BorderPatrol: Isolating Events for Black-box Tracing

Eric Koskinen
Department of Computer Science
Brown University
115 Waterman St., Providence,
Rhode Island 02912, USA
ejk@cs.brown.edu

John Jannotti
Department of Computer Science
Brown University
115 Waterman St., Providence,
Rhode Island 02912, USA
jj@cs.brown.edu

ABSTRACT

Causal request traces are valuable to developers of large concurrent and distributed applications, yet difficult to obtain. Traces show how a request is processed, and can be analyzed by tools to detect performance or correctness errors and anomalous behavior.

We present BorderPatrol, which obtains precise request traces through systems built from a litany of unmodified modules. Traced components include Apache, thttpd, PostgreSQL, TurboGears, BIND and notably Zeus, a closed-source event-driven web server. BorderPatrol obtains traces using *active observation* which carefully modifies the event stream observed by modules, simplifying precise observation. *Protocol processors* leverage knowledge about standard protocols, avoiding application-specific instrumentation.

BorderPatrol obtains precise traces for black-box systems that cannot be traced by any other technique. We confirm the accuracy of BorderPatrol's traces by comparing to manual instrumentation, and compare the developer effort required for each kind of trace. BorderPatrol imposes limited overhead on real systems (approximately 10-15%) and it may be enabled or disabled in at run-time, making it a viable option for deployment in production environments.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; D.2.5 [**Software Engineering**]: Testing and Debugging–Distributed debugging, testing tools, tracing; K.6.1 [**Project and People Management**]: Systems analysis and design; K.6.4 [**System Management**]

General Terms

Performance, Measurement

Keywords

Performance debugging, black box systems, distributed systems, performance analysis, causal paths

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

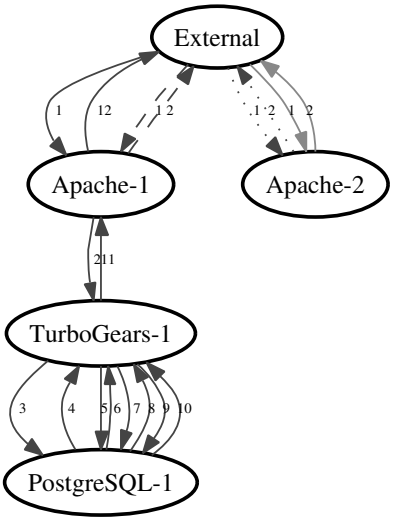

Figure 1: A BorderPatrol trace of a three-tier web application. Four external requests (each shown with a different arrow type) are handled by Apache in two processes. The first request, shown in solid black, is passed on to a web-application written in Python which makes several database calls to fulfill the request. The programs were not modified to obtain the trace.

1. INTRODUCTION

Today's large-scale applications consist of many independent modules that leverage concurrency for performance. In many cases, the components are developed by different groups and in different languages. Individual components may exploit concurrency with threaded, multi-process, or event-driven designs.

Regardless of this heterogeneity, developers want answers to questions about their entire applications. "What path through the system do search requests take, and where do they spend the most time?" or "What resources are used by clients reading email, as compared to sending email?" The principals of interest in these queries are requests, not individual modules. Developers would benefit from tracing tools that follow single requests as they are passed between modules, including third-party binary modules, even as those requests are passed and returned from remote, unmodified systems. Figure 1 shows a trace produced by BorderPatrol, our tool for tracing unmodified systems of black-boxes.

Beyond being used as an aid to developers, recent work has shown that request traces can be valuable input to au-

tomated tools. Systems such as Pinpoint [7], Pip [13], and Stardust [19] analyze precise request traces to identify faulty modules, discover anomalous request paths, and make capacity plans. However, these systems rely on simple trace gathering techniques that are unable to trace complex systems.

Obtaining precise request traces in a heterogeneous, concurrent system is difficult. It is insufficient to obtain traditional trace data, such as system call or function call logs, since these logs do not indicate when high-level requests have been handed off from module to module. Instead, most tracing systems have advocated module-specific programmer-supplied instrumentation which requires source-level access and an understanding of the module architecture. While some[1] have avoided instrumentation, they have sacrificed precise traces for statistical information.

When an application spans multiple modules, or when a module multiplexes several requests, request flow does not follow module control flow. Instead requests are executed in fragments by modules that multiplex their time among many such requests. These modules may use operating system abstractions such as processes or threads, or they may manage concurrency themselves, using an event loop or user-level threading package.

BorderPatrol follows requests as they move through this cacophony of modules, written by disparate teams, loosely aggregated with protocols that do not share a unifying request abstraction. We understand, before we begin, that perfection is impossible. In the general case, precise black-box tracing is impossible, since modules may act in arbitrary ways inside their "black-boxes" particularly when presented with simultaneous requests. However, our observation is that real applications are not arbitrary. Through careful observation and a light-weight form of module isolation, causal paths can be reconstructed in real-world systems.

Previous tracing systems have also recognized the impossibility of precise tracing without programmer assistance. Some have required programmer assistance in the form of a "bread crumb" trail emitted by instrumentation, or by stashing request identifiers inside of datastructures that are carried throughout a framework. Neither solution is satisfying in a heterogeneous environment. Others have sacrificed precision in favor of statistical results. In these systems, common paths through unmodified modules can be found with some probability, but precise traces of specific anomalous paths cannot be determined. We discuss all of these approaches in Section 8.

Our compromise, and therefore our contribution, is different. We present a tracing technique that *actively isolates black-box inputs* so that request paths can be precisely observed, without materially affecting the overall application's ability to multiplex requests. *Event isolation* (Section 3.3) unbundles concurrent input events in order to allow the observation of a module's behavior on a per-event basis. When event isolation is impossible or undesirable, we identify request propagation by inspection. *Message witnesses* (Section 3.2) identify matched messages, usually request/response pairs. Event isolation and message witnesses are provided by *protocol processors* (Section 3.1), an abstraction that allows developers to implement protocol-specific, rather than implementation-specific, tracing. A single HTTP protocol processor can be used to trace various web servers, web proxies, or even XML-HTTP services.

The techniques described in this paper are realized in a tool called BorderPatrol, which is publicly available. BorderPatrol uses library interposition to insert protocol processors between the unmodified black-boxes of a multi-module system. For BorderPatrol to be effective, the black-boxes must follow certain assumptions that we outline in the following section. We argue that these assumptions are reasonable because they follow from common software architectures.

Our evaluation consists of case studies (Section 6) and a performance evaluation (Section 7). We show that BorderPatrol reconstructs causal paths through a range of diverse servers including Apache, thttpd, Zeus, BIND, PostgreSQL, and TurboGears, without modifications to server source code or the use of statistical methods. Further, we show that the overhead of tracing is about 10-15% and can be activated at runtime, making it a viable technique for production environments.

2. BLACK BOX MODEL

BorderPatrol seeks to follow the repeated transfer of a request from one black-box module to another in order to construct causal paths that show which modules handled a given request, in what order, and for how long. For example, when a web application queries a database, we want to associate the computation in the database with the original HTTP request. Although BorderPatrol treats modules as "black boxes," it makes some assumptions about the way real-world applications work that allow it to follow request transfers.

Request traces can be thought of as chains that are made up of two types of links. *External* links connect the output of one black-box module to the input of another. *Internal* links connect a module input to a module output.

External links can be observed by monitoring communication channels using any number of techniques, *i.e.* network snooping, virtual machine monitoring, and system or library call interposition.

The internal links of black boxes cannot be observed directly. Instead, BorderPatrol makes inferences based on the assumption that the internals of black boxes are *honest*, *immediate*, and *independent*. A black box is honest if it faithfully implements the basic structure of the protocols it participates in. It is immediate if, when presented with a single input event, it processes the input event before requesting another input. Finally, black boxes are independent if they process concurrent input events in the same way that they would have processed the events if they arrived sequentially, except for timing effects. The remainder of this section details these assumptions, and describes why we expect that the operation of real-world black boxes operate within them.

