emulator where a centralized manager or process uses user-defined configuration parameters to start the desired workload, represented by clients sending requests to the server. Traditionally, client emulators are built over the popular thread abstraction, where each thread represents a single client. Threading is attractive because straight-line codes (or scripts) are easily parallelized to emulate multiple clients. However, existing threading implementations become problematic when a very large number of clients are emulated because of their high resource and management overhead. Two mechanisms are commonly used to overcome this limitation. One mechanism simply limits the number of threads or clients [10]; a client waits for response from the server before issuing a new request. The authors of [4, 18] have shown that this can significantly impact the performance numbers since it does not sustain a specific request rate. Instead of threads, an event-driven design that is based on non-blocking I/O is used to sustain the request rate [4], but this incurs added complexity.

*The work in this paper is supported in part by the Saudi Arabian Ministry of Higher Education, the US National Science Foundation under Grant E1A-9806280, and Hewllet Packard's University Program

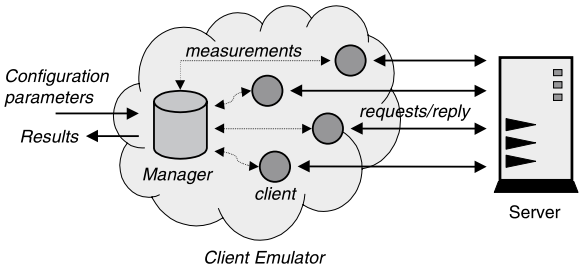

Figure 1: Request model for a typical client emulator

The heart of the design challenge is scalability. On one hand, client emulators should be powerful enough to model complex concepts such as sessions and transactions. On the other hand, thousands of such clients must be emulated in order to stress-test high-capacity servers. We believe that threading remains a good abstraction in a client emulation tool, but lacks certain optimizations for this purpose. Furthermore, an efficient communication abstraction is needed to facilitate client management, data collection, and event synchronization.

Having just the flexibility for creating and managing new clients is not enough. Emulation tools must also be "customizable." Unlike commodity applications, client emulation tools are used by researchers and system designers, and often require a large degree of application-specific customization. A monolithic application is difficult to modify and extend in order to keep in pace with the evolution of server applications. For this reason, a modular design should be used following similar design objectives to applications like Apache [13] and microkernels like exokernel [12].

When creating a client emulation tool or software, the following four objectives should therefore be considered: accuracy, efficiency, extensibility, and scalability. Simply, one would like a tool to accurately represent a specified client model. It should efficiently minimize the amount of resources needed to execute such a model. It should scale beyond the boundaries of a single machine. Finally, it should be easily extended to create new and sophisticated client models and to allow a large degree of customization. Efficiency and scalability are necessary for maximizing the number of clients that can concurrently be emulated. Accuracy and extensibility are necessary for usability.

In this paper we present *Eve*, a client emulation tool for creating client models, that meets all four goals. *Eve* consists of two powerful abstractions. The first is an optimized user-level thread library that simplifies client emulation. This library combines the ease-of-programming of traditional threading implementations with the improved performance of event-driven counterparts. The second abstraction is a distributed shared variable (DSV) architecture that simplifies the management of, and supports the scalability of, clients across multiple machines. *Eve* uses a modular design to integrate both abstractions into an extendable tool that can stress-test powerful multi-tier servers.

This paper is organized as follows. We present related work in Section 2. We give a general overview of *Eve* in Section 3 and discuss its implementation in Section 4. Section 5 gives a possible deployment example. Section 6 studies some performance issues of *Eve*. The paper ends with concluding remarks in Section 7.

2 Related Work

Table 1 contrasts the key differences between *Eve* and other popular client emulation tools with respect to accuracy, efficiency, extensibility, and scalability. What distinguishes *Eve* from the other tools is its open

Emulation Tool	Target Server	Request Model	Architecture	Client Model	Client Implementation	Extensible	Scalable
SpecWeb	Static Web	Closed	Monolithic	Fixed	Thread	No	Yes
SURGE	Static Web	Hybrid	Monolithic	Fixed	Event	Limited	No
HTTPerf	Static Web	Hybrid	Monolithic	Fixed	Thread	Limited	No
LoadRunner	General	N/A	Closed	Script-base	Thread	Limited	Yes
Eve	General	Any	Modular	Programmable	Thread	Yes	Yes

Table 1: Detailed comparison between popular emulation tools and Eve

architecture [16], where by modularizing all its components, *Eve* is able to achieve a high level of flexibility and customizability. Tools like SpecWeb [10] and SURGE [5] 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 [18] do provide hooks for extension, but they tend to be limited to certain functionality. Other extremes, typical of commercial products like LoadRunner, completely close their architectures thus limiting their extensibility.

