# Making Events Less Slippery With *eel*

Ryan Cunningham and Eddie Kohler
*University of California, Los Angeles*
*rcunning@gmail.com, kohler@cs.ucla.edu*

## Abstract

Event-driven programming divides a program's logical control flow into a series of callback functions, making its behavior difficult to follow. However, current program analysis techniques can preserve the event model while making event-driven code easier to read, write, debug and maintain. We designed the Explicit Event Library (*libeel*) to be amenable to program analysis, and created tools to graphically expose control flow, verify resource safety properties, and simplify debugging. The result sustains the advantages of event-driven programming while adding the important advantage of programmability.

## 1 Introduction

Coping with asynchronous events generated by unpredictable sources is a fundamental systems problem, with two fundamentally dual solutions [12]: threads and event-driven programming. Despite controversy old and new [8, 14, 18, 19], both models have their place—and in particular, event-driven programming is here to stay. In some contexts, such as interrupt handlers and embedded systems, a connection-oriented thread model doesn't fit the problem or isn't supported by underlying layers. In others, such as Web serving, event-driven programs achieve the best published performance [11, 17] and expose important information, such as blocking points [8].

Unfortunately, event-driven programs remain difficult to understand. Control flow is divided into many cooperatively-scheduled callback functions, obscuring context and programmer intent. This makes it hard to write event-driven programs and, worse, hard to analyze and debug them when they go wrong. Although threaded programs have their own difficulties, particularly with synchronization, threading doesn't obfuscate programs in the same way. So are threads the only model suitable for dependable software? Put another way, must tools for improving event-driven programmability "effectively duplicate the syntax and run-time behavior of threads" [18]?

We show that current program analysis techniques can preserve the event-driven programming model while making event-driven programs easier to read, write, debug, and maintain. We designed a simple event library—*libeel*, the Explicit Event Library—to be amenable to program analysis. All relevant arguments are presented directly to the library, rather than stored in heap structures requiring pointer analysis. Also, a *group identifier* argument encourages the programmer to group callbacks dealing with the same conceptual connection, enabling easy discovery of the program's logical control flow. With the help of this library, we built tools that graphically expose the event-driven control flow; that verify program properties, such as that all resources allocated on a path are freed; and that simplify debugging. Two programs, crawl-0.4 [15] and plb-0.3 [5], were ported to *libeel* from the *libevent* library [16]. The *eel* tools helped us understand these programs and uncovered several bugs, while preserving the advantages of event-driven programming.

Our contributions are *libeel*, an event notification library that facilitates readable programming and (through its group identifiers) easy analysis, and the *eelstatechart*, *eelverify*, and *eelgdb* tools built above it.

## 2 Event Programming

This section explores some typical event-driven code for fetching an HTTP document, demonstrating common problems with event-driven software's readability, writability, and debuggability. The code is in Figure 1.

First, we try to understand the code. The control path clearly proceeds from `http_fetch` to `readheadercb` following a read readiness event, or to `timeoutcb` after a timeout expiration. However, it is not clear what happens following the return on line 29. One would have to read the function `http_parseheader`, and any functions it calls, in order to determine the next callback in the chain, if any. Determining the control flow of event-driven programs often requires reading the entire function call graph to assemble the callback chain.

Determining where files, memory, and other resources are reclaimed also becomes a complicated process. Callback functions can allocate either local resources, which last only as long as the callback function itself, or long-lived resources, which are passed to the next callback as part of the connection state. Furthermore, one callback function can free resources passed to it by a prior callback. When reading the code, it's difficult to tell how resources should be categorized—and, for example, whether the absence of a "free" function represents a memory leak.