2.1 Honesty

Sometimes, internal links can be established by observing the contents of input and output messages. This is common when a request is passed out of a module using the same protocol that passed the request in, so that an identifier is visible in both messages, for example in the nonce of an RPC request/reply, or the URL in a proxy server request that is forwarded to the origin server. We refer to these identifiers as *witnesses* and BorderPatrol assumes they are accurate if they exist. We expect that bugs affecting witnesses are so basic that they are unlikely in systems mature enough to

consider tracing. Witnesses are used only to patch paths when BorderPatrol's request following techniques cannot be used, such as when building a path through a remote, untraced module.

In effect, a "dishonest" module is violating its specification so badly that it's hard to define what BorderPatrol *should* do. If a module changes the identifiers associated with a request in an unrecognizable violation of the protocol, is it really forwarding the original request? BorderPatrol does *not* trace black-box systems, it traces systems made up of black-boxes. The definition of a black-box is an unknown module that adheres to a specification, and honesty requires that adherence.

2.2 Immediacy

Usually, determining internal links is more difficult than matching witnesses. An internal link matches a module's input messages to its output messages. However, the protocols used may be unrelated and the follow-on messages may carry no identifying information that can be tied to the original request. For example, a web page request arrives as an HTTP request, but the application server may issue SQL requests to the database server that bear no resemblance to the original HTTP request. No witness oriented approach could produce internal links for such modules.

In addition, modules multiplex requests. Tracing cannot be accomplished simply by maintaining a map between requests and the module (or even the thread within the module) which will be handling it, then associating any further communication with that request. For example, event-driven systems rotate among outstanding requests using a single thread. Additionally, a single process may collect multiple inputs (via `read` for example), and then work on them consecutively with no externally identifiable break between them.

Our model assumes that black boxes are immediate–they are composed of pieces we refer to as *fragments* that do *not* multiplex requests. A fragment is an internal control path that handles individual inputs and processes them to completion. These fragments usually do not process an entire request. The execution of the fragment runs from one input event (such as data becoming available on a socket) to another, not from request start to finish.

Since fragments immediately begin work on the request associated with their input event, BorderPatrol can determine internal links by supplying that input, noting the output caused by the fragment, and connecting the two. BorderPatrol takes a general view of *output* that includes any interaction with an outside module, such as connection creation. We describe this process in detail in Section 5.5.

Immediacy is illustrated in Figure 2. On the left, two concurrent requests enter a black-box module, and since nothing is known about the module internals, it is impossible to match the inputs to the outputs. However, the right side of the diagram illustrates the module's true structure. Although BorderPatrol cannot determine this structure, it is easy to see that if the events are supplied independently, the output can be matched with the input.

2.3 Independence

Our final assumption about black-box modules is independence. We assume that modules will respond to two sequential inputs the same way they would have treated those

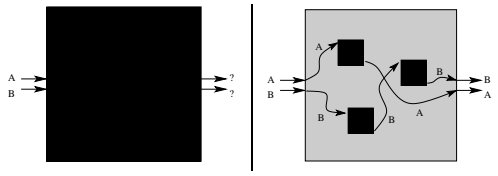

Figure 2: An illustration of the *immediacy* and *independence* assumptions. Immediacy tells us that when A is supplied, the black box's next observable action will be to create the output labeled A. Independence tells us that the black box would not have treated A differently had it been supplied simultaneously with B.

inputs if they had been received simultaneously. In a concrete example, we assume that an application that uses `poll` will not behave differently if it must call `poll` twice to obtain two ready file descriptors.

Like immediacy, the independence assumption says nothing of entire requests, only the individual *events* that comprise them. We believe that inside real-world modules, multiple input events are immediately separated by the structure of well-written software. Libraries such as `libevent` [10] and `libasync` [11], dispatch to independent event handlers for each event. If multiple events are supplied to an event-driven module, the internal event loop dispatches these events serially to event handlers. In threaded applications, independence is even clearer. Each thread in these applications blocks waiting for a single next event (such as the completion of a read) to proceed. There is no danger that the behavior of these applications will change when event arrival is serialized.

Although batch-oriented interfaces are a common performance optimization, we explain in Section 4 how BorderPatrol is able to take advantage of independence without foiling these optimizations. Briefly, the batch interface is still used across the slow kernel-user boundary, but individual events are supplied from BorderPatrol's interposition layer to the original module.

Independence does not imply that modules will function internally in an identical fashion when events are reordered. For example, a server that uses caching techniques will respond with an alternate code path when the cache is populated by an earlier request. BorderPatrol will faithfully follow the new path. The independence assumption says that the alternative path is *correct*, not identical to what would have occurred without BorderPatrol. This change means that BorderPatrol is not the optimal debugging tool for finding errors due to race conditions or locking mistakes that require fine-grained replay of identical traces. However, correct implementations of black-box specifications will exhibit independence, and remain correct while running under BorderPatrol.

3. ACTIVE OBSERVATION

BorderPatrol employs *active observation* to observe and subtly modify the event streams sent and received by monitored modules. Active observation, allows BorderPatrol to precisely trace modules that follow the assumptions of the previous section without materially affecting module behavior.

Protocol processors implement active observation in a modular way, encoding protocol knowledge, not implementation

knowledge. Protocol processors are used to understand and separate multiple messages on a single channel. These processors also record witnesses that allow path reconstruction when external modules are used that are not traced by BorderPatrol, and protocol specify attributes that may be interesting to distinguish requests for user queries.

3.1 Protocol Processors

BorderPatrol uses library interposition to pass input to protocol processors before passing it on to unmodified modules. The protocol processors identify message boundaries, log protocol-specific attributes that users may wish to query, and track message witnesses (see below). Although the development of protocol processors requires more specialized knowledge than pure black-box approaches, the knowledge is not application-specific, but protocol-specific. We have used the same HTTP protocol processor to trace many different web servers with wildly varying implementations. Furthermore, these protocol processors do not fully implement the protocol, they usually understand little beyond the basic "envelope" of the protocol messages.

Protocol processors operate by looking for message delimiters or length counts in the data stream, and understand messages only enough to log application-specific identifiers such as URLs, SQL queries, or sequence numbers. The interface from the interposition library to the protocol processor has been designed to make these tasks easy. As a result, the protocol processors we have implemented are between 30-150 lines of code. We discuss the details of the protocol processor interface and implementation in Section 5.2.

3.2 Message Witnesses

Message witnesses are the simplest way to establish the internal links from module inputs to outputs. A *message witness* is data that can be extracted from input messages and output messages to allow direct linking. Unfortunately, witnesses are unlikely when input and output messages are of different protocols, so they are useful mainly for linking requests and replies, particularly for requests into modules that cannot be traced by BorderPatrol, such as remote web services.

Protocol processors find and log message witnesses such as URLs and RPC identifiers. When these witnesses reappear in responses, BorderPatrol can stitch together the request trace that might have otherwise been lost when the request was being handled by a remote module.

3.3 Event Isolation

In order to directly follow internal links without witnesses, BorderPatrol supplies input events to modules one at a time. BorderPatrol then monitors the module's output, and assumes (due to immediacy) that any activity can be attributed to the same request as the input event. As a concrete example consider `poll`, an interface that modules use to obtain events for any number of file descriptors. At the time `poll` is called, BorderPatrol has tracked the input that is available on each channel, and can attribute each potential input to a high-level request. By returning only one event at a time to the black-box module, BorderPatrol can attribute the following fragment of computation and outputs to the request associated with the event.

The events returned by `poll` are indivisible, they can be attributed to only one request. However, when a module reads input data, there is the danger that input from several messages, and therefore several requests, is combined. Protocol processors allow BorderPatrol to isolate events at the protocol level, preventing reads containing multiple messages.