The notion of an open architecture has been studied extensively in software engineering. Researchers as well as system designers use this approach to implementing anything from web servers like Apache [13] to microkernels like Mach [1] and Exokernel [12]. What is common among all these designs is a highly optimized core that efficiently connects various components together to allow a high degree of customization. Following this efficient communication core approach, *Eve* uses a lightweight distributed shared variable abstraction similar to that on Midway [7] and Munin [6]. Emulated clients in *Eve* are then implemented as plug-in libraries or modules that interact using this communication core.

As mentioned in Section 1, SURGE and HTTPerf implement their clients using a single process in conjunction with non-blocking I/O to sustain the offered rate. Each I/O operation is considered as a separate event, where an event can be creating a new connection, sending data, or timing out an existing connection. The single-process event-driven approach has two disadvantages. First, it does not scale easily across multiple machines and across heterogeneous client population. Second, it is error-prone since programming non-blocking I/O can be tricky.

Alternatively, one would like to use threads to simplify programming. However, traditional threading libraries have their own shortcoming: they sacrifice performance for simplicity. Generally, thread implementations can be user-level, kernel-level, or a mix of the two [20]. Kernel-level threads offer the maximum flexibility since the kernel knows when a thread blocks and can schedule another ready thread. However, this is at the expense of high overhead in terms of both resource usage and switching time. User-level threads, such as those in [2, 15, 21], on the other hand, are lighter-weight but since they rely on a smaller subset of kernel threads or processes, only a fixed number of threads are allowed to block. A mix of the two, like the one in [3], addressed this problem by providing kernel support (callbacks) to the user-level thread manager that informs it of blocking activity. Unfortunately, this scheme requires kernel modification and may be too slow to be useful when thousands of threads are emulated.

Eve-threads are a very light-weight user-level threading library that requires no special kernel support. Unlike other user-level threading libraries, it integrates I/O operations into its abstraction, which is necessary to maximize process utilization. Finally, *Eve*-threads deploy other performance enhancements that improve both measurement (time tagging) and memory consumption (stack switching).

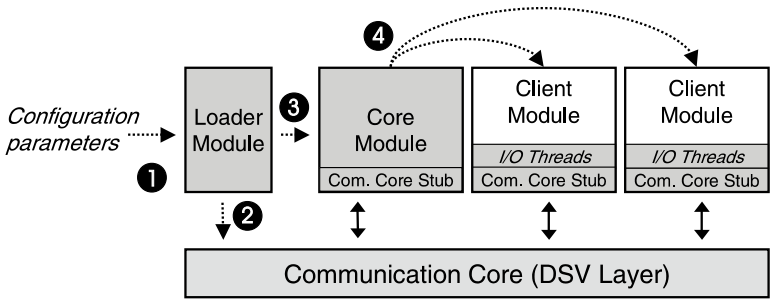

Figure 2: Design of *Eve*

3 Design

Eve is a programming environment that provides powerful mechanisms for specifying and emulating a wide range of client models as well as taking measurements. *Eve* has the capability of running as many instances of a given client model (or a combination of models) as necessary to analyze servers under test. As the need for more resources arises, *Eve* can be configured to add more machines to expand its set of usable hosts.

Figure 2 shows a high-level design of *Eve* in which it is decomposed into three basic components:

- Modules constitute the basic building blocks of *Eve*. They are connected through a standard interface, and offer a great deal of flexibility in extending and scaling *Eve*'s functionality.
- Communication Core glues all modules together and because it is distributed, *Eve* easily scales beyond the boundary of a single machine.
- I/O-Threads are used to simplify the creation of a single client. This highly optimized user-level thread library abstracts away all complexity of using non-blocking I/O without losing any performance advantage.

3.1 Eve Modules

To allow a high degree of customization, *Eve* implements all of its components, including its core components, as modules. Core modules are used to initialize all client modules, facilitate communication between these modules, and at the end of an experiment, allow for the reporting of measurements. The bare-bone *Eve* (shaded boxes in Figure 2) consists of a loader module that initializes the Communication Core, sets up environment variables based on the user-defined configuration parameters, then loads the Core modules.

A client model (e.g., one that mimics user shopping behavior) is encapsulated in a separate module and uses a standard interface, called *interface point*, to connect to other modules. A module is implemented as dynamic link library (DLL) and is loaded on demand. It shares information among other modules using the Communication Core, for example, to update or obtain measurement data. Figure 2 highlights a generic design of a client module where it shows the interaction between the client and the Communication Core, and other modules.

Client modules can be constructed to dynamically link other modules that provide specific functionality. For example, the request inter-arrival time generator is implemented as a separate module. Because it is linked at run-time, only the configuration file needs to be changed to modify the behavior of clients.