"Stack ripping" [6] makes this even worse. When a sequential, blocking function is modified to wait for an event, it must move all of its relevant state information, possibly including stack variables, to the heap structure passed to the next callback. For example, Figure 1's line 6 writes an HTTP request to a file descriptor using a nor-

```
1  // assume uri->fd is ready for write
2  void http_fetch(struct uri *uri, eel_group_id gid) {
3    char req[1024];
4    // create the HTTP request and write it to uri->fd
5    snprintf(req, sizeof(req), "%s %s HTTP/1.0\r\n" ...);
6    atomicio(write, uri->fd, req, strlen(req));
7    // wait for a read event on uri->fd or timeout
8    eel_add_read_timeout(gid, readheadercb,
         timeoutcb, uri, uri->fd, HTTP_READTIMEOUT);
9  }
10 // the timeout occurred before uri->fd was ready to read
11 void timeoutcb(eel_group_id gid, void *arg, int fd) {
12   // clean up all resources; ends the callback chain
13   uri_free_gid((struct uri *)arg, gid);
14 }
15 // uri->fd is ready to read
16 void readheadercb(eel_group_id gid, void *arg, int fd) {
17   char line[2048];
18   struct uri *uri = arg;
19   // read some data from uri->fd
20   ssize_t n = read(uri->fd, line, sizeof(line));
21   if (n == -1) {
22     if (errno == EINTR || errno == EAGAIN)
23       goto readmore; // wait for another read event
24     uri_free_gid(uri, gid); // real error: free and return
25     return;
26   } else if (n == 0) // ... handle other conditions
27   // ... copy unparsed header info into uri structure
28   http_parseheader(uri, gid);
29   return; // What callback is next???
30 readmore:
31   // wait for another read event or timeout
32   eel_add_read_timeout(gid, readheadercb,
         timeoutcb, uri, uri->fd, HTTP_READTIMEOUT);
33 }
```

**Figure 1**: Code from a version of crawl-0.4 [15] ported to *libeel*, showing part of a typical HTTP document fetch.

mal, non-blocking write. While this particular write is extremely unlikely to block in practice, true non-blocking I/O would require that any unused portion of `req` be passed on to the next callback.

Stack ripping complicates writing as well as reading. Consider a programmer writing Figure 1's code in top-down order. Once she finishes writing `readheadercb`, she might write `http_parseheader`. Unfortunately, this involves cleaning up some subset of `readheadercb`'s state; and whenever `readheadercb`'s state changes, `http_parseheader` must change too.

Say the programmer now wishes to debug by stepping line by line through the source code, observing variable values. She runs the program in a debugger and sets a breakpoint at line 6 to begin the process. After stepping a few lines to the end of `http_fetch`, the debugger steps to the calling function—but this is the dispatch loop. There is no convenient way to continue stepping on to the next line of logical program flow (11 or 16). Debuggers don't follow the logical control flow of event-driven programs, making stepping inconvenient.

In practice, programmers have avoided these problems primarily by turning to threads, whose explicit control flow improves programmability. Memory is more easily managed because stack variables can be used across blocking calls. Other resources are more easily managed because control paths that exit the function are more visible. Debugging is easier (assuming the debugger has thread support). Programmers that choose to use events, often for performance reasons, suffer through with ad-hoc solutions. For instance, separate documentation might be manually created to show the callback chain; memory and resource management is most likely done manually; `printf` debugging rules the day. Some systems combine events' cooperatively-scheduled execution model with thread-like code via automatic stack management [6, 19]; but this may not support multiple outstanding callbacks on the same connection, and still requires the programmer to revalidate shared state after each blocking call [6].

## 3 THE *eel* TOOLS

Our *eel* tools and a library framework attack all these problems at their common source: the difficulty of following an event-driven program's control flow. The *libeel* library simultaneously facilitates event-driven programming and program analysis: we designed the library specifically to avoid the aliasing and state issues that typically complicate analysis of C-based programs. Nevertheless, *libeel* programs are truly event-driven, not event-based programs in threaded clothing.

The tools leverage *libeel* to extract control-flow information from arbitrary event-driven programs. The results are displayed or used to verify program properties. *eelstatechart* visualizes the program's control flow in the form of a simple chart. The *eelverify* framework can detect resource leaks and other mistakes common to event-driven programs. Lastly, a modified *gdb* lets the programmer transparently step through the callback chain, simplifying debugging. Each tool plays a role in the programming process: *eelstatechart* in program comprehension, *eelverify* in checking, and *eelgdb* in debugging.

The *libeel* library was initially based on *libevent* [16], another event library, although it has considerably diverged. The *eel* tools were built using the C Intermediate Language (CIL) framework for C program manipulation and analysis [2], the BLAST software verification system [1], *gdb* [4], and Graphviz's *dot* [3].