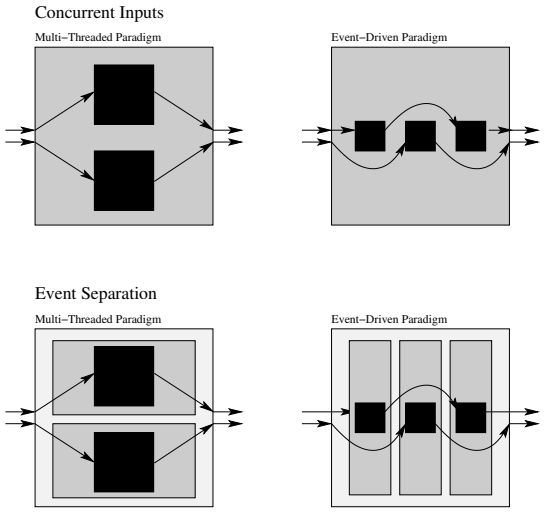

Figure 3: BorderPatrol works because real-world servers have straightforward internal structure. Multi-threaded servers dispatch events independently, to separate threads (left). Event-driven server execute in fragments that can be pieced together by running them sequentially (right).

When data arrives on a channel with a protocol processor, the data may contain multiple input events that should be isolated. The protocol processor examines the data in search of message boundaries. If a boundary is found, only the data leading up to the boundary is returned. On the next read, the processor considers the cached data first, though it may find yet another boundary, and again return a "short read."

4. WHY DOES BORDERPATROL WORK?

Do real-world applications follow our assumptions and decompose cleanly to code fragments that operate on individual events? Can BorderPatrol obtain that decomposition? This section explores typical application architectures and explains when and why applications can be decomposed and traced accurately.

Fundamental to real-world interactive programs, of which servers are a subset, is the ability to handle concurrent requests. Therefore, these applications must be able to accept new requests continuously, even as previous requests are still being processed. There are several common paradigms for multiplexing requests. Using the taxonomy presented by Pai *et al.* [12], we consider some of the most popular.

Multi-process or Multi-threaded. Servers written in the MP/MT style maintain a pool of individual threads (or processes). These threads loop, continuously accepting new requests, processing each one to completion. In pseudo-code:

```
while (fd = accept())
  while (req = read(fd))
    handle_request(req);
  close(fd);
```

Such a server is immediate and independent. While inside `handle_request`, the server will service only a single

request. It may interact with additional modules to aid in servicing the request, but BorderPatrol's tracing job is easy. For example, the request might be an HTTP request for a page containing user customized data obtained via an RPC interface. BorderPatrol attributes the RPC to the top-level request, and continues path reconstruction in the destination module. If the destination module is not running BorderPatrol, a witness in the RPC response can reestablish the request path, treating the entire remote module as a single black box. BorderPatrol does *not* simply assume that sequential behavior across the RPC call, in two separate fragments, is related. BorderPatrol follows the request back into the web server from the remote module. The link is established by active observation, not by assumptions about single-threaded code.

Single Process Event-driven. SPED servers multiplex requests across a single thread. In pseudo-code:

```
while(1)
  events = poll();
  for e in events
    handler = find_handler(e);
    execute(handler, e);
```

The handling of a single request is divided into many smaller stages. The equivalent of `handle_request()` might consist of several handlers: (1) parse the request and initiate a connection to the RPC server (2) complete the connection to the RPC server (3) write a message to the RPC server (4) read the response from the RPC server and (5) compute the HTML response and write it to the client. Further, each of these stages might re-register the same handler to complete a lengthy operation.

BorderPatrol ensures that the SPED process receives only one event at a time, so all of the following actions, until the next input, can be attributed to the input event's request. An illustration of this architecture appears on the right-hand side of Figure 3.

Asymmetric Multi-Process Event-Driven AMPED servers are similar to SPED servers, with the addition of helper processes used to simulate asynchronous I/O. BorderPatrol observes the requests from the main process and attributes the work of the helper to the high-level request that initiated contact with the helper. BorderPatrol will require a protocol processor in the case that the communication between the main process and helper persists on a single channel. Simpler interactions with subprocesses that span a single request can be handled by BorderPatrol's default "One-shot" protocol processor. We expect that these ad-hoc protocols are conveniently delimited or use fixed frame sizes. We expect implementations in the 10s of lines.

Work Queues. Applications that make use of "hidden" work queues to pass requests from module to module will present a problem for BorderPatrol's tracing because of unobservable fragment interactions. Work queues may be implemented with internal data structures that cannot be observed without more invasive techniques. However, some work queue implementations do have standardized interfaces, and if they are implemented as shared libraries or via IPC, fragment interactions might be observed by an "API processor" akin to BorderPatrol's protocol processors. Regardless we were somewhat surprised, but pleased, not to find this model in the many modules we examined.

Whodunit [6] aims to derive information from (nearly) unmodified servers that pass requests in this manner. There is potential synergy with BorderPatrol's mechanism, though Whodunit's output is statistical making it inappropriate for use in precise tracing without further adaptation.

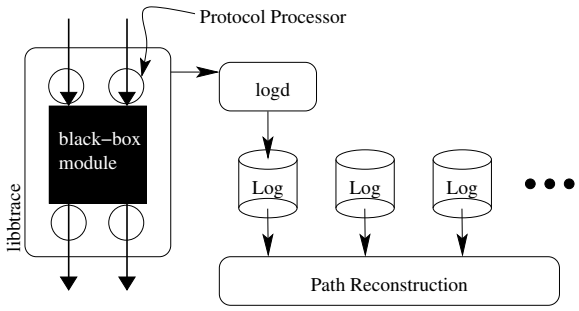

Figure 4: System Overview. The solid black box represents a traced application module. Communications (messages, IPC, and signals) are monitored by the protocol and/or API processors. Events are relayed through the Logging Daemon to a raw database. Databases from multiple hosts may be aggregated, and causal paths are reconstructed.

User-level Scheduling. User-level threads may also present difficulties for BorderPatrol, depending on implementation. Cooperative thread packages context switch only when the current thread attempts a blocking system call. A non-blocking version is substituted, and the thread context is switched. The thread may be resumed when an OS notification indicates the operation would not block. This architecture is identical to SPED for BorderPatrol's purposes and presents no difficulties. However, other thread packages use asynchronous signals in order to support preemption at regular intervals. BorderPatrol does not currently support the interception of these signals. Even if it did, these signals are periodic and indicate only that time has passed. Treating the package as a black box would prevent BorderPatrol from understanding the user-level scheduling code and knowing which thread has been activated. BorderPatrol could resume tracing when an interaction with a known resource is observed.

5. IMPLEMENTATION

BorderPatrol collects trace data from unmodified modules through the extensive use of library interposition which can be supplemented with data from a kernel module. The trace data is aggregated and processed in one forward pass to obtain request paths. An overview of the architecture is given in Figure 4.

5.1 Library Interposition (`libbtrace`)

BorderPatrol's interception framework is a series of wrapper functions for roughly 20 standard library functions. Our library of wrappers, `libbtrace`, is pre-loaded before system libraries using the `LD_PRELOAD` mechanism. `Libbtrace` intercepts calls to `libc`, isolates events, invokes protocol processors, and emits logging events. Often, an intercepted function invokes the real `libc` routine as part of its work.

`Libbtrace` tracks the requests associated with each connection in a process. BorderPatrol tracks all connection creation operations (`open`, `socket`, `pipe`, *etc.*) and modification operations (`close`, `dup`, `fcntl`).

`Libbtrace` also tracks messages as they flow through read and write operations. Many connections do not need to be

monitored since request causality doesn't flow across them. For example, the work involved in opening a file should be attributed to the current request, but the request typically doesn't flow into the file (although we believe BorderPatrol could be enhanced to track architectures such as mail servers in which requests pass through the file system). For simple file operations, BorderPatrol simply logs the interaction.

By contrast, requests *do* flow over many connections in a distributed system (*e.g.* FastCGI connections and database connections). In these cases, BorderPatrol (a) identifies the protocol involved (b) invokes the protocol processor on `read`/`write` operations on the connection, and (c) buffers data and events when event isolation requires it.

Currently, BorderPatrol does not trace some interfaces that it ought to in order to gain the most comprehensive coverage. For example, signals, the kevent API, and the AIO system calls are all ignored. These interfaces do not appear to pose fundamental challenges, they have been neglected thus far only because of their rarity.