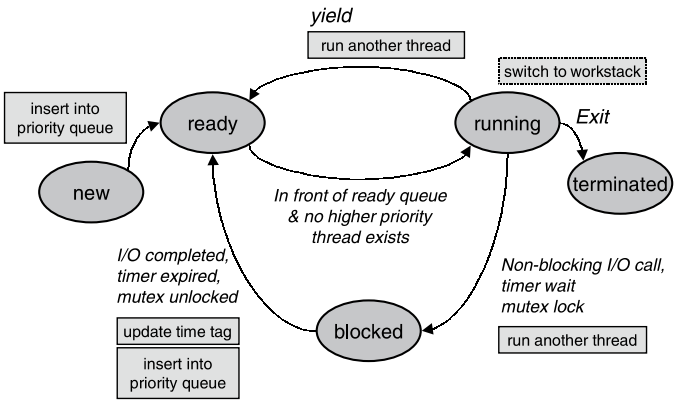

Figure 3: State transition diagram. Transition conditions are represented in italicized text. Transition actions of the scheduler are represented in boxed text.

3.2 Communication Core

Eve implements its communication core using a distributed shared variable (DSV) layer (Figure 2). It is used by individual clients to communicate among themselves to share data and perform synchronization. Since a typical client emulation environment data is mostly shared at the beginning, during the initialization stage, and at the end, during the statistic collection stage, with the occasional data sharing during synchronization, a simplified version of Munin [6] is used to implement this layer. Our design follows a client/server model, where modules (the clients) use a thin stub to establishes a communication channel to the DSV repository (the server). If a module implements multiple threads, then all threads share this single communication channel.

Eve provides two standard calls to access shared and synchronization variables. They are `eve_in` and `eve_out`, which are a restricted version of Linda's `in` and `out` methods [8]. This abstraction hides the majority of the underlying variable management from the programmer while unifying the access method to all shared variables. Release consistency is chosen based on the observation that most statistical values are updated only at the end of a simulation or at relatively coarse progress snapshots. Enforcing stronger consistency would unnecessarily increase overhead.

To allow synchronization, *Eve* uses two other functions `eve_lock` and `eve_unlock`. These provide a simple *mutex* context. A client that tries to lock a variable that is already locked by another client will block until that client variable is unlocked. If more than one client are waiting for a lock, then the highest-priority one will run first. *Eve* supports priority inheritance to prevent priority inversion.

3.3 I/O-Threads

The thread abstraction, because of its simplicity, is an ideal solution for emulating a wide variety of client modules. However, one must be careful about the design of the thread architecture when thousands of simultaneous threads must be supported. For example, if clients time out after 30 seconds, to sustain a rate of 200 requests/sec (which can be handled by off-the-shelf web servers), then the worst-case number of needed threads is $200 \times 30 = 6000$ threads, one thread for each outstanding request.

Eve implements a lightweight thread library where each new request or client can be easily handled by a separate thread. With potentially thousands of threads that need to be supported, a natural question is why would our scheme work while others don't? We identify five key design decisions enabled us to answer this

question:

- **Integrate I/O handling into the design of the thread library to provide greater control over potentially expensive I/O calls:** By transforming blocking I/O calls into non-blocking ones, *Eve* can schedule and execute a different task. Whenever the I/O call is completed, the calling thread is then placed in a ready queue (Figure 3). These calls are then a natural place where a thread would yield to another ready thread.
- **Enforce cooperative or non-preemptive multitasking [20] to maximize processor usage time and minimizes switch overhead:** Implementing cooperative multitasking was a natural consequence of using non-blocking I/O. By avoiding the use of preemption, we eliminated the need for signals, synchronization (to lock thread control block structures) as well as the need to support reentrant functions. While implementing preemptive multitasking, as opposed to cooperative multitasking, is not difficult, as we will see later, doing so would have prevented other optimizations. One may wonder if cooperative multitasking negatively influences the accuracy of real-time events. While this is true to some extent, usual client models are I/O-heavy, implying a short period between two scheduling operations.
- **Support priority queues to give soft real-time tasks, such as timers, priority over other tasks:** Supporting a large number of threads requires more than just a simple FIFO scheduler, typical of most user-level thread implementations. Since some tasks, especially timeouts, should be handled immediately, priority scheduling was necessary to support these soft real-time requirements [17].
- **Use *time tagging* to timestamp when a thread is ready; this way the thread can compensate for the long wait caused by supporting a large number of threads:** The time delay between two scheduling-slices may be long for a large number of threads. For example, if there are 1000 ready threads and each thread takes 1 msec, then there is an average thread delay of 0.5 sec. This delay can easily degrade accuracy of measurement and is common in most client emulators. To minimize this inaccuracy, *Eve* uses *time tagging* where each thread is tagged with the time when it became ready. A simple call can then be used to retrieve this time and can be easily incorporated into the client code.
- **Minimize resource and memory consumption per thread to support a large number of threads. *Eve* provides a *work stack* for functions that require large stack space.** Memory usage can explode easily when supporting a large number of threads. *Eve* uses a 4KB stack per thread. Smaller stacks can also be used. While we recognize that 4KB may not be enough in some cases, *Eve* introduces a large, shared, *work stack* that can be used during one scheduling-slice of any thread. Since threads are only switched at specific instances, they can safely take advantage of the larger stack space. This is another advantage for cooperative multitasking.

As we will show in the evaluation section, the above five features enabled us to maximize efficiency while reducing overhead.