### 3.1 The *libeel* interface

The *libeel* library, like other existing event libraries [8, 16], provides a single unified interface for registering, canceling, and dispatching callbacks. It abstracts system dependencies, such as the choice of *select* or a more-scalable variant [7, 13]. Figure 2 shows part of its interface. The event functions register a callback for an I/O event on the given file descriptor, or for a timer that goes off after a certain number of milliseconds. Other functions combine I/O with timeout events. The design challenge was to provide a usable, minimal interface that simultaneously enables analysis.

```
// Group operations
eel_group_id eel_new_group_id(void);
void eel_delete_group_id(eel_group_id gid);
// Event functions
eel_event_id eel_add_timer(eel_group_id gid, eel_callback cb, void *cb_arg, int timeout_milliseconds);
eel_event_id eel_add_read(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
eel_event_id eel_add_write(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
eel_event_id eel_add_error(eel_group_id gid, eel_callback cb, void *cb_arg, int fd);
```

**Figure 2**: Some of the *libeel* interface, including showing group identifier new and delete calls and event registration functions.

*libeel*'s interface is simpler than some other event notification libraries in that the callback functions are explicitly named for each event registration, and there is a one to one pairing of registrations and callback calls. *libeel* also requires the programmer to specify the logical connection to which an event applies, via group identifier arguments in all event registration calls. Explicit functions create and destroy group identifiers. It is typically easy to add group identifiers to an event-driven program: context data and resources passed along the call chain are usually allocated in a single location and deallocated in another; the group identifier can be created and released at these sites as well. Group identifiers somewhat resemble thread identifiers, but differ in that there can be multiple callbacks outstanding for the same group. The other *eel* tools trace group identifier values through the program's callbacks to extract logical code paths.

Initially, we considered using *libevent* directly, but doing so proved difficult. Event registration in *libevent* requires two library calls, one to set up a parameter data structure and one to actually register:

```
event_set(&ev, fd, EV_READ|EV_WRITE|EV_PERSIST,
          callback, NULL);
... // analysis must check whether ev has changed
event_add(&ev, &timeout);
```

This allows persistent (automatically recurring) registrations and multiple event types registered to the same callback function, and encourages persistent `ev` structures. For example, one `ev` might be initialized at the beginning of the program, then reused liberally throughout. Thus, whole-program alias analysis might be necessary to determine the callback function registered by a particular `event_add`, complicating both control flow analysis and human understanding.

*libeel* avoids these issues by requiring that all parameters be presented as explicit arguments, and by disallowing recurring registrations. The resulting one-to-one correspondence between a single event registration and a single callback firing decouples the semantic cases. These interface design differences keep the *libeel* semantics simple enough for program analysis and as flexible as *libevent* (although the latter is less verbose and marginally more efficient). Porting a *libevent* program to *libeel* is straightforward: separate out the multiple event types and persistent event registrations into independent callback functions and registrations. However, in cases where registration parameters are set distant from actual registrations (typically because the parameter structure is reused throughout the program), one must do whole program reasoning to determine what events are being registered to what callback function.

### 3.2 *eelstatechart*: visualizing the callback chain

*eelstatechart* helps *libeel* programmers better understand a program's control flow. Short of modifying C syntax in a non-trivial way, asynchronous execution can best be visualized using a graph. The chart we generate here is equivalent to the graph described by Lauer and Needham in 1978 [12] and the blocking graph described by von Behren et al. [18] Nodes in an *eelstatechart* are labeled with callback function names and edges with abbreviations for I/O or timer events. The purpose is to make the program's underlying structure more obvious, helping the programmer understand the common paths and how connections progress. Callbacks obscure even simple programs by removing context; *eelstatechart* recovers each callback's context in the program.

*eelstatechart* performs a static analysis; the tree of event registrations and their associated callbacks is determined while following the creation, use and release of group identifiers through the static callgraph of the program. *eelstatechart* starts by visiting all function definitions and their static function calls to build a call graph. When a *libeel* call is encountered it marks the calling function with a label indicating the operation performed. The source of the group identifier is located and added to the label as well. Finally, these labels are percolated all the way up the callgraph. To export the chart, the labels are traversed from callback to callback beginning with the program entry point.

