BorderPatrol: Isolating Events for Precise Black-box Tracing

John Jannotti Eric Koskinen
Brown University
{jj,ejk}@cs.brown.edu

Abstract

High-level causal request traces are of interest to developers of large concurrent and distributed applications. These traces show how a request is processed as it passes through several modules which may be processes, threads, machines, or devices. They aid programmer understanding and are increasingly analyzed by tools used to detect performance and correctness errors. *Precise* traces are more useful than statistical approaches because they can detect anomalous behavior and allow decisions at run-time. Since these traces are difficult to obtain without application-specific instrumentation of each module of the system, much of the recent work that analyzes request traces is limited to applications for which source code and developer expertise is available.

We present BorderPatrol, which obtains precise request traces through systems built from a litany of unmodified modules, written in varied languages, with varying architectures. These include Apache, thttpd, PostgreSQL, TurboGears, BIND and notably Zeus, a closed-source event-driven HTTP/1.1 web server, which uses helper processes. BorderPatrol obtains these traces using *active observation* which slightly modifies the event stream observed by system modules, simplifying precise observation. *Protocol processors* aid active observation by leveraging knowledge about standard protocols and interfaces between concurrent modules, avoiding the need for implementation-specific instrumentation.

BorderPatrol obtains precise traces for black-box systems that cannot be traced by any other technique. Further, it does so with limited overhead on real systems (approximately 10-15%) making it a viable option for deployment on production systems.

1 Introduction

Today's large-scale applications consist of independent modules (processes, threads, devices) that leverage concurrency for performance. In many cases, the components are developed by different groups and in different languages. Individual components may use 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. Tracing tools must follow single requests as they are passed between modules, including third-party binary modules and even as those requests are passed and returned from remote, untraced systems.

Beyond building traces for inspection by developers, recent work has shown that these traces can be valuable input to automated tools. Systems such as Pinpoint [7], Pip [13], and Stardust [18] rely on precise request traces to identify faulty modules, discover anomalous request paths, and make capacity plans.

Unfortunately, obtaining precise request traces in a heterogenous, 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. While some 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. Generally, requests will be 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 propagate 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.

Module parallelism is our chief challenge. When a modules operates on two requests simultaneously, it is impossible to know which of its actions are attributable to which request.

Previous tracing systems have also recognized this impossible challenge, and have compromised in various ways. Many require *developer instrumentation.* Developers must manually record request transitions in order to reconstruct paths from the resultant bread-crumbs. Others provide the instrumentation for the developer, but only as part of a *rigid framework.* For example, a web application framework might make appropriate callouts to a tracing infrastructure without help from the developer. However, if the developer makes an ad-hoc call to a module that is not supported by the framework, the path is incomplete. Finally, some systems *accept imprecision.* Rather than following the path of a request precisely, a statistical model is built from the repeated observation of module inputs and outputs. Although paths can be obtained for unmodified modules with some probability, precise traces of specific anomalous paths cannot be determined. We discuss all of these approaches in much more detail 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 3.2) 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 3.1) identify matched messages, often request/response pairs. Event isolation and message witnesses are provided by *protocol processors* (Section 3.3), an abstraction that allows developers to implement protocol-specific, rather than implementation-specific, tracing. A single HTTP protocol processor can be used to trace any web server, web proxy, or even XML-HTTP services.

The techniques described in this paper are realized in a tool called BorderPatrol, which is publicly available. 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 the source code or the use of statistical methods. Further, we show that the overhead of tracing is about 10-15%, making it a viable technique for using at runtime on industrial-grade software.

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 generally 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. Each of these monitoring techniques imposes a module *resolution*—the size of black boxes. Network monitoring, for example, imposes a machine-sized black box. We assume that BorderPatrol's interposition and kernel based monitoring is sufficiently fine-grained to observe the passage of requests between black boxes.

Internal links, on the other hand, cannot be directly observed by the very nature of "black box." BorderPatrol assumes that the unobserved internals of black-box modules 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.

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 request/reply modules and in proxy servers. We refer to these identifiers as *witnesses* and BorderPatrol assumes they are accurate if they exist. We expect that bugs at such a low level are unlikely in production systems. However, witnesses are currently 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.