4 Implementation

Eve is implemented using the C programming language on the Linux operating system (kernel version 2.2.14). With the exception of a single header file, cross-platform porting is straightforward. Assuming that the reader has prior knowledge of basic concepts behind threading [20] and DSV [22], we only describe the parts that differ from traditional implementations. In general, we provide a high-level description of *Eve*'s components but focus on their integration. Because of our layered design, we give a bottom-up description of our implementation (i.e., I/O-threads, Communication Core, and Modules). This will give better insight into the inter-working of each component.

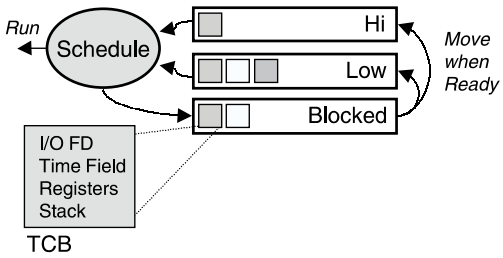

Figure 4: *Eve* scheduler

4.1 I/O-Thread

The implementation of I/O-threads library complies with the POSIX threads standard [14].[1] Essentially, the library provides calls to create or kill a thread, and yield to another thread: `eve_create`, `eve_exit`, and `eve_yield`, respectively. A Thread Control Block (TCB) is associated with each thread (Figure 4) that holds four elements:

- a memory pointer that points to the thread stack region in the heap and is allocated when the thread is created,
- CPU registers including the current program counter (PC) and stack pointer (SP),
- a file descriptor (FD) that indicates the threads' interest in performing an I/O operation, and
- a time field (TF) that is used to time out an I/O operation, implement event timers, and support time tagging.

Creating a thread is straightforward. When `eve_create` is called, the registers of the calling thread are saved in its TCB and a new TCB and local stack is created for the new thread. The new stack is then loaded with the created thread's function arguments and return function pointer; the SP is updated with the new stack's value. The PC is also updated with the new function to run; finally, a long jump is executed.

Figure 4 shows the architecture of our scheduler which also implements priority scheduling. *Eve* has two priorities: *hi* for running soft real-time tasks, such as timers, and *low* for running everything else. Whenever a thread issues a `yield` call, the scheduler is executed to insert the yielding thread in the appropriate queue and to determine the next thread to run. *Eve*'s scheduler is stateless. The actual code is an inline function that uses the yielding thread's stack space to complete its operations. This approach minimizes the number of context switches, but requires storing the priority queues in the global address space.

I/O-threads integrate non-blocking I/O into its abstraction to maximize its throughput. All I/O calls are implemented as wrapper-functions that will issue the I/O request, set the corresponding FD and TF fields in the TCB, and then cause the calling thread to yield to another one. Currently, *Eve* supports commonly-used I/O function: `accept`, `connect`, `send`, `receive`, `read`, `write`, `open`, and `close`. Timers are implemented similarly by only setting the TF filed.

A thread can block for three reasons: waiting for I/O to complete, for a timer to expire, or for a mutex to unlock (Figure 3). The scheduler uses the `select` system call to determine if an I/O function has completed or a timeout has expired. If either condition is met, the corresponding threads are then moved to the ready queue. Using `select` has its disadvantages. As shown in [19], Real-Time Signals or `/dev/poll` can significantly improve system performance. Unfortunately, both methods either lack some necessary capabilities (e.g., implementing `connect`) or lack support by other operating systems. Hence, we will migrate to one of these schemes as they mature.

[1] The time of this writing, signals are not fully supported

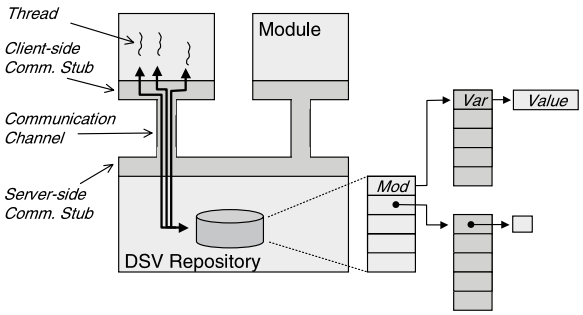

Figure 5: Architecture of the Communication Core

When a large number of threads are supported, a delay may occur between the time the thread is ready and the time it is run. This delay is characterized by the average queuing time of all threads, which can negatively affect time measurements that are performed by threads. I/O-threads solve this problem by implementing *time tagging* where the scheduler overwrites the TF field with time that the thread was moved to the ready state. This value can be accessed via `eve_gettimeofready()`. While this is only an approximation to the precise time that the I/O has completed, time tagging reduces the error from the average queuing time of all threads to the average running time of a single thread.