Figure 3 shows the primary *eelstatechart* for crawl-0.4 [15], a simple Web crawler. The code from Figure 1 appears on the right side of the figure. `http_fetch` is called by `http_connectioncb`, creating the read readiness event and timeout event seen heading down and right from `http_connectioncb`. Once in `readheadercb`, the chart shows arrows indicating the callback registrations from line 31. It also shows an arrow to "delete", indicat-

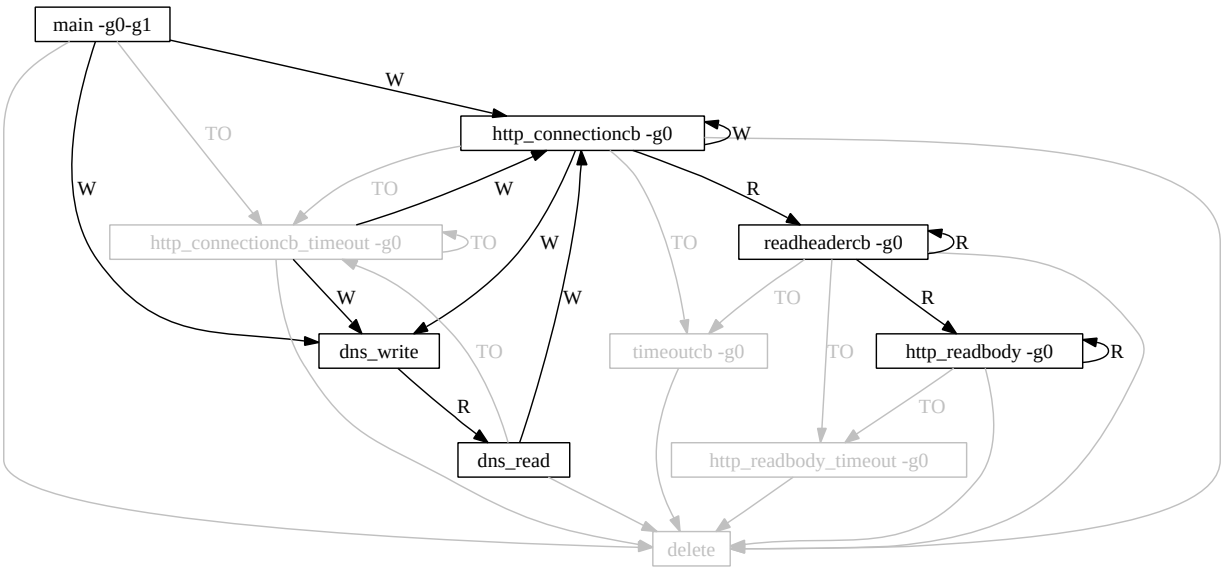


**Figure 3**: The primary *eelstatechart* for crawl-0.4 [15]. Each rectangle names a callback function. Each arrow indicates the next callback in the chain. Arrows are labeled with abbreviations of the event causing the callback to be fired: "W" is write, for example. Arrows pointing to "delete" indicate the end of the callback chain. Gray rectangles and arrows indicate timeout or delete paths, which typically correspond to errors.

ing that the callback chain can end, in this case from a call to `uri_free_gid` on line 24 or elsewhere. The call to `http_parseheader` on line 28 extends the callback chain to `http_readbody` or `http_readbody_timeout`, which go on to repeat back to `http_readbody` or end the chain. By just reading the code it is not apparent what callbacks might be generated inside the call to `http_parseheader`; *eelstatechart* clearly conveys this information.

*eelstatechart* will generate an approximation of the true chart, rather than the true chart, if an event registration uses a variable to name a callback function (rather than a naming a callback function directly), or due to complex use of function pointers elsewhere in the code. This hasn't happened in the programs we've converted so far. One remaining challenge is to create a chart that is easily read but also contains all pertinent information. For example, it would be especially nice to show what lines generated which events. We collect enough detail to provide this information, but it would clutter the chart beyond easy readability. Another challenge is visualizing cases where more than one next callback is registered, i.e. the control proceeds down both callback chains in an unspecified order—a particularly flexible pattern.