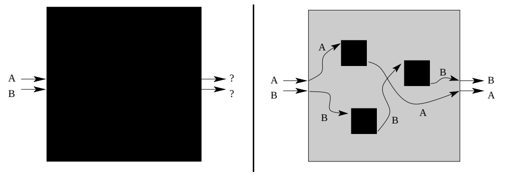

Figure 1: An illustration of the *immediacy* and *independence* assumptions. Immediacy tells us that when A is supplied, the black box's next observable action will be to create the output labeled A. Independence tells us that the black box would not have treated A differently had it been supplied simultaneously with B.

2.2 Immediacy

Usually, determining internal links is more difficult than matching witnesses. Modules may receive concurrent requests on one or more input channel, and then initiate connections to several other modules. The protocols used may be unrelated and the follow-on messages may carry no identifying information that can be tied to the original request.

The difficulty for internal link inference is that modules multiplex requests. For example, event-driven systems rotate servicing a number of outstanding requests. Additionally, a single process may collect multiple inputs (via `read` for example), and work on both 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*. A fragment is an internal control path that handles individual inputs and processes them until completion. Informally, a fragment is a short stretch of code during which the process performs internal computation on behalf of a single request.

These fragments do *not* usually 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 more detail in Section 5.5.

Immediacy is illustrated in Figure 1. 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.

In the Section 3, we explain how BorderPatrol takes advantage of this assumption, and in Section 4 we explain why traced modules retain their ability to multiplex requests.

2.3 Independence

Our final assumption about black-box modules is independence. We assume that modules will treat two sequential inputs the same way they would have treated those 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. Inside most real-world modules, multiple input events will be immediately separated anyway. Libraries such as `libevent` [9] and `libasync` [10], dispatch to event handlers for individual events. Even if multiple events are supplied to an event-driven module, the event loop dispatches these events serially to event handlers. In threaded applications, events are serialized to an even greater extent. These applications usually block 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 events are isolated.

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.

3 Active Observation

We have created BorderPatrol to monitor and modify communication and control channels between modules to simplify request tracking. BorderPatrol uses *active observation* to observe and subtly modify the event streams sent and received by monitored modules. The goals is to ease request tracking by logging identifying information and to schedule module input to allow independent observation of the handling for each event.

In this section we discuss the techniques that allow BorderPatrol to isolate events and execute fragments independently. BorderPatrol uses simple *protocol processors* to understand and separate multiple messages on a single channel. These processors also supply witnesses that allow path reconstruction even when certain types of external modules are used where BorderPatrol is not deployed.

BorderPatrol's active observation is more intrusive that previous tracing approaches. As such, there is a greater risk that BorderPatrol's observation changes the behavior of the modules it traces. Sections 4 and 7 explore this concern in detail. Until then, we note only that we believe, and our experiments have borne out, that the vast majority of applications are essentially unaffected. Our intuition was based on the understanding that modules written to handle concurrent requests over multiple communication channels must be impervious to small message delays. Our experimental evidence is the correct and efficient operation of several large software systems operating under active observation.

3.1 Message Witnesses

As noted in the previous section, black-box tracing is complicated mostly by the need to establish internal links from module inputs to outputs. Message witnesses, when available, greatly simplify this problem. A message witness is data that can be extracted from input messages and output messages to allow matching, for example a request ID. Unfortunately, witnesses are unlikely when input and output messages are of different protocols, so they are useful mainly for linking requests and replies. BorderPatrol does not normally use witnesses, preferring to follow the path of a request more directly using event isolation. However, witnesses provide the ability to construct traces where event isolation is impossible because a module can not be controlled by BorderPatrol. For example, witnesses allow the integration of remote web service modules info BorderPatrol traces.

3.2 Event Isolation

In order to directly follow internal links, BorderPatrol supplies input event to modules one at a time. BorderPatrol monitors the module's output, and assumes that it 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. By returning only one event at a time to the black-box module, BorderPatrol can attribute the work of the module to the work of the request associated with the event. The module will then call `poll` again, and one more event will be supplied.

The events returned by poll are indivisible, they can be attributed to only one request. BorderPatrol has predetermined the association of each file descriptor to a given request, based on when the descriptor was opened, or the request of the input message to be read next. 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 multi-request reads.

3.3 Protocol Processors

BorderPatrol presents 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. 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 most basic "envelope" of the protocol messages.

Protocol processors look for message delimiters or length counts in the data stream, and look into 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. The protocol processors we have implemented are between 30-150 lines of code, including some boilerplate, as shown in Figure 3.