As mentioned in Section 3, to reduce memory allocated for thread stacks, *Eve* provides a large work stack allocated at the initialization stage that can be used by any thread, in addition to small per-thread stack. When a thread is running, it can switch between the two stacks via the macro function `eve_stackswitch()` that switches the stack pointer (SP) between the thread's local stack and the large shared work memory. Local variables are allocated on the local stack, but continue to be accessible since x86 processors reference them using the stack base pointer (BP). Data on the work stack is only guaranteed to be valid during a single scheduling slice (i.e., between two `yield` calls). Since we are using cooperative multitasking, we are guaranteed that no other thread will corrupt the stack. This allows us to allocate very minimal per-thread stacks (allowing systems to scale to large number of threads), but still allow complex execution that requires large runtime stacks.

4.2 Variable Sharing Across Modules

The Communication Core follows a client/server design where the server process, CC-server, is initialized to listen to a well-known server TCP port. Modules (e.g., clients) will initiate a socket connection to the server process, which, upon success, will cause the server process to create a separate worker-thread (using I/O-threads) to handle the module's requests. Unfortunately, sockets incur a higher initialization overhead and slower message delivery for local calls than optimized IPC alternatives. However, unifying the communication architecture (local vs. remote calls) allowed a machine-independent implementation that is also cross-platform portable.

Variables are managed centrally by the CC-server. They are organized in a two-level directory where the first-level directory corresponds to the module names and the second-level to the variable names (Figure 5). Each shared variable is thus identified by the tuple *(module name, variable name)*. Using the module name as an identifier avoided name conflicts across different modules. *Eve* assumes a strong trust model where modules can access variables from other modules by correctly specifying the corresponding values in the variable identifier.

Accessing and modifying shared variables are done via `eve_in`, and `eve_out`. These calls provide a

Module	Description
Manager	*Single and multi-host client manager*
HTTP	*Supports HTTP 1.0 & partial 1.1 functionality*
TCP	*Implements closed, hybrid connection models for TCP clients*
UDP	*Open-loop for UDP clients*
DoS	*Implements denial of service SYN-attack and ICMP-flood*
Distributions	*Supports Uniform, Poisson, Pareto, Weibull, Zipf dist. & log file based replays*
Statistics	*Simplifies measurements processing (mean, var, confidence)*

Table 2: Currently supported modules

simple, unified approach to access all data. Additionally, `eve_add` and `eve_remove` are used to add and remove module or variable directory entries. Each operation is atomic in the sense that the calling thread will block until the corresponding operation has completed. Threads can freely share a single connection to the CC-server since only one thread has access to the connection at a time. Eager release consistency [22] is implemented where `eve_lock` and `eve_unlock` correspond to the traditional *acquire* and *release* synchronization primitives. When a variable is unlocked, it will immediately update the values in the central repository. Figure 6 shows an example of how shared variables can be used.

4.3 Connecting Modules

Eve modules are implemented as plug-in dynamic link libraries (DLLs) where each DLL is required to implement two interface points, `eve_init` and `eve_exit`, for initialization and clean-up, respectively. Modules are free to export other functions to provide added functionality. A unique identifier is associated with each module and is part of the modules configuration parameters. This identifier allows modules to reference each other's shared variables as described in Section 4.2. Furthermore, the identifier is used to load and unload modules using the `eve_load` and `eve_unload` primitives, respectively.

Upon successful initialization, a module is directly connected to the CC-server and can read its configuration parameters that were installed by the loader module. Once all modules are initialized, then an experiment can be started. The process of initializing *Eve* is relatively straightforward in the single host configuration (highlighted by the numbered circles in Figure 2). When multiple hosts are used, one host must be designated as the master to load the configuration parameters and synchronize the beginning and end of the experiment with other hosts. The master also hosts the CC-server process. Once the master installs the experiment parameters, the other hosts start the necessary clients and then wait for the completion of the experiment. Using a master/daemon approach provided *Eve* with a central point of control.

Table 2 summarizes the current modules that are supported by *Eve*. While existing modules provide adequate functionality, our focus in this paper was to provide an architecture that can extend easily to match the functionality of mainstream client emulation tools.

