Resolving Ambiguous Paths Using BorderPatrol

Ning Shi
Brown University
ning@cs.brown.edu

Abstract

BorderPatrol successfully uses active observation to obtain precise causal paths as requests traverse many black-box modules, such as Apache, Zeus, Postgres, MySQL, and BIND. However, BorderPatrol was limited in several ways that prevented operators from relying on these traces. Several assumptions were made about the internal designs of traced modules, and when these assumptions were violated, BorderPatrol returned incorrect paths without warning.

Here, we have shown how BorderPatrol can apply conflicting evidence to detect assumption violations, backtrack to discover possible explanations for these conflicts, and use "known-good" paths to advise users which possible explanation is most likely.

In addition, we demonstrate several enhancements that make BorderPatrol suitable for tracing web applications in common deployment scenarios, including multi-machine tracing, new protocol processors, and greater flexibility in protocol determination. We demonstrate BorderPatrol's ambiguous path resolution and real-world applicability by tracing HAProxy [1], Lighttpd, and MySQL exactly as it is normally deployed.

1 Introduction

BorderPatrol [2] was built to solve the problem of discovering the relationships between modules and constructing request paths in complex distributed applications. It gives developers and maintainers information about what resources a request uses, which are the anomaly requests that consume resources more than expected.

BorderPatrol does not require modules to be modified in order to trace them. Each module is treated as a black box. BorderPatrol uses active observation which carefully modifies the event stream observed by these modules, simplifying precise observation. It also uses protocol processors to leverage knowledge about standard protocols, avoiding application-specific instrumentation.

However, BorderPatrol has difficulties in constructing the correct request paths when the applications use internal work queues or user-level scheduling. Single process event-driven applications which use internal queues to buffer work units and multi-thread applications which also use internal work queues all fall into this category of applications which present problems to BorderPatrol.

BorderPatrol tries to follow requests in those applications which use internal work queues or user-level scheduling and does not know when it fails. Our contribution is improvements to BorderPatrol so that it does not only know when it fails to resolve request paths, but also tries its best to recover correct request paths. We use immediacy and message witnesses extensively to spot failures in path resolution and to recover correct paths. A catalog is built based on correct paths which BorderPatrol is confident about. The catalog is used later as a guidance to resolve those which BorderPatrol thinks confusing.

Our evaluation consists of case studies and an evaluation. We show that the improved version of BorderPatrol knows when it cannot resolve paths and how it tries to recover possible correct paths. Our experiments were performed on servers and proxies include MySQL, Lighttpd, and HAProxy, without modifications to server source code.

2 Background

BorderPatrol constructs causal paths of requests by observing inputs and outputs to and from modules which are considered as black boxes. The paths show which modules were used in which requests, what resources each request accessed, and how long it took a certain module to process the inputs. Although BorderPatrol treats modules as black boxes, it makes assumptions about how real-world applications work.

2.1 Black Box Model

Request traces can be categorized into two types, namely internal and external. Internal links are those which connects a module input to a module output. External links are those which connects a module output to the input of another module.

Only external links can be observed without modifying the source code of modules. BorderPatrol makes assumptions about internal links. It assumes the internal links must be *honest*, *immediate*, and *independent*. A black box module is honest if it faithfully implements the protocols it uses. Immediacy says that it should process the input event immediately after it is presented with. At last, a black box is independent if it handles concurrent requests the same way as it would if the requests arrived sequentially. All three assumptions must be true for each module so that BorderPatrol can construct the request paths correctly.

However, these assumptions are not necessarily true in all possible applications. BorderPatrol argues that most real-world applications do follow these assumptions. These assumptions make it possible for BorderPatrol to guess the internal request flow in each module.

2.2 Active Observation

In the hope of precisely tracing modules which follow the assumptions mentioned in the previous section, BorderPatrol uses *active observation* which consists of the following techniques.

Protocol processors adds knowledge of different protocols to active observation. They understand different protocols well enough to separate messages on a single channel. BorderPatrol has protocol processors for HTTP (1.0 & 1.1), FastCGI, PostgreSQL, X11 (client-side only), DNS (client-side only), One-shot, and Line-oriented protocols. ”One-shot” protocol processor processes protocols which include only one request/reply pair in a single TCP connection. The ”line-oriented’ protocol processor handles protocols which use newlines to delimit request/reply messages. *Message witnesses* are the data in input and output messages which can be extracted and used to match requests and replies. Since protocol processors are implemented using library interposition, they can process incoming data before forwarding to the black boxes, and process outgoing data before passing it on. *Event isolation* uses protocol processors to separate multiple requests into individual requests so that BorderPatrol can follow internal links without witnesses (due to immediacy).

Message witnesses are facts embedded in messages which bind them together. For example, a one-shot connection is identified by both of its end points. If two distinct messages have the same witnesses, they must be part of the same request. In other words, unlike the immediacy assumption, message witnesses are facts which can be trusted directly.

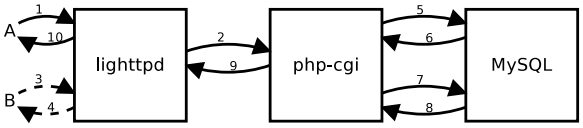


Figure 1: An example showing BorderPatrol traces. There are three black-box modules involved, an event-driven web server Lighttpd which uses php-cgi as the backend to handle PHP file requests, and a MySQL database. Two external requests **A** and **B** arrives at the web server one after the other. All messages are sent in the labeled order. Request **A**, shown in solid lines, is passed on to php-cgi to handle the PHP script which further accesses the database. Meanwhile, request **B**, which only reads a static file, arrives.