The protocol processor interface consists of just 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 a channel is closed, processors deallocate the memory 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 safely be passed 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. Otherwise, the entire buffer will be passed on, including cases in which the buffer represents a partial message. In these cases, for the convenience of protocol processors, BorderPatrol will buffer the partial message and re-invoke `pp_read` on the old data. 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. All data is passed through to the output chan-

```
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 2: 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 can always operate from the start of the message.

nel immediately, so pp_write will be invoked until it returns PP_NEED_MORE or 0. Most of the time, outbound data need not be cached, except for when the processor returns PP_NEED_MORE in which case the partial message is cached until more outbound data can be shown to the processor.

Two protocol processors must be written for most protocols. The read and write functions, described above, 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.

An example pp_read for HTTP is shown in Figure 2. HTTP is a simple, sequential protocol in which each request is separated by two pairs of linefeed/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 out of the header and logged.

BorderPatrol's real HTTP processor is 105 lines long. Figure 3 shows line counts for several other protocol processors, each less than 150 lines long. Section 5 discusses the implementation details that support this simple interface.

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 3: 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.

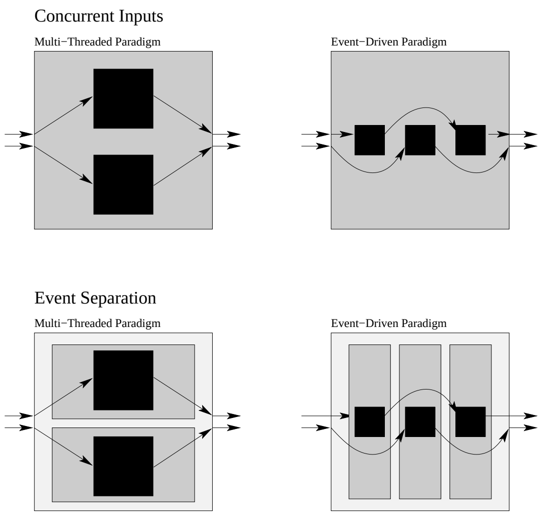

Figure 4: BorderPatrol works because real-world servers have straightforward internal structure. Multi-threaded servers dispatch events independently, to separate threads (left). Event-driven server execute in fragments that can be pieced together by running them sequentially (right).

4 Why does BorderPatrol work?

Do real-world applications decompose cleanly to code fragments that operate on individual requests? 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 (nearly) continuously accept new requests, 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);
```

Once inside `handle_request()`, such a server is well-behaved in the sense of our assumptions about immediacy and independence. While inside this function, 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* assume that sequential behavior, across fragments, is necessarily related. BorderPatrol actively follows the request back into the web server. It does not assume the web server continues on the same request across multiple input events.

Single Process Event-driven. SPED servers have a drastically different architecture. Rather than multiplexing requests across multiple threads, all computation is contained within a single thread, which multiplexes among the requests it handles. 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 five 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. In fact, 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. BorderPatrol obtains control at the start of each handler, so it can determine the request designation at the start of each handler. An illustration of this architecture appears on the right-hand side of Figure 4.

Asymmetric Multi-Process Event-Driven AMPED is largely the same as SPED, 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 of a single request can be handled by the "One-shot" protocol processor. We expect that these ad-hoc protocols are conventiently delimited or used fixed frame sizes. We expect implementations in the 10s of lines.

Workqueue. 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. Workqueues 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] is an effort 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.

User-level Scheduling. User-level threads may also present difficulties for BorderPatrol, depending on implementation. Cooperative thread packages switch threads 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. Other thread packages use asynchronous signals in order to support preemption. BorderPatrol does not current support the interception of these signals. Even if it did, treating the package as a black box would prevent BorderPatrolfrom knowing which thread has been swapped in. BorderPatrol could only resume tracing when an interaction with a known resource is observed.

5 Implementation

In this section we discuss our implementation of the mechanisms described in Section 3. Briefly, trace data is collected from unmodified modules by library interposition and a kernel module. The trace data is aggregated and processed in one "forward temporal join," to obtain request paths. An overview of the architecture is given in Figure 5.