5 An Example: Simple Client

We illustrate the applicability of *Eve* with the following example that combines common features of typical client emulation tools. Our example follows models similar to those proposed in [5, 9] where clients' access behaviors are divided into active and inactive periods. An active period represents a client clicking on a link followed by the browser requesting the document and its embedded objects relatively quickly (basically, as

```
1   int NEW_REQUEST() {                                  // Connect to server, compose msg, send it, & get reply
2     eve_connect ();
3         // Read cookie. Assume that server name = domain name and it is passed to the SIMPLE_CLIENT as an argument
4     eve_out(server_name, cookie);
5         // Include cookie in message. In our implementation of compose_next_message (not shown), request follow zipf distribution.
6     message = compose_next_message(cookie);
7         // Send message and receive response. A timeout value is also associated with both operations. Here a constant value is passed.
8     if (eve_send (sfd, message, TIMEOUT) < 0 ) return -1;
9     if (eve_receive (sfd, message, TIMEOUT) < 0) return -1;
10        // Set the Cookie if available.
11    cookie = extract_cookie(message);
12    eve_in(server_name, cookie);
13    return 0; }

14  void SIMPLE_CLIENT () {
15    session_length = pareto(ALPHA_1,BETA_1);           // Use Pareto distribution for session length.
16    while ( session_length-- ) {
17      active_period_length = pareto(ALPHA_2,BETA_2);   // Use Pareto distribution for active period length.
18      while ( active_period_length-- ) {
19        next_request_time = weibull(ALPHA_3,BETA_3);   // Use Weibull distribution for request inter-arrival times.
20        eve_sleep( next_request_time );
21        if ( new_request() < 0) eve_exit(0);           // Get request and check if successful.
22      }
23      inactive_period_length = pareto(ALPHA_4,BETA_4); // Use Pareto distribution for inactive period length.
24      eve_sleep(inactive_period_length);
25    }
26    eve_lock("sc", "session_count" );                  // lock the shared variable
27    eve_in("sc", "session_count", &cnt);               // read recent value
28    cnt++;
29    eve_out("sc", "session_count", &cnt );             // update value
30    eve_unlock("sc", "session_count"); }               // unlock the shared variable
31    else eve_exit(0); }

32  void EVE_INIT () {
33    eve_add("sc", "session_count");                    // initialize shared variables session_count
34    while ( 1 ) {
35      next_request_time = poisson(LABMDA);             // Use Poisson distribution for session inter-arrival times
36      eve_sleep(next_request_time);
37      eve_create(simple_client);                       // Create a new thread for every session
38    }}
```

Figure 6: C-Code like implementation of simple client that support sessions and cookies. This program omits irrelevant function parameters as well as error handling code.

fast as the browser can parse the requested web file and start new TCP connections). In contrast, inactive periods represent clients' think times which are normally much longer than the active period. We call the combination of an active and inactive period a *request episode*. Therefore, as a client browses through a web server, s/he is generating a series of request episodes which are further defined as a *client session*. If a request from a given session is unsuccessful, the entire session is aborted.

Our simple client (Figure 6) implements the following four important features.

- A client arrival rate is sustained regardless of server response. Each client is represented by a new session. Each session consists of a series of request episodes (active and inactive periods).
- A different probability distribution is associated with client arrivals, session length, request inter-arrival times of an active period, number of requests in an active period, and the length of an inactive period.
- Cookies are supported so that servers can distinguish between client sessions.

- Statistics regarding server throughput in terms of sessions/sec are collected.

Figure 6 shows a slightly-modified version of the actual code that is used to implement the client. To improve readability, we omit confusing function parameters and error handling code. The following four subsections detail each of the four features while highlighting the usefulness of *Eve*.

5.1 Session Support

Supporting sessions in *Eve* is trivial. Basically, `next_request_time` (Figure 6 line 35) determines the session's arrival rate — implemented here to follow a Poisson distribution — and a new thread is associated with every session. The function `SIMPLE_CLIENT` uses a Pareto distribution (Figure 6 line 15) to determine the client's session length (i.e., number of request episodes). In each request episode, a series of requests are issued (active period) (Figure 6 lines 18-21) followed by an inactive period (Figure 6 lines 23-24). If a request is not successful, the entire session is aborted.

5.2 Distributions

Eve supports a variety of statistical distributions: Uniform, Poisson, Pareto, Weibull, and Zipf. Other distributions can be easily integrated into *Eve*. In our example, the client uses Pareto, Weibull, and Poisson distributions to generate session length (Figure 6 line 15), active period length (Figure 6 line 17), inactive period length (Figure 6 line 23), request inter-arrival times (Figure 6 line 19), and session inter-arrival times (Figure 6 line 35). In each case, the distribution parameters are defined in the user-defined configuration parameters which are automatically stored in the DSV repository upon initialization. *Eve* is not limited to a statistical distribution to generate different client behaviors. A client can be configured to use a trace replay module to base its behavior on pre-recorded traces.

5.3 Cookie Support

Cookies are signatures which are stored on host machines that allow servers to remember the "state" of previous requests from the corresponding host. Cookies are primarily used in e-commerce transactions, remembering user preferences and tracking user sessions. If Cookies were to be supported, then subsequent requests to a domain (identified in the server's reply) must include the same Cookie. In a typical emulation tool, once a Cookie and the corresponding domain are extracted, they are distributed to all clients connected to the same server.

DSV in *Eve* simplifies this task by having each Cookie stored in a separate shared variable. The first client that obtains the Cookie would add a shared variable with the domain name of the Cookie as the variable identifier (Figure 6 lines 11-12). Other clients can get the value of the Cookie by issuing a read request, `eve_in`, on the domain name (Figure 6 line 4). In our example, we assume the availability of functions that extract from the server's reply and add it to a request packet. In addition, it assumes that the domain name is identical to the server name, although it can be a subset.