5.2 Protocol Processors

`Libbtrace` also contains the protocol-specific code that implements protocol processors. These processors provide tracing for any application that implements the protocol, regardless of architecture. While we envision the possibility of dynamically selecting the appropriate protocol processor by observing passing data, BorderPatrol currently selects the appropriate processor based on conventions such as port number, Unix domain path, or executable name.

The protocol processor interface consists of four functions, two of which are used for initialization and tear-down. The following descriptions use pseudo-code data types to elide the details of C typing and buffer handling.

pp_state pp_init() Processors allocate and initialize a structure to store protocol specific state for a given communication channel in between invocations of the processor. The allocated state is passed as the first argument to all other functions.

void pp_shutdown(pp_state) When an application closes a communication channel, processors are called to allow the deallocation of resources obtained in `pp_init`.

int pp_read(pp_state, buffer) When data arrives on an input channel, `pp_read` is invoked to log and demarcate requests. The processor returns the number of bytes from the buffer that may be passed safely to the application without crossing a protocol message boundary. If the border between two requests is found in the buffer, the processor returns the offset of the boundary. If there is no message boundary, the entire buffer may be passed through to the application even if the buffer only represents a partial message. However, for the convenience of protocol processors, BorderPatrol can buffer partial messages in order to supply `pp_read` with the complete message when more data arrives. The processor indicates the desire for buffering by returning `PP_NEED_MORE`.

int pp_write(pp_state, buffer) When data is being written to an output channel, `pp_write` demarcates and logs, just as `pp_read`. However, when BorderPatrol writes data, there is no need to perform event isolation. The protocol processor is invoked only to log events and witnesses.

Two protocol processors must be written for most protocols. The read and write functions are used to process the messages for a protocol in one direction. The write function is invoked at the sender, and read at the receiver, but they perform nearly the same work, except for a difference in logging a receive or send. To process a protocol in both directions, a second protocol processor is used that understands the format of response messages.

```
int pp_http_read(pp_http_t state, buffer buf) {
  switch(state->s) {
    case DONE_1_0:
      return buf.length();

    case AWAIT_HEADER:
      i = find_re(buf, "GET.*?HTTP/1.1\r\n.*?\r\n\r\n")
      if (i==0) return PP_NEED_MORE;

      url = extract_url(buf)
      httpv = extract_version(buf)
      log(http_req, url, state->seq++)
      state->s = httpv == 1.1 ? AWAIT_HEADER : DONE_1_0;
      return i
  }
}
```

Figure 5: Example protocol processor for client to server communication using HTTP. `pp_http_read` illustrates an HTTP protocol processor for client to server communication. Due to the simplified interface, `pp_http_read` always operates from the start of the message.

Protocol Processor	Lines of Code
HTTP (1.0 & 1.1)	105
FastCGI	118
PostgreSQL	147
X11 (client-side only)	50
DNS (client-side only)	27
One-shot	28
Line-oriented	37

Figure 6: Protocol processor line counts. Each count includes *both* the client- and server-side of the protocol, except where noted. "One-shot" is used to handle any protocol with one request/response per connection. "Line-oriented" handles any protocol that uses newline to delimit sequential messages.

An example `pp_read` for HTTP is shown in Figure 5. HTTP is a simple, sequential protocol in which each request is separated by two pairs of line-feed/newline characters. This example is organized in a state transition style. The `DONE_1_0` state only applies to HTTP/1.0 clients. Once a request header is received, the protocol processor enters this final state since HTTP/1.0 forbids reusing a connection for multiple requests. In the alternate state `AWAIT_HEADER`, the processor looks for the request separator. If it isn't found in the current data, it returns `PP_NEED_MORE`, indicating that the processor should be invoked again when more data has arrived. While the partial request is cached for the benefit of the processor, BorderPatrol also passes it through to the application because there is no danger that the partial request contains a request boundary. Finally, when the complete request is recognized, attributes are parsed from the header and logged.

BorderPatrol's actual HTTP processor is somewhat more complex, but not much, weighing in at 105 lines total for all functions. Figure 6 shows line counts for several other protocol processors, each less than 150 lines long.

BorderPatrol tracks data flowing through each file descrip-

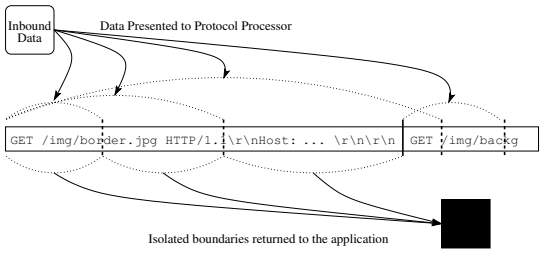

Figure 7: Data may arrive in chunks (dotted lines) on a multiplexed channel without regard to message boundaries. Protocol processors ensure that this data is supplied to applications in chunks that respect message boundaries (solid line), allowing BorderPatrol to track the behavior of applications from the point that a message has been provided, without confusion from further additional messages.

tor and maintains cursors to indicate which portions have been sent to the application, presented to the protocol processor, and not yet considered. Figure 7 demonstrates this tracking for HTTP.

If there is any data that the application has not yet collected, as much data as possible is passed to the application, considering the application's buffer size and the position of the protocol processor. Otherwise, there may be additional data, collected during a previous `read`, that the protocol processor has not yet seen. This happens whenever a protocol processor consumes a partial message or one of two contiguous messages. Finally, in the event that the protocol processor has been presented with all data in the buffer (even if it contains a portion of the next protocol chunk) and there is no cached data that can be passed to the application, the real version of `read` is used to refill the internal buffer.

Using the protocol processor on *out-bound* data is far simpler. Application `write`s are never shortened. Instead, the protocol processor is called repeatedly until all messages in the stream have been identified and logged. The remaining data is buffered until the next time `write` is invoked.

When BorderPatrol retains data in order to perform event isolation, it must also modify the result of any future call to `poll`. The buffered file descriptor is labeled readable regardless of its actual condition. In this way the application will call `read` again, which can be fulfilled from the buffer.

5.3 Kernel Page fault Monitor

User-level library interception is insufficient for capturing entry and exit from some kernel-related processes. BorderPatrol can install monitoring points in the Linux kernel in order to observe page-fault activity. Some processes use `mmap` to allow the operating system to page in data on demand without an explicit call to `read`. Library interposition cannot be used to observe I/O that results from page faults on `mmap`-ed pages, because page faults cause a transparent trap to the kernel. BorderPatrol includes a kernel process, `pftrace` that logs page faults in specified processes. `pftrace` uses `kprobes` to register call-backs whenever page faults occur. The process ID and time-stamps are passed through `relayfs` to a user-space daemon, which forwards them to the logging daemon.

5.4 Logging

Traces collected from the interposition library and the kernel page fault monitor are sent across a named pipe to the per-host logging daemon (`logd`). The logging daemon exists to collect events from traced processes, buffer them, and write them in batches to disk. Each thread maintains a separate connection to the log daemon, so events from different threads may be received out of order. However, events from any particular thread are ordered by the pipe.

The volume and frequency of events motivated a binary logging format to limit space requirements and avoid repeated calls to expensive formatting functions. Each event consists of a fixed-length header, optionally followed by a character string and a number of integers. The event header record includes process and thread identifiers, a cycle count time-stamp, and event details, such as system call arguments and return values.

5.5 Recovering Request Paths

Events are collected from each module and sorted by clock cycles. The correlation of external links between modules with the internal links within modules provides the causal path of a request. Two rules allow the construction of paths while scanning forward in time:

1. When a module receives a message associated with request r, a fragment initiates computation for r.
2. When a fragment computing r sends a message, that message is associated with r.

Moving forward through an event stream, BorderPatrol reconstructs the history of modules, the communication channels they engage in, and messages transmitted. As virtual time proceeds, BorderPatrol maintains a mapping from file descriptors to communication channels as they are created, duplicated or destroyed. During the execution of fragments, a module *designation* identifies which request the module is currently processing. Finally, events from protocol processors indicate when *messages* are transmitted or received. In accordance with Rule 2 above, these messages are associated with the sender's current designation. An event signaling receipt of such a message updates the recipient's designation.