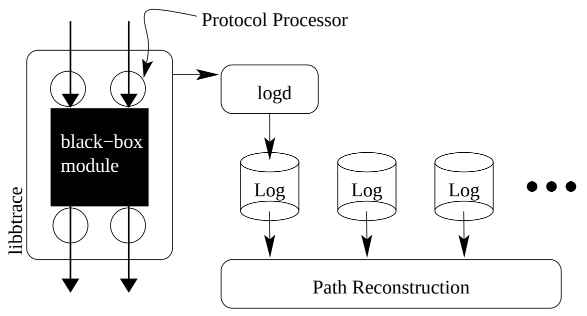

Figure 5: System Overview. The solid black box represents a traced application module. Communications (messages, IPC, and signals) are monitored by the Protocol and/or API processors. Events are relayed through the Logging Daemon to a raw database. Databases from multiple hosts are then aggregated, and causal paths are reconstructed.

5.1 Library Interposition (`libbtrace`)

The core of BorderPatrol is a series of wrapper functions for roughly 20 standard library calls in a library, `libbtrace`. Using library interposition (i.e. `LD_PRELOAD` on Linux), `libbtrace` intercepts calls to `libc`, isolates events, invokes protocol processors, and emit logging events. Usually, a wrapper invokes the real `libc` routine as a part of its work.

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

`Libbtrace` also tracks data as it flows through `read` and `write` operations (including variants such as `send` and `recv`). Many connections need not 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 have thought of cases such as mail servers in which we could track causality through the file system). For simple file operations, BorderPatrol simply logs the interaction.

By contrast, requests *do* flow over the other 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. We see no reason why these interfaces pose fundamental challenges, but we have not seen them in use enough to motivate their inclusion.

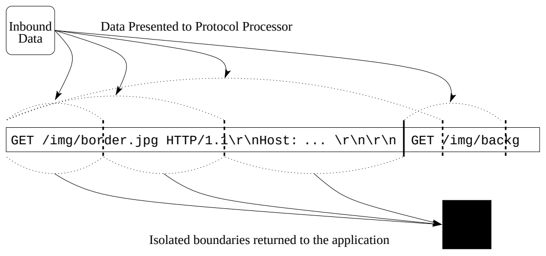
Inbound Data
Data Presented to Protocol Processor
GET /img/border.jpg HTTP/1.1\r\nHost: ... \r\n\r\n GET /img/backg
Isolated boundaries returned to the application

Figure 6: Example of data being presented to the Protocol Processor and data being forwarded on to the application.

Protocol Selection For any connection, the appropriate protocol processor must be selected. One might accomplish this with an identification function that behaves much like the Unix `file` command. By examining the first few bytes of data on a channel, it could converge on a protocol identity. For example, HTTP is easily recognized by the initial string `"GET ..."`, whereas FastCGI begins with a binary record format. Currently BorderPatrol determines protocol identity based on conventions such as port number, Unix domain path, or executable name.

5.2 Protocol Processors

`libbtrace` additionally houses all protocol processors. Each protocol is implemented once, and then it can be used on any application that implements the protocol. Section 3 describes the protocol processor interface; this section provides implementation details.

When data arrives on a channel with a protocol processor, the data cannot be written directly into the application's buffer since it may contain multiple input events that should be isolated. Additionally, once the protocol processor demarcates message boundaries, the messages may exceed the size of the buffer that the application made available.

For *inbound data*, BorderPatrol interposes on `read`. If there is a protocol processor for the file descriptor the inbound data is handled as follows:

```
ssize_t read(int fd, void *buffer, int len) {
  if (tab[fd].hasProcessor()) {
    if(tab[fd].hasCachedDataForApp())
      return tab[fd].returnUpTo(buffer,len);

    if(tab[fd].hasCachedDataForProcessor()) {
      tab[fd].AdvanceProcessor();
      if(tab[fd].hasCachedDataForApp())
        return tab[fd].returnUpTo(buffer,len);
    }

    ssize_t r = real_read(fd,buffer,len);
    if(r<=0) return r;
    tab[fd].appendCache(buffer,r);
    tab[fd].AdvanceProcessor();
    assert(tab[fd].hasCachedDataForApp());
    return tab[fd].returnUpTo(buffer,len);
  }

  else return real_read(fd,buffer,len);
}
```

Tab tracks data flowing through each file descriptor and maintains cursors to indicate which portions have been sent to the application, presented to the protocol processor, and not yet considered.