5.4 Collecting Statistics

Collecting statistics is simplified by the DSV where measurements can be updated by all participating clients. In our example, we were interested in tracking the number of successful sessions. Therefore, upon completion of a successful session, the client increments the corresponding shared variable before ex-

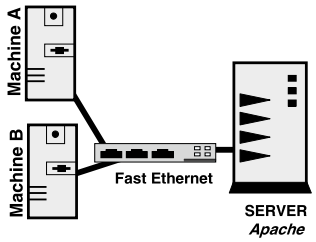

Figure 7: Testbed

iting (Figure 6 lines 26-30). Additional statistics can also be defined to indicate other variables such as the throughput in number of requests, response time, number of failed connections, etc.

What is important about this simple example is that the client's code does not change even in situations where clients are distributed across multiple hosts. Furthermore, *Eve* has successfully hidden away most of the complexity of managing distributed clients, scaling clients to stress-test powerful servers, and collecting measurements.

6 Evaluation

We studied two aspects of *Eve*'s performance: basic operation overhead and maximum number of supported clients. We used the testbed in Figure 7 to study *Eve* in a well-controlled environment. In all testing scenarios a server machine (Intel Pentium-based PC with 1.7 GHz and 512 MBytes memory) was running off-the-shelf Apache 1.3 web server. Two client machines (Pentium-based PCs 600 MHz with 512 MBytes memory) connect to the server through a FastEthernet switch. These machines are used to test the performance of *Eve* in both single and multi-host configurations. Since we only focus on the performance of *Eve*, client requests follow a Poisson-distributed inter-arrival times, allowing our measurements to reach the desired confidence level relatively quickly. Alternatively, using a heavy-tail distribution, such as Pareto, would require a very long time for measurements to converge [11]. Measurements were repeated until the estimated error was under 5% (with 95% confidence). Also, all requests were made to a single static file to maximize the throughput of the server.

6.1 Operation Overhead

Table 3 shows the basic overhead for common *Eve* operations. As shown in the table, the cost of both creating and switching between threads is very small, thus enabling support of a large number of simultaneous threads. I/O operation, however, requires more system resources partly because of our implementation (i.e., using `select`). However, even in such a case, it is relatively easy to support 1000 threads.

Accessing shared variables is the most expensive operations because of our use of socket streams. While other implementations, such as local IPC calls, may reduce the overhead for local DSV accesses, it can not dramatically improve the DSV's performance in multi-host configurations. The need for strong consistency is the primary reason behind this overhead. For a multi-host configuration, data updates must be performed at the central server, which unfortunately also reduces the benefits of automatic data caching. We therefore recommend designing clients to limit DSV access to the beginning and end of an experiment and minimally access the DSV during the experiment.

Operation	Cost
Creating a thread	4 usec
Switching between threads	0.63 usec
Average I/O call	25 usec
DSV initialization	340 usec
DSV access (server)	0.62 usec
DSV access (local)	100 usec
DSV access (remote)	210 usec

Table 3: Cost of operations in *Eve*

6.2 Client Support

We measured the resource requirements in terms of CPU utilization and memory requirements as the number of simultaneous clients is increased. These measurements were collected from the `/proc` file system in Linux. Figure 8 shows the resource requirements on a single machine (Pentium 600 MHz with 512 MBytes of RAM). The figure shows two important points:

- The resource requirements grow linearly with the number of simultaneous connections. A linear fit showed the cost of a single connection as 0.05% of the CPU and 29 KBytes of memory. Since the CPU was the bottleneck in our test scenario, we did not evaluate *Eve*'s behavior when memory starts thrashing.
- The number of simultaneous connections is estimated as *offered load* $\times$ *average response time*. Therefore, the offered load is inversely proportional to the server's response time. For example, if requests have an average response time of 8 sec, then only 200 reqs/sec can be sustained by *Eve*.

Unfortunately, *Eve* does not provide an automatic method for detecting the maximum number of simultaneous clients that a host is able to support. This is basically done using a trial-and-error technique. However, since the design of *Eve* easily scales to multiple hosts, the process of increasing the number of simultaneous requests is relatively straightforward: starting daemon process on remote hosts and changing a single configuration file. Neither the client models nor data measurements and collection needs to be changed.

7 Conclusions and Future Work

Client emulation tools must not only be simple to use, but also accurate and scalable. They must also be extensible to keep up with evolving server applications. In this paper we presented a new tool called *Eve* which met these four design objectives. This was the result of the integration of three design decisions. First, I/O-threads allowed client models to be implemented using traditional straight-line code and run with little modifications. Second, distributed shared variables simplified communication between various components of *Eve*. Finally, *Eve*'s modular design allowed for greater extensibility as well as application-specific customization.