### 3.3 *eelverify*: a verification framework

*eelverify* is a framework for verifying properties of *libeel* programs. It provides a set of program transformations and instrumentation points for *libeel* programs, as well as verifiers that use these transformations. For instance, *eelverify* can verify that group identifiers are not leaked anywhere along the callback chain. It first performs a simple program transformation so that callback functions can be verified independent of each other. Then BLAST [10] is used to instrument the `eel_group_id` type, *libeel* calls, and callback function returns such that if a group identifier is leaked, an error label is reached. Other properties can be verified using a similar approach.

Using *eelverify* we found a few actual bugs (and a few false positives) from two programs that, together, had about 15,000 lines of uncommented C code. One interesting bug stands out in plb-0.3 [5], an HTTP load balancer. The offending code segment is in a callback function, `client_forward_request`, executed following a read readiness event. It then attempts to execute the read call. On an error read result it checks for `EINTR`, which indicates that a signal interrupted the read attempt. Typically this situation is handled by waiting again for a read readiness event, but the callback simply returns without registering any callback or releasing resources. Here, `EINTR` would result in a failure to forward HTTP POST data from the client to the server. *eelverify* found this bug because the group identifier passed into the callback function was not used or released along the call path. It's worth noting that this bug might be hard for an automatic checker to detect [9]. Since different callbacks were set on different paths, some exit points deleted the group identifier, while others did not.

*eelverify* implicitly assumes that *libeel* is correct; it uses the *libeel* semantics but acts on its functions as if they were language keywords. *libeel* cannot be verified directly because it uses function pointers and com-

plex data structures to manage callback dispatch. Under this assumption its analysis is sound, however, meaning *eelverify* never will report a false negative. Function pointer usage inside callback functions can lead to false positives, however.

*eelverify* provides a framework for verifying a broader class of resource properties, including those that follow a paired calling pattern such as create/release, alloc/free, or open/close, within the context of a *libeel* event-driven program. For example, it might ensure that file descriptors are always closed after being opened, or that they are not used after being closed. However, *eelverify* can currently verify properties only along a single instance of the callback chain; it ignores any dependencies between instances or between separate chains.

### 3.4 Debugging with *eel*

*eelgdb*'s extensions consist of a few new commands that allow stepping line by line through a *libeel* callback chain. 'Cnext' is similar to the *gdb* 'next' command, except that if the current line matches a pattern indicating the addition of a *libeel* event, it will create a new temporary conditional breakpoint at that callback function's header. These breakpoints will only stop the program if the group identifier argument equals that of the currently active callback. Thus, program execution can continue until the next breakpoint in the logical connection, allowing for transparent stepping to the next logical point in the program. The result is that the debugger allows callbacks for other connections to be dispatched while it is waiting for the next relevant event, but returns control to the user once an event for the current connection has triggered. The analogous situation in the threaded model is that when an I/O call blocks, the debugger executes code on other threads while it waits for the I/O call to complete.

For example, consider debugging the code:

```
1    ...
2    atomicio(write, uri->fd, req, strlen(req));
3    eel_add_read(gid1, readheadercb, uri, uri->fd, 1000);
4  }
5  void readheadercb(eel_group_id gid2, void *arg, int fd) {
6    ...
```

Assume the program is run in *eelgdb*, which hits a breakpoint on line 2. The user executes 'cnext', which causes the debugger to step to line 3, just as 'next' would. When 'cnext' is applied to line 3, a *libeel* pattern matches the line of source code, extracting the expression `gid1` and the identifier `readheadercb`. (As with the other *eel* tools, *eelgdb* does not currently handle function pointer usage in event registrations.) Next it evaluates `gid1`'s value at line 3 (e.g. `0x007A224F`) and sets a conditional breakpoint as follows: **`tbreak`** `readheadercb` **`when`** `(gid2 == 0x007A224F)`. Then it steps over line 3 to line 4. The user can then 'continue' to allow the program to proceed or step back to the calling function. Once the program is continued, if the read event is triggered on the same group identifier, the *libeel* dispatch loop will call `readheadercb` and hit the breakpoint on line 5. The user then proceeds debugging the same instance.