BorderPatrol can only determine the single most direct cause of a given fragment's executions. For example, consider a module that invokes two other modules and proceeds when both have responded. BorderPatrol will recognize that the request has moved into each of the called modules, but when the first returns, no further action will be observed. When the second module returns, BorderPatrol will determine that the request has returned and will continue within the original module, perhaps to move on yet again. In summary, BorderPatrol correctly follows forks in the request processing path, but cannot directly detect joins because they reflect information about what might have happened (if a different module had returned last), rather than what did happen in the particular trace.

In addition to explicit module communication through IPC or data streams, causal paths also continue across process creation. Often a module will spawn a helper module to assist computation. For example when a web server receives a request for a CGI URL, it will `fork` a process which then `exec`s the CGI. Spawned modules consist of an implicit initial fragment which is associated with the same request that the parent was processing the moment it called `fork`.

The rules we use to recover request paths are similar in spirit to the work of Isaacs *et al.* [5] in which *temporal joins* correlate events in accordance with an application-specific join schema to reconstruct paths. BorderPatrol obtains ex-

plicit internal and external causal links, so it is immediately known when requests enter and exit modules.

As a result, BorderPatrol is application independent. In contrast to join schemas, protocol processors exist solely to identify request boundaries, and contain no application-specific information.

Events on a single host use the cycle count as a total order, but these clocks will not be perfectly synchronized across multiple hosts. Since messages are logged at transmission and receipt, an approximate mapping between clocks on each host can be obtained. This mapping is sufficient to order messages, though one-way delays will not be measured precisely.

6. CASE STUDIES

Before considering performance overhead in the next section, we first show how BorderPatrol copes with two typical scenarios that require manual instrumentation to obtain precise paths with previous tracing systems.

6.1 Multi-Threaded Tiers (dearinter.net)

dearinter.net is a social networking web site which invites users to post and vote on public questions. dearinter.net consists of a multi-threaded Python application tier (TurboGears [20]) between an Apache web server [3] front-end and a PostgreSQL database back-end.

The tiers of dearinter.net communicate using several standard protocols. Web requests arrive as HTTP requests, Apache forwards application requests to TurboGears as FastCGI messages, and TurboGears issues queries to the database through the PostgreSQL protocol. BorderPatrol contains protocol processors for each of these protocols. The processors are straight-forward, and none is longer than 150 lines of code.

Examining an access log excerpt from a typical page load motivates the need for event isolation using protocol processors. Here we see that a top-level "question" page is loaded, followed by almost simultaneous requests for several embedded images.

```
9:32:42.03 /question/521  HTTP/1.1 200  1949
9:32:42.24 /img/House.jpg HTTP/1.1 200 19317
9:32:42.30 /img/Mark2.jpg HTTP/1.1 200 18820
9:32:42.34 /img/Meter.jpg HTTP/1.1 200 19947
```

Figure 8 illustrates a portion of the events logged by BorderPatrol during this page load. Only the log entries for the Apache process are shown, in order to minimize details while motivating protocol processors. First, the client establishes a connection. The HTTP protocol processor recognizes a request for the URL `/question/521`. To service the request, Apache connects to the FastCGI server (not shown), which responds with data that is returned to the client. The images are also served through the dearinter.net application server.

BorderPatrol separates protocol messages when multiple messages occur over a single connection, a property shared by all of the protocols used by dearinter.net. Apache supports HTTP/1.1 pipelining over persistent connections, FastCGI allows multiple outstanding requests over a single socket, and the PostgreSQL protocol allows clients to issue multiple outstanding queries.

Notice the **ProtocolIsolate** event just after the request for `House.jpg`. As the application is reading, the HTTP processor notices the boundary between two HTTP requests.

KCycles	Event
2,000,585	ProtocolInit(3) → https
2,000,592	Accept(16,0) → (3,:60983-:80)
2,000,860	ProtocolMsgRecv(3,https) [/question/521]
2,002,447	Socket() → 5
2,002,524	ProtocolInit(5) → fcgic
2,002,526	Connect(5,0) → (:40682-:9797)
2,002,591	ProtocolMsgSend(5,fcgic) [URI=/q...]
2,432,164	ProtocolMsgRecv(5,fcgic)
2,432,201	Close(5)
2,432,260	ProtocolMsgSend(3,https) [200]
2,435,414	ProtocolMsgRecv(3,https) [House.jpg]
2,435,462	**ProtocolIsolate**(67,161)
2,436,817	Socket() → 5,
2,436,914	ProtocolInit(5) → fcgic
2,436,916	Connect(5,0) → (:40683-:9797)
2,436,969	ProtocolMsgSend(5,fcgic) [House.jpg]
2,559,082	ProtocolMsgRecv(5,fcgic)
2,559,135	ProtocolMsgSend(3,https) [200]
2,560,658	Close(5)
2,560,808	ProtocolMsgRecv(3,https) [Mark2.jpg]
2,562,252	Socket() → 5
2,562,348	ProtocolInit(5) → fcgic
2,562,351	Connect(5,0) → (:40684-:9797)
2,562,391	ProtocolMsgSend(5,fcgic) [URI=Mark2...]
2,596,653	ProtocolMsgRecv(5,fcgic)
2,596,703	ProtocolMsgSend(3,https) [200]
2,598,234	Close(5)
...	...

Figure 8: Log of events relevant to the Apache process with Event Isolation enabled on dearinter.net. Dashed lines indicate the beginning of a code *fragment*. Fragments begin at every input event and when `poll` indicates that a file descriptor has become writable.

Rather than passing the compound request to the application, it isolates and passes through the first of the two. Apache continues immediately by contacting the FastCGI server and relaying `House.jpg`. Afterward, Apache calls `read` again to collect the second request for `Mark2.jpg`.

To illustrate *independence*, we generated the same workload with event isolation disabled in BorderPatrol. Now a single call to `read` fetched multiple image requests. Regardless, Apache handled the requests sequentially—it created a connection to the FastCGI server, relayed the first image, and then repeated the process for the second image. This serial behavior is an artifact of Apache's architecture, not BorderPatrol.

This scenario is a concrete example of a module interaction that cannot be precisely deciphered without instrumentation using any other tracing tool. If Apache were to read in both requests it would be impossible to correlate which FastCGI connection corresponded to which client request. In this example, both requests are for images that are handled quite similarly, and we might happen to know that Apache handles requests sequentially. In general, the requests might be quite different, and require several module interactions to service. An error in constructing the causal path might, for example, attribute database access to a request for a static image rather than a dynamic Python page.

Validating Traces. We used BorderPatrol to create traces from a workload generated by actual dearinter.net clients from an access log provided by its developers. We validated the correctness of the traces in two ways: by detailed discussion with the developers and by comparison with ex-

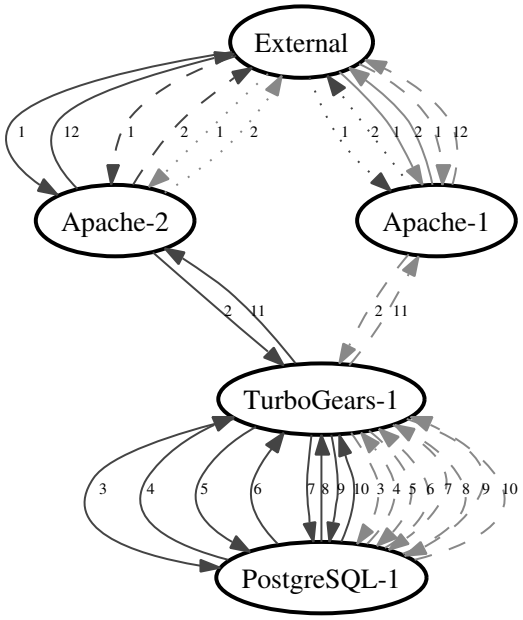

Figure 9: The original trace of the application shown in Figure 1. BorderPatrol made it easy to see that two requests where being handled by the database in a workload that should only require one. The application was reconfigured to serve images from Apache.

plicit instrumentation. During these discussions, the developers confirmed even the most surprising aspects of the trace, and found multiple opportunities for optimization.