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 *outbound* data is far simpler. Application writes 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 call to `poll`. The buffered file descriptor should be 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 installs 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-tamps are passed through `relayfs` to a user-space daemon, which forwards them to the logging daemon.

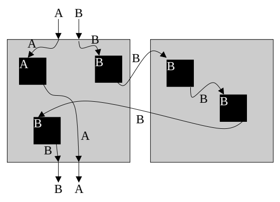

Figure 7: Correlating the outputs of atomic computation units with the inputs of others.

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.

Figure 7 provides a diagram of communication between fragments, illustrating these rules.

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.

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 explicit 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 can use the cycle count as a total order, but these clocks may not be synchronized across multiple hosts. Since we track message transmission and receipt, we can obtain a mapping between clock cycles on multiple hosts.

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 writeable.

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 in previous tracing systems.

6.1 dearinter.net

dearinter.net[1] 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 [19]) between an Apache web server [3] front-end and a PostgreSQL database back-end.

The tiers of dearinter.net inter-operate by communicating using several standard protocols. Web requests arrive as HTTP requests, Apache forwards application requests to TurboGears as FastCGI messages, and finally TurboGears issues queries to the database through the PostgreSQL protocol. BorderPatrol contains protocol processors for each of these protocols. All 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 (for brevity, we elide several irrelevant fields and renamed some images).

```
clienthost 9:32:42.03 /question/521  HTTP/1.1 200  1949
clienthost 9:32:42.24 /img/House.jpg HTTP/1.1 200 19317
clienthost 9:32:42.30 /img/Mark2.jpg HTTP/1.1 200 18820
clienthost 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 incoming data as an HTTP request for the URL `/question/521`. To service the request, the, 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.

Apache, like most other modern web servers, supports HTTP/1.1 pipelining over persistent connections. The elided protocols also send multiple messages over a single connection: from the web server to the FastCGI process, and from the FastCGI process to the database.

Notice the **ProtocolIsolate** event just after the request for `House.jpg`. As the application is reading, the

[1]Though dearinter.net is not our site, we have used a pseudonym due our association with the developers.

HTTP Protocol Processor notices the boundary between two HTTP requests. Rather than passing the compound request to the application, it isolates the first of the two. Apache immediately contacts the FastCGI server, relays `House.jpg`, and *then* calls `read` again to collect the second request for `Mark2.jpg`.

We generated the same workload with event isolation disabled. 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 after forwarding it to the client, repeated the process for the second image. This serial behavior is an artifact of the MP architecture, not BorderPatrol. (The recreation of the FastCGI connection is an artifact of a poor FastCGI implementation in Apache, an artifact that BorderPatrol does not rely on for correct operation.)

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. From the dearinter.net workload we used – based on access log files provided to us by the developers – we extracted request traces. Our methodology generated complete traces, from which we extracted the relationship between client requests (URLs) to messages to the PostgreSQL tier that contain SQL queries. Taking a single request (one for the URL `/tag/rabbits` for example) the corresponding queries and cycle counts where found to be:

```
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
```

We spoke with the developers of dearinter.net and confirmed that these queries match the structure of the application, and therefore this request trace is correct. The first transaction corresponds to the user authentication code, which generates a new unique identifier to store as a browser cookie. Subsequently, the second transaction begins by looking up the user's identity, and generating the content of the page. 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 an inefficiency in the structure of the application. Finally, for each question (in this case there's only one: question id 1107) and associated author, statistics are loaded, as well as the list of all the other tags associated with the question. Through each state, BorderPatrol followed internal and external links to obtain the causal path without knowledge of the internals of dearinter.net.

6.2 Event-Driven Web Server (Zeus)

Zeus [20] is an enterprise-scale commercial web server. Being a commercial product, the source is unavailable to us. We have no direct knowledge of the internals of Zeus, though we are aware it is a high-performance event-driven design.

Included with Zeus is an extensive Administration web application, which allows for the configuration of Zeus and the web sites it serves. Since Zeus is based on HTTP and FastCGI, we reused the protocol processors implemented for dearinter.net; no additional work was necessary. Loading the Cluster Configuration page of the Administration application yields the following access log entries, among others.

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

The events collected are listed in Figure 9. As in the previous case study, activity begins with the arrival of a client connection. However, Zeus subsequently connects to the 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

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

and relays it to the client. Shortly thereafter, the client requests a dynamically rendered GIF (`statimg.fcgi`[2]) and a static GIF (`1t.gif`).

Although the requests are nearly simultaneous, BorderPatrol can correctly correlate the FastCGI activity with `statimg.fcgi` rather than with `1t.gif`. This tracking does not come at the expense of serializing requests as Figure 9 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`. Having both balls in the air, it next receives the response from the FastCGI server which it forwards to the client. Finally, Zeus reads the static image off the disk and forwards it to the client.

Validating Traces. From these events that we captured with event isolation enabled, we reconstructed request paths. In the absence of a representative for Zeus, we carefully examined these paths to see if they matched our intuition.

[2]This file is not part of the standard administration application package – we created it for illustrative purposes

6.3 Other Cases

In addition to these detailed case studies that include traces through Zeus, Bind, Apache 1.3, TurboGears, PostgreSQL, we have traced many other modules in combination with these components. We have successfully traced Perl scripts used as CGI and FastCGI components, multiple web servers such as thttpd and a Java web server.

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. All of our evaluations were run on a server with a single 2.0GHz Athlon CPU and 500MB of RAM.

When an application is traced with BorderPatrol, our shared library `libbtrace` is interposed between the application and `libc`. Library interposition by itself has negligible overhead: less than 1%. However, `libbtrace` contains our core implementation, and there are several routines that produce a small amount of overhead. When the first application call is trapped, we initialize pointers to the *real* `libc` version of some of our methods since we will need them within `libbtrace`. Additionally, we initialize some data structures, and connect 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 10. We ran experiments measuring how latency of a web server (Apache) degrades under various workloads as concurrent clients are increased. Bandwidth graphs are omitted for brevity. 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-loop feedback clients, so performance reaches a plateau at saturation.

The upper left 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. Across the entire range of concurrency, the mean overhead was 5.37%.

In the upper right benchmark, a workload was generated consisting of a single 1MB file, repeatedly fetched by increasingly many clients. The file immediately is

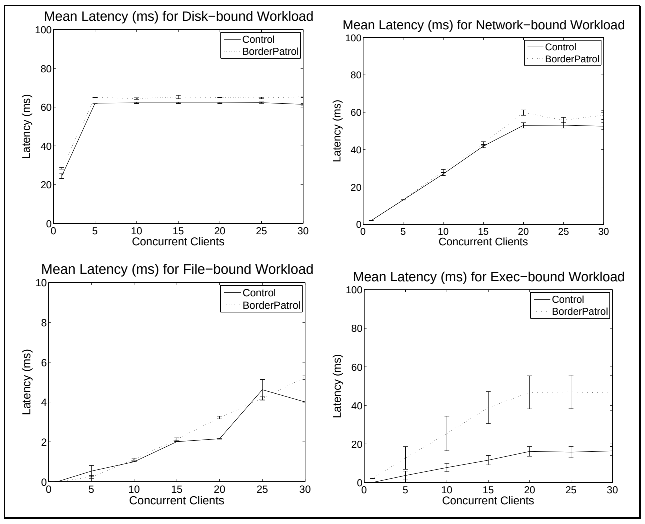

Figure 10: Latency overhead for three different micro-benchmark workloads. Each graph shows the control (untraced web server) as a solid line, and our implementation as a dotted line. The workloads are *disk*-bound, *net*work-bound, and *exec*/fork-bound.

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 it's ability to use the network. Across the entire range of concurrency, the mean overhead was 7.65%.

The third benchmark in the lower left 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 lower right benchmark illustrates the overhead when workloads involve `fork` and `exec` operations. In this benchmark, clients requested a CGI written in C. Perl CGI scripts had a slightly higher latency (due to Perl initialization) and so were less directly indicative of our implementation's overhead. 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%.

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 there 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 quickly amortized. We will now turn to more realistic workloads that illustrate this point.

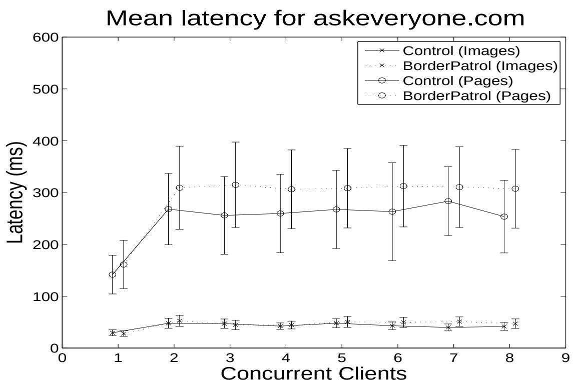