## 4 Related Work

In 1978, Lauer and Needham proved that threads and events are duals [12]. Most still researches believe that one or the other is better, however. Ousterhout argued that threads are a bad idea because they perform poorly, and concurrency issues make them error-prone [14]. Von Behren et al. argue, in contrast, that event-based programs are too difficult to write, for the reasons we have explained [18]. They aimed to improve the performance of threads to match that of events; Capriccio's compiler analyses and runtime techniques change a threaded program's runtime behavior into that of a cooperatively-scheduled event-driven program [19]. Even here, events and threads are dual: events need no compiler help for performance, since they perform well already; instead, we use analyses and *static* techniques to improve the programmability of events to match that of threads (or, arguably, better that of preemptively-scheduled threads, because there are no concurrency issues). Adya et al. named "stack ripping", identified it as a major issue with event-driven programming, and introduced a mechanism for automatically managing multiple stacks [6]. The *libeel* library leaves the user to manage the stack manually, and the existing *eel* tools address the problems that result. *Eel*-like tools for a system with automatic stack management would address its problems instead—for instance, by checking that any stack copies of global state are revalidated after each blocking call.

Several projects focus on building fast web servers, or fair web servers, using events [11, 17] or a combination of events and threads, as in SEDA [20]. Dabek et al. describe a C++ library, *libasync*, for building robust event-driven software [8]. *libasync* primarily addresses callback safety by using C++ templates to cross-check callback function types and context data. It also adds reference-counted objects to ameliorate some resource management issues. We focused on enabling and building static tools that check safety issues and facilitate program clarity; reference counting and type checking would be complementary.

## 5 Conclusion

The *eel* library and program analysis tools help programmers evade common problems with the event-driven model, while remaining inside that model. We are working on further improvements to visualization to differentiate success and error execution paths, and on verifying

other properties such as proper file descriptor usage. We are also working on a program transformation, in conjunction with modifications to BLAST, that would allow verification using regular BLAST specifications, instead of those phrased to verify properties of callback functions independent of each other. This would let us verify properties that require simultaneous analysis of more than one callback. As well, collecting profiling information tagged with group identifiers could aid in debugging resource bottlenecks in *libeel* programs. Even now, however, the *eel* tools make it easier to read, write, debug and maintain event-driven programs. Code will be available at `http://read.cs.ucla.edu/`.

## ACKNOWLEDGEMENTS

We gratefully acknowledge Rupak Majumdar for discussions and BLAST aid, and the anonymous reviewers for helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 0427202.

## REFERENCES

[1] BLAST: Berkeley Lazy Abstraction Software Verification Tool. URL `http://www-cad.eecs.berkeley.edu/~rupak/blast/`.

[2] CIL—Infrastructure for C program analysis and transformation. URL `http://manju.cs.berkeley.edu/cil/`.

[3] Graphviz—graph visualization software. URL `http://graphviz.org/`.

[4] GDB: The GNU Project Debugger. URL `http://www.gnu.org/software/gdb/gdb.html`.

[5] PLB—Pure Load Balancer: A free high-performance load balancer for Unix. URL `http://plb.sunsite.dk/`.

[6] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, June 2002.

[7] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proc. 1999 USENIX Annual Technical Conference*, pages 253–265, Monterey, California, June 1999.

[8] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop*, September 2002.

[9] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Château Lake Louise, Alberta, Canada, Oct. 2001.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

[11] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2004.

[12] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems*, pages 408–423. INRIA, October 1978.

[13] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track (USENIX-01)*, June 2001.

[14] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[15] N. Provos. crawl—a small and efficient HTTP crawler. URL `http://www.monkey.org/~provos/crawl/`.

[16] N. Provos. libevent—an event notification library. URL `http://www.monkey.org/~provos/libevent/`.

[17] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proc. 2004 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2004.

[18] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. HotOS-IX: The 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[19] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, Lake George, New York, Oct. 2003.

[20] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Château Lake Louise, Alberta, Canada, Oct. 2001.