We used the trace to create two types of user digestible output. First, we generated a graphical view of the paths through each module, as shown in Figure 9. Using this figure alone, the dearinter.net developers noticed their suboptimal configuration was causing access to the database for certain static images. With a minor modification, the application was reconfigured and yielded Figure 1, shown earlier.

Second, we found all PostgreSQL messages associated with specific URL requests. A request for `/tag/rabbits`, caused the following cycle-stamped queries:

```
316264 BEGIN; SET TRANSACTION ...
316522 SELECT NEXTVAL('tg_visit_id_seq')
317336 INSERT INTO tg_visit (id,visit_key,expiry)
       VALUES (419704,'5c4...','2007-03-18...
335990 SELECT expiry,... FROM tg_visit WHERE id = 419704
336605 END
479741 BEGIN; SET TRANSACTION ...
479891 SELECT id,user_id FROM tg_visit_identity
       WHERE visit_key = '5c4...'
484013 SELECT id,tag,count FROM tag WHERE tag = 'rabbits'
485311 SELECT id,tag,count FROM tag WHERE tag = 'rabbits'
485928 SELECT id FROM qu_tag WHERE exttag_id = 1528
487024 SELECT question_id FROM qu_tag WHERE id = 2914
487778 SELECT title,sum,weight,user_id,numcomments,...
       FROM question WHERE question_id = 1107
511741 SELECT user_name,email,... FROM tg_user ...
514104 SELECT id FROM qu_tag WHERE question_id = 1107
514841 SELECT tag_id FROM qu_tag WHERE id = 2911
515353 SELECT tag_id FROM qu_tag WHERE id = 2912
515782 SELECT tag_id FROM qu_tag WHERE id = 2913
516238 SELECT tag,count FROM tag WHERE id = 1525
516874 SELECT tag,count FROM tag WHERE id = 1526
517362 SELECT tag,count FROM tag WHERE id = 1527
539487 END
```

The first transaction generates a new unique browser cookie to act as a temporary user identifier. The second transaction generates the content of the page using several `select`s. First an identifier for the "rabbits" tag is obtained, from which the list of questions associated with that identifier can be loaded. The developers confirmed that the subsequent identical query was due to a previously unknown inefficiency in the structure of the application. Finally, statistics and related questions are loaded for each question (here, only 1107). BorderPatrol followed internal and external links to obtain the request path without knowledge of the internals of dearinter.net.

Indeed it would have been possible to obtain the above query log by simply inspecting PostgreSQL access logs. However, examining access logs is only effective when a single request is being served by the system. As soon as multiple requests arrive the log begins to conflate requests. Even if a given server's log has unique per-request identifiers (such as a session ID) it would not be possible to correlate the activity in one server's log with the activity generated by the same request in another server. To find the bug mentioned above, one would have to set up an instance of dearinter.net outside of a production environment and than manually inject requests one at a time. By contrast, BorderPatrol can automatically accumulate per-request information across all subsystems involved.

We also manually instrumented all tiers of the dearinter.net application. Modifications were necessary in Apache's accept loop, HTTP processing, and FastCGI implementation; in TurboGears' FastCGI library; in Python's SQLObject [16] library; and in Postgres' connection handling. Explicit instrumentation matched BorderPatrol's inferred traces precisely. Of course, each modification consisted of only a single line of code, but we were struck by the difficulty in two ways. First, three very different code bases were involved, spanning two languages. Second, it was difficult to know for sure that our instrumentation was correct and complete.

The instrumentation would have missed communication that we were not explicitly aware of, such as DNS lookups. BorderPatrol's interposition can observe all communication. Worse, our instrumentation would have been incorrect if the internal architectures of these programs had used multiple threads to satisfy a single request, or multiplexed requests on a single thread. Instrumentation code must explicitly observe the internal links that BorderPatrol infers.

To observe these links, request information can be placed in a data structure that is passed throughout the module and logged during communication. Unfortunately, real systems rarely pass a request structure through their entire code base. For example, SQLObject is not tightly tied to TurboGears. When instrumenting SQLObject, the request object is not accessible. In this case, it is convenient to keep the request object in thread-local storage. However, this technique is prone to error if the module to be instrumented does not have a one-to-one mapping of threads to requests. In these cases, we believe manual instrumentation would require pervasive understanding of the entire module to manage knowledge of the current request.

6.2 Event-Driven Web Server (Zeus)

Zeus [21] is an enterprise-scale commercial web server, only available to the public in binary form. We have no direct knowledge of the internals of Zeus, though we are

aware it is a high-performance event-driven design. Without source, instrumentation is impossible, and with an event-driven architecture, binding requests to threads will not succeed.

BorderPatrol traced Zeus, including its use of FastCGI, using the same protocol processors that traced dearinter.net, plus a DNS processor for Zeus's reverse lookups. A sample access pattern is shown below.

```
7:58:10.03 GET /.../index.fcgi?... HTTP/1.1" 200
7:58:10.16 GET /.../statimg.fcgi HTTP/1.1" 200
7:58:10.17 GET /.../1t.gif HTTP/1.1" 200
```

KCycles	Event
1,137,563	ProtocolInit(8,:41170-:80) → https
1,137,567	Accept(4) → 8
1,137,756	Socket() → 9
1,137,758	ProtocolInit(9) → dnsc
1,137,780	Connect(9) → (:32784-:53)
1,137,817	ProtocolMsgSend(9,dnsc,3668)
1,140,325	ProtocolMsgRecv(9,dnsc,3668)
1,140,350	Close(9)
1,140,387	ProtocolMsgRecv(8,http,0) [GET,index.fcgi]
1,141,262	Socket() → 9
1,141,342	ProtocolInit(9) → fcgic
1,141,346	Connect(9) → (/tmp/s.zeus)
1,141,540	ProtocolMsgSend(9,fcgic)
1,405,294	ProtocolMsgRecv(9,fcgic)
1,405,297	**ProtocolIsolate**(8,11683,0)
1,405,625	ProtocolMsgSend(8,https,0) [200]
1,407,236	ProtocolMsgRecv(9,fcgi)
1,407,238	ProtocolIsolate(8,16,0)
1,409,622	ProtocolMsgRecv(8,http,1) [GET,statimg.fcgi]
1,409,687	ProtocolIsolate(101,193,0) → 0
1,409,811	ProtocolMsgSend(9,fcgi)
1,409,862	ProtocolMsgRecv(8,http,2) [GET,1t.gif]
1,421,876	ProtocolMsgRecv(9,fcgi)
1,421,878	ProtocolIsolate(8,9716,0) → 0
1,422,567	ProtocolIsoPoll(0,0,0) → 2
1,422,590	ProtocolMsgSend(8,https,1) [200]
1,422,666	ProtocolMsgRecv(9,fcgi)
1,422,668	ProtocolIsolate(8,16,0)
1,422,927	Open(1t.gif) → 10
1,422,953	Close(10)
1,422,980	ProtocolMsgSend(8,https,2) [200]
...	...

Figure 10: Log of events with Event Isolation enabled on Zeus. Bold events show protocol processors demarcate message borders, detect data parameters, and perform event isolation (**ProtIsolate**).

The events collected are listed in Figure 10. As in the previous case study, activity begins with the arrival of a client connection. Zeus then connects to a name server to reverse resolve the client IP address. The DNS protocol processor tracks the outstanding DNS request using a witness that consists of the UDP 4-tuple and DNS request ID. BorderPatrol properly constructs paths and attributes time spent in remote, unmonitored modules.

After the name is resolved, Zeus reads an HTTP request from the client for `index.fcgi`. A FastCGI subprocess is forked and a connection is established via a Unix domain socket. Finally, Zeus writes a FastCGI message to the FastCGI server, receives the response, and relays it to the client. Next, the client requests a dynamic GIF (`statimg.fcgi`).