We are currently working on enhancing the functionality of *Eve*. In its current state, *Eve* does not profile the overhead of clients, nor does it determine the maximum number of clients that can run on each machine. Both operations mush be performed manually. In heterogeneous client machines, it is desirable to automatically profile each machine to maximize its throughput. Some internal improvements that can also

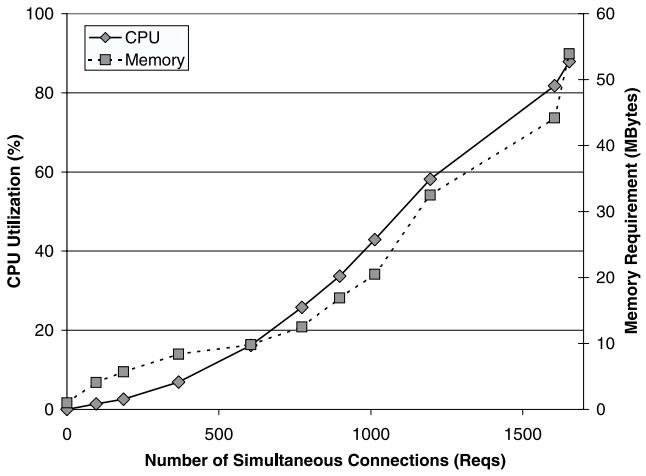

Figure 8: Client Load

optimize the overall throughput of the system. We are currently improving the message exchange between different components, i.e., using local IPC methods instead of standard sockets. We are also researching alternative implementations to the `select` system call in I/O-threads, e.g., using `/dev/poll`.

8 Acknowledgements

We would like to acknowledge helpful discussions with, useful comments from, and the support of Sharad Singhal of HP Research Labs, and Padmanabhan Pillai of Real-Time Computing Laboratory, University of Michigan.

References

[1] ACCETTA, B., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer 1986 USENIX Conference* (1986), pp. 93–112.

[2] ANDERSON, T. E. Fastthreads user's manual. Tech. rep., University of Washington, January 1990.

[3] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. E., AND LEVY, H. W. Scheduler Activations: Effective Kernel Support for the User-Level management of Parallelism. *ACM Transactions on Computer systems 10*, 1 (February 1992), 53–79.

[4] BANGA, G., AND DRUSCHEL, P. Measuring the capacity of a web server. In *Proceedings of The USENIX Symposium on Internet Technologies and Systems* (December 1997).

[5] BARFORD, P., AND CROVELLA, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *In Proceedings of Performance'98/ACM Sigmetrics'98* (May 1998), pp. 151–160.

[6] BENNETT, J. K., CARTER, J. K., AND ZWAENEPOEL, W. 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.

[7] BERSHAD, B. N., ZEKAUSKAS, M. J., AND SAWDON, W. A. The Midway Distributed Shared memory System. In *Proceedings IEEE COMPCON Conference* (1993), IEEE, pp. 528–537.

[8] CARRIERO, N., GELERNTER, D., AND LEICHTER, J. Distributed Data Structures in Linda. In *Proc. ACM Symposium on Principles of Programming languages* (1986), pp. 236–242.

[9] CHERKASOVA, L., AND PHAAL, P. Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites. In *Proceedings of Seventh International Workshop on Quality of Service* (May 1999), IEEE/IFIP event.

[10] COMMITTEE, S. D. SPECweb . Tech. rep., April 1996. http://www.specbench.org/osg/web/.

[11] CROVELLA, M., AND LIPSKY, L. ong-Lasting Transient Conditions in Simulations with Heavy-Tailed Workloads. In *In Proceedings of the 1997 Winter Simulation Conference* (1997), pp. 1005–1012.

[12] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. W. Exokernel: An Operating System Architecture for Application-Level Resource management. In *In Proceedings of the 1995 ACM Symposium on Operating System Principles* (December 1995), ACM, pp. 251–266.

[13] FIELDING, R. T., AND KAISER, G. E. The Apache HTTP server project. 88–90.

[14] GALLMEISTER, B. O. *POSIX.4 Programming for the Real World*. O'Reilly and Associates, Inc., 1995.

[15] HAINES, M. On designing lightweight threads for substrate software. In *Proceedings of the 1997 Annual Technical Conference* (1997), USENIX, pp. 243–255.

[16] KICZALES, G., LAMPING, J., LOPES, C. V., MENDHEKAR, A., AND MURPHY, G. Open Implementation Design Guidelines . In *International Conference on Software Engineering* (May 1997), pp. 481–490.

[17] KRISHNA, C. M., AND SHIN, K. G. *Real-Time Systems*. The McGraw-Hill Companies, Inc., 1997.

[18] MOSBERGER, D., AND JIN, T. Httperf — A Tool for Measuring Web Server Performance. Tech. rep., HP Research Labs. http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.

[19] PROVOS, N., AND LEVER, C. Scalable Network I/O in Linux. In *Proceedings of the USENIX Technical Conference, FREENIX track* (June 2000).

[20] SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. *Applied Operating System Concepts*. John Wiley and Sons, Inc., 2000.

[21] STEIN, D., AND SHAH, D. Implementing Lightweight Threads. In *In Proceedings of 1992 USENIX Summer conference* (1992), pp. 1–9.

[22] TANENBAUM, A. S. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.