Figure 11: Latency and bandwidth overhead versus concurrent clients for dearinter.net (Apache, TurboGears and PostgreSQL) under a representative workload generated by replaying actual access logs.

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

Figure 12: 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 time-wise overhead that we discuss below, executing the application with BorderPatrol accumulates log entries as summarized in Figure 7.1. 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 11. 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 11. 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.

Zeus. Figure 13 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

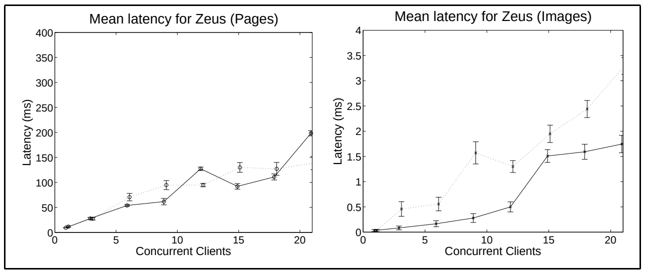

Figure 13: Latency and bandwidth overhead versus concurrent clients for Zeus (closed-source high-performance event-driven web server) under a workload of mixed images.

Application Component	Traced	Syscalls
Apache 1.3 and TurboGears	No	324227
Apache 1.3 and TurboGears	Yes	357248
PostgreSQL	No	14582
PostgreSQL	Yes	16194

Figure 14: BorderPatrol introduces a modest overhead in terms of system calls. Apache and TurboGears execute 10% more system when traced. PostgreSQL executes 11% more.

FastCGI, a workload more reminiscent of the systems we are focused on, the overhead is lost in the noise. 100% error bars also show that BorderPatrol has not negatively affected Zeus' concurrency profile.

8 Related Work

To the best of our knowledge, no previous work has attempted to determine request paths without application-specific instrumentation or resorting to statistical models. Fundamentally unique to our work is the *active isolation of black-box inputs* which allows us to precisely observe request paths without materially affecting an overall application's ability multiplex requests. In this section we discuss some previous approaches.

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.

One such technique is Magpie [5] which 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. At entry or exit to the submodule, a second trace event is recorded that matches the last module to the current module. Magpie builds a path by joining these 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.

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 [17, 16, 15, 11] also 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 [16] to many-tier reconstructions in [17, 15].

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, though Magpie's general temporal join seeks to reduce this requirement.

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], whose focus is described further in below, is designed specifically for J2EE web applications that associate each request with exactly one thread. This direct association allows any module to record the request it is working on by examining a thread-local variable. By contrast, the technique presented in this paper obtains request paths automatically, allowing 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 a pervasive change to applications and network 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 some degree of 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, an approach which is not always correct.

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.

Analysis From Causal Paths. Analysis of causal paths is an emerging area of research. These analyses *assume* causal paths can be obtained and perform higher-order analysis such as failure detection or capacity planning.

One such example is Pip [13], which has shown that bugs can be found by specifying how causal paths flow through a distributed system and dynamically detecting deviations.

Another such project is Pinpoint [7], whose focuses is on problem determination. Their goal is to find faulty modules by recording the modules involved in handling any particular request and applying data mining techniques to the cases that can be categorized as failures. As such, their contribution is mainly their technique for finding faults once traces have been obtained. As mentioned above, they use a domain-specific data-flow tagging approach rather than automatically determining request designation. BorderPatrol could be used to ease the adoption of any of these 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.

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*, 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. 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 *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, 2005.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, 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*, March 2007.

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

[8] X-Trace: A Pervasive Network Tracing Framework. Rodrigo fonseca and george porter and randy h. katz and scott shenker and ion stoica. In *Proc. 4th USENIX/ACM Symposium on Networked Systems Design and Implementation*, April 2007.

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

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

[11] Mercury^{TM}. Diagnostics. `http://www.mercury.com/us/products/diagnostics/`.

[12] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server. In *Proc. 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. 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation*, 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. 15th International World Wide Web Conference*, May 2006.

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

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

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

[18] 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. ACM SIGMETRICS Conference*, June 2006.

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

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