Although the requests are nearly simultaneous, overlap in time, and are handled by a single thread in Zeus, BorderPatrol can correctly correlate the FastCGI activity with `statimg.fcgi` rather than with `1t.gif`. This tracking does not require serialized requests, as Figure 10 illustrates. Event isolation supplies Zeus with the message for `statimg.fcgi` first, and Zeus immediately contacts the FastCGI server. With that connection in progress, Zeus returns to its event loop and receives the request for `1t.gif`. With the second request in hand, Zeus receives the response from the first request which is forwarded to the client. Finally, Zeus reads the static image and forwards it to the client.

Validating Traces. Zeus is a binary module, and we have no relationship with the Zeus developers, so validating BorderPatrol's traces was challenging. We carefully examined request paths to ensure that, for example, the expected files were opened on the path associated with the associated URLs and that paths reconstructed during a heavy, concurrent workload matched the individual paths reconstructed from single requests.

6.3 Other Cases

In addition to the above case studies that include traces through Zeus, Bind, Apache 1.3, TurboGears, and PostgreSQL, we have traced many other modules in combination with these components. BorderPatrol has successfully traced Perl scripts used as CGI and FastCGI components, multiple web servers such as thttpd and a Java web server, and several simple protocols using our generic "one-shot" and "line-oriented" processors.

7. PERFORMANCE EVALUATION

Our methodology introduces overhead. In this section, we quantify this overhead both as absolute micro-benchmarks, and under realistic workloads for the case studies examined in Section 6. Our experiments were conducted on a server with a single 2.0GHz Athlon CPU and 512MB of RAM. In all experiments the overhead of collecting and writing logs is included. The logging daemon is run on the local machine with the experimental application, although it could be run remotely.

`Libbtrace` is interposed between the application and `libc` at run-time. Library interposition by itself has negligible overhead: less than 1%. However, `libbtrace` contains initialization code that produces startup overhead. For example, when the first application call is trapped, pointers are initialized to the real `libc` versions of the routines that are interposed upon. Then BorderPatrol initializes various data structures and connects to the logging daemon. The first and all subsequent trapped application calls typically involve some logging, invoking protocol processors (in the case of I/O) and implement event isolation.

7.1 Micro-benchmarks

A series of micro-benchmarks is shown in Figure 11. We ran experiments measuring how the request latency of a web server (Apache) degrades under various workloads as the number of concurrent clients are increased. In each graph, the solid line indicates the control scenario – measurements of a pure Apache server. The dotted line indicates the measurement of Apache wrapped with our tracing layer `libbtrace`. All workloads were generated by closed-

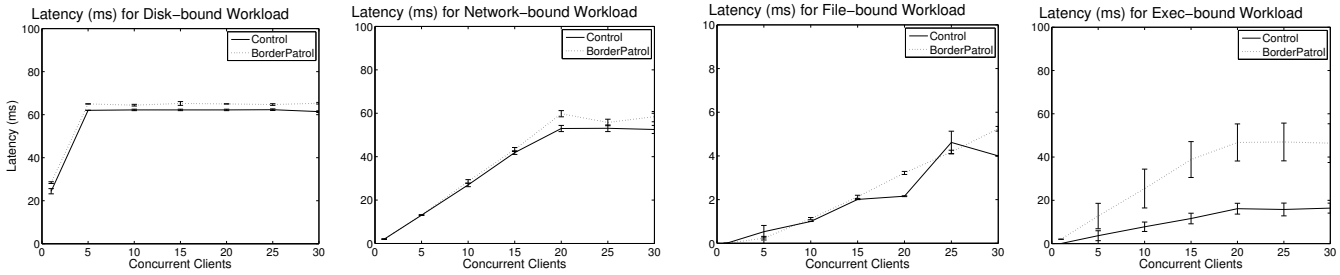

Figure 11: Latency overhead for three different micro-benchmark workloads. Each graph shows the untraced web server as a solid line, and the BorderPatrol traced version as a dotted line. The workloads are *disk*-bound, *net*work-bound, small-*file* bound, and fork/*exec*-bound.

loop feedback clients, so performance reaches a plateau at saturation.

The leftmost benchmark shows latency degradation under a disk-bound workload. We generated a variety of files at 10MB each, and the clients fetched random subsets. At around 5 concurrent clients, the server becomes saturated as it cannot serve files faster than they can be loaded from disk. The overhead was roughly the same for any level of concurrency, with an overall mean of 5.37%.

In the second benchmark, a workload was generated consisting of a single 1MB file, repeatedly fetched by increasingly many clients. The file is immediately loaded into the buffer cache, so this test measures the overhead of a network-bound workload. As compared with the disk-bound benchmark, it takes longer to reach a plateau but does reach a plateau when Apache maximizes its ability to use the network. Most of the overall 7.65% overhead came from the highly concurrent workloads.

The third benchmark shows the overhead for a workload consisting of one small file repeatedly fetched by concurrent clients. The file immediately is loaded into the buffer cache and so the workload is representative of system-call intense scenarios. Across the entire range of concurrency, the mean overhead was 37.2%.

Finally, the far right benchmark illustrates the overhead when workloads involve `fork` and `exec` operations. In this benchmark, clients access a URL served by a bare-bones C program. During `fork`, our implementation performs several initializations, such as connecting the child process to the logging daemon and allocating our bookkeeping data structures. Our wrapped `exec` call performs additional initialization such as looking up the *real* `libc` calls through dynamic linking, and more extensive bookkeeping initialization. The mean overhead for this benchmark was 307.7%, an enormous penalty, but one that is paid only for exec. Real workloads, particularly in performance critical applications, will not be exec-heavy.

In summary, our methodology generates the most overhead for workloads that involve a large amount of process creation because each time a new process is created some initialization routines must execute. However, if the process is subsequently used to load data from disk (such as a database) or communicate with other processes (such as a web or application server) the cost is negligible. We will now turn to realistic workloads that illustrate this point.

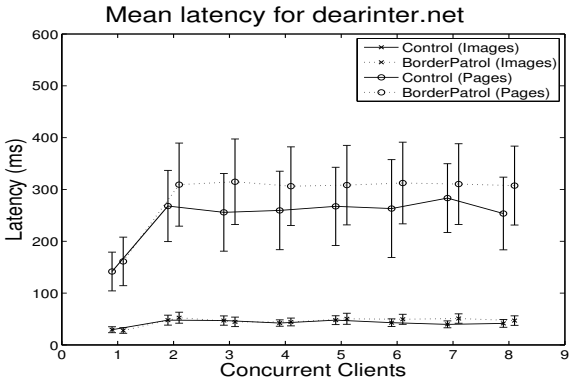

Figure 12: Latency and bandwidth overhead versus concurrent clients for dearinter.net (Apache, TurboGears and PostgreSQL) under a representative workload generated by replaying actual access logs.

Case Study	Events	Log (MB)	Time (s)
dearinter.net	603,962	21.63	46.29
Zeus	268,973	10.84	203.45

Figure 13: dearinter.net consumed approximately 470kB of log space per second during our benchmark runs. Zeus consumed approximately significantly less – 53kB per second – as the communication channels had few attributes to be logged.

7.2 Case Studies

We now revisit the case studies discussed in Section 6, and analyze the overhead for a more realistic day-to-day workload. In addition to the computational overhead that we discuss below, executing the application with BorderPatrol accumulates log entries as summarized in Figure 13. The logs are not particularly large and, of course, could be deleted when their likely value has declined.

dearinter.net. The overhead of our implementation on dearinter.net is shown in Figure 12. Here the workload involves more computation and random disk access than in the micro-benchmarks and so it quickly reaches capacity. Additionally, the workload includes both static and dynamic content, so we show the overhead for each in Figure 12. For the higher-latency dynamic pages, the overhead of our implementation is 16.96%, whereas the overhead is 8.4% for static images and JavaScript. The variance profile was unchanged with our tracing methodology enabled.

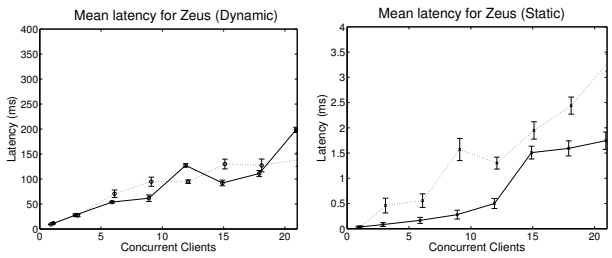

Figure 14: Latency and bandwidth overhead versus concurrent clients for Zeus (closed-source high-performance event-driven web server) under a workload of mixed images.

Application Component	System calls Traced	Untraced
Apache 1.3 and TurboGears	357248	324227
PostgreSQL	16194	14582

Figure 15: BorderPatrol introduces a modest overhead in terms of system calls. Apache and TurboGears execute 10% more system when traced. PostgreSQL executes 11% more.

Zeus. Figure 14 illustrates the latency overhead of our implementation. We generated two workloads: dynamic FastCGI pages to the left and static images on the right. The mean overhead for dynamic pages is 2.0%, while static images have a 96.4% overhead. Zeus is highly tuned for serving static pages that fit in memory, so it is unsurprising that BorderPatrol imposes a larger relative penalty. When serving dynamic content through FastCGI (a workload more reminiscent of the systems we are focused on) the overhead is lost in the noise. The 100% error bars also show that BorderPatrol has not negatively affected Zeus' concurrency profile by causing some requests to be queued excessively.

8. RELATED WORK

Previous work in request tracing has generally focused on either simplifying the instrumentation burden, or the use of statistical methods that eliminate instrumentation but also lose the ability to trace precisely.

Instrumentation. The most accurate way to correlate concurrent inputs with outputs is to leverage application-specific knowledge and explicitly declare which input corresponds to which output.

Magpie [5] seeks to provide precise traces of applications while minimizing the burden on developers. Their approach is two-pronged. First, they simplify instrumentation requirements by applying a general *temporal join* to logged events. A temporal join allows a submodule to emit trace events without knowledge of the global request that invoked it. Magpie builds a path by joining locally significant attributes across modules to produce a path. In addition, Magpie takes the pragmatic step of modifying an application framework, Microsoft's IIS and SQL Server. Modules written within this framework require no further modification, though calls to external libraries would not be traced.

TraceBack [4] uses program analysis to inject runtime instrumentation into modules that enables a source-statement reconstruction of program execution. From that reconstruction, Traceback attempts to reconstruct paths using techniques similar to [1].

A variety of commercially available products use similar techniques. These products instrument application frameworks (such as WebSphere, WebLogic, Oracle E-Business, and Siebel) with logging calls to annotate the nodes of a causal path. The products range from the simplistic 2-tier reconstructions in [17] to many-tier reconstructions in [18, 15].

As we found in our experience with dearinter.net (see Section 6.1), a shortcoming of instrumentation is a practical one: all points in the application where inputs arrive must be modified. In large-scale applications where components span developer groups, are written in multiple languages, and may lack source code, modifying the application (or the frameworks) is not always possible. Further, developers may need to modify the application to make necessary information available at the time it is needed for logging, adding to their burden. Magpie's general temporal join seeks to reduce this requirement though it is not clear that a join-able attribute is easily accessible in applications that multiplex requests on threads.

More distantly related, King [9] shows how information flow can be used to detect intrusion. Like BorderPatrol, the work tracks the inputs and outputs of processes. However, where BorderPatrol uses knowledge about protocols to distill a causal tree into per-request branches, King's work simply collects the causal tree entirely so that a root cause can be found.

Pervasive Frameworks. Alternatively, some approaches enforce infrastructure change. Specifically, the interface of all modules is widened to include request information. This work assumes that all participating modules will be modified to implement the new interface.

Pinpoint [7] is designed specifically for J2EE web applications that associate each request with exactly one thread. This association allows any module to record the request it is working on by examining a thread-local variable. By contrast, BorderPatrol, allows applications to be written in almost any language, to use a variety of execution models (multi-threaded/event-driven), and to cross process and machine boundaries.

Causeway [2] advocates pervasive changes to applications and protocols in order to bundle meta-data alongside existing module communication. X-Trace [8] is philosophically similar work that focuses on debugging paths through many network layers. Each layer must be modified to carry X-Trace meta-data that allows path reconstruction. BorderPatrol focuses on tracing without changing applications.

Probabilistic Correlation. An alternative approach avoids augmenting the control- or data-flow by compromising on precision: the correlation between inputs and outputs can be done statistically. HP Labs has used this approach on network traffic [1], and more recently [14] on a per-process granularity using library interposition. In both cases, causality is inferred from the relative time-stamps of input arrivals and output departures.

Whodunit [6] obtains *transactional profiles* that follow request hand-offs that occur in shared memory, invisible to BorderPatrol's tracing mechanism, by observing and analyzing module lock usage. Whodunit obtains aggregate performance information, rather than precise traces of individual requests. Nonetheless, it represents an interesting complementary approach that could augment BorderPatrol in the context of difficult to trace architectures.

Analysis From Causal Paths. Analysis of causal paths is an emerging area of research. These analyses [13, 7] *assume* causal paths can be obtained and perform higher-order analysis such as failure detection or capacity planning. BorderPatrol provides the opportunity to take advantage of this research in systems that are otherwise untraceable. Pip [13] finds bugs by dynamically detecting request paths that deviate from paths specifications provided by developers. Pinpoint [7] finds faulty modules by recording the modules involved in handling each request and applying data mining techniques to failure cases. BorderPatrol could be used to ease the adoption of these promising tools.

9. CONCLUSIONS

The lesson of BorderPatrol is that traces can be obtained for unmodified programs without sacrificing precision. We present a model for understanding the behavior of black-box distributed systems. Our model allows us to safely employ a mechanism that isolates conflated input events arriving at a module, without preventing the application from multiplexing requests. Rather than requiring systems to adopt new conventions to make request paths explicit, we are able to automatically extract causal paths through active observation.

BorderPatrol is freely available from `http://cs.brown.edu/research/borderpatrol/`.

Acknowledgments

The authors would also like to thank Yanif Ahmad, Shriram Krishnamurthi, Kiran Pamnany, Steve Reiss, and the anonymous reviewers for their valuable feedback on improving the presentation.

This research supported by the National Science Foundation under Grant No. CNS-0614944. We thank the NSF for their support.

10. REFERENCES

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[2] Khaled Elmeleegy Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*. IEEE Computer Society Technical Committee on Operating Systems, 2005.

[3] Apache HTTP server. `http://httpd.apache.org/`.

[4] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: First fault diagnosis by reconstruction of distributed control flow. In *Proc. of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation (PLDI'05)*, 2005.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (SOSP'04)*, December 2004.

[6] Anupam Chanda, Alan Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proc. of the 2nd European Conference on Computer Systems (EuroSys'07)*, March 2007.

[7] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. of the International Conference on Dependable Systems and Networks (IPDS Track)*, 2002.

[8] Rodrigo Fonseca, George Porter, Randy h. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proc. of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'07)*, April 2007.

[9] Samuel King Analyzing Intrusions Using Operating System Level Information Flow. Ph.D. thesis. September 2006.

[10] libevent. `http://www.monkey.org/~provos/libevent`.

[11] David Mazières. A toolkit for user-level file systems. In *Proc. of the General Track: 2001 USENIX Annual Technical Conference*, 2001.

[12] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server. In *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.

[13] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. PIP: Detecting the unexpected in distributed systems. In *Proc. of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'07)*, May 2006.

[14] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: Blackbox performance debugging for widearea systems. In *Proc. of the 15th International World Wide Web Conference (WWW'06)*, May 2006.

[15] Quest Software®. PerformaSure®. `http://www.quest.com/performasure/`.

[16] SQLObject. `http://www.sqlobject.org/`.

[17] Symantec. Indepth. `http://www.symantec.com/enterprise/products/category.jsp?pcid=1021`.

[18] Wily Technology. Introscope®. `http://www.wilytech.com/solutions/products/Introscope.html`.

[19] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proc. of the ACM SIGMETRICS Conference*, June 2006.

[20] TurboGears. `http://www.turbogears.org/`.

[21] Zeus web server. `http://www.zeus.com/`.