Figure 1 shows an example of BorderPatrol traces. The setup includes an event-driven web server Lighttpd which talks to php-cgi using FastCGI, and MySQL as the database. BorderPatrol uses library interposition and protocol processors to obtain resource accessing events of the three black-box modules. These traced events are transferred to a central log file. Once the events are recorded, they are sorted by time stamp. A joiner then correlates events by temporal joining them to construct paths.

Notice that in the traces, even though BorderPatrol cannot observe any of the internal links in the black boxes, it knows that message 2 belongs to request **A** because of immediacy. As soon as the application server module (php-cgi in this case) receives message 2, it initiates computation for request **A** immediately due to immediacy. Once the computation for request **A** is done in the database, the database sends a reply message back to the application server module. Since message 6 and message 5 use the same protocol, and message 6 is a reply to message 5’s request, they are matched up by witness. For the same reason, message 8 is matched up with message 7, message 9 with 2, message 10 with 1, and message 4 with message 3.

In order to handle concurrent requests, real-world applications multiplex requests one way or the other. In the example above, Lighttpd multiplexes requests by using an event-driven model. A single request is divided into smaller tasks. Each task can be done in a relatively short period of time. This way, Lighttpd can switch to requests which have tasks ready to be processed while the current task blocks.

There are several paradigms of multiplexing requests as presented by Pai *et al.* [3], namely multi-process or multi-threaded, single process event-driven, asymmetric multi-process event-driven, work queues, and user-level scheduling. While BorderPatrol works well on most of

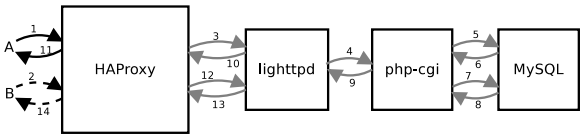


Figure 2: An example showing ambiguous paths. Request **A** is shown in solid lines, request **B** in dashed lines. Grey solid lines means that they can belong to either request. The requests are the same as those in Figure 1. But now HAProxy sits in front of the web server. HAProxy receives request **B** immediately following request **A**. Then HAProxy forwards one of the request to the web server first. After getting a reply from the web server, the other request is forwarded. Because of internal work queue, BorderPatrol does not know if the first request being forwarded to the web server is request **A** or request **B**.

them, it has problems constructing paths for applications that make use of internal work queues and user-level scheduling. In these two paradigms, requests might be queued up and scheduled in any order. Without understanding the internal links, immediacy has difficulties correlating the outputs of a black box to the requests.

A big drawback of BorderPatrol is that it does not know when any of the assumptions are violated. In other words, BorderPatrol is not self-aware. Although BorderPatrol works well on most applications, there are types of applications which do not follow the assumptions. When BorderPatrol runs on these applications, it does not know when it makes mistakes.

Applications which use internal work queues and user-level scheduling to multiplex requests are among those that violates the assumptions. HAProxy is a high availability, load balancing HTTP proxy. It allows users to specify a maximum number of outstanding requests in each web server backend. Once the maximum connection is reached, no more requests are forwarded to the backend until some or all of the existing ones finishes. Meanwhile, new requests are forwarded to other backends which have not saturated the maximum connections. If all backends have reached their maximum numbers of connections, HAProxy will push all new requests into its own internal work queue and wait until some backends become available. Once one or more backends become available, HAProxy will dequeue the requests in the work queue based on a specified scheduling algorithm and forward them to the web servers.

Figure 2 shows the same requests as in Figure 1 but with HAProxy added in front of the web server. Assume at the time request **A** arrives, all backends have reached the maximum numbers of outstanding requests in HAProxy. No backends become available until some time after request **B** comes. So both request **A** and **B** are queued up in HAProxy upon arrival. When the web server finishes processing the existing requests, HAProxy resumes forwarding requests again. In this case, immediacy tells BorderPatrol that message 3 belongs to request **B**, because it follows message 2's arrival immediately. Consequently, accesses to the application server and the database are all assigned to request **B**. For request **A**, BorderPatrol thinks it arrives at HAProxy and returns without interacting with any other modules. However, we know that request **B** only reads a static file. It does not require any CGI application, nor does it need to access the database. On the other hand, request **A** calls a PHP script which accesses the database.

3 Ambiguous Paths

In applications that make use of internal work queues and user-level scheduling, paths may be ambiguous without tracing the internal links. BorderPatrol did not have the capability of detecting such ambiguous paths. In order to resolve them, we need to identify the ambiguous paths first.

3.1 Identifying False Paths

Ambiguous paths are identified by observing immediacy and witness mismatches. When a mismatch takes place, it is likely that immediacy is wrong about what the internal links should be. Because immediacy is merely a guess of what might happened inside a module based on the assumptions mentioned in Section 2.1, it is less reliable. On the other hand, message witnesses are facts provided by messages themselves, which must be correct. The same example in Figure 2 shows such a mismatch.

Each pair of arrows in Figure 2 represents a request-reply pair in a protocol. So each reply message can be matched up with the corresponding request message by witness. BorderPatrol will process the events in the flow depicted in Figure 3. Notice that immediacy thinks message 11 should belong to request **B**, since the last time this module processed an event it belonged to request **B**. However, there is a witness relating message 11 to message 1, which marked the beginning of request **A**. This conflicts with what immediacy says. Hence, the paths which are currently being built are possibly wrong. BorderPatrol will go on and switch the currently processing request to **A**. Message 14 will then be considered belonging to request **A** by immediacy. But witness says differently because message 14 is a reply to message 2, which was believed to belong to request **B** at that moment.

With the capability of identifying false paths, BorderPatrol is able to detect its own mistakes and become self-aware. The makes the system more robust against different types of applications. Even in applications which BorderPatrol cannot handle, it will not generate false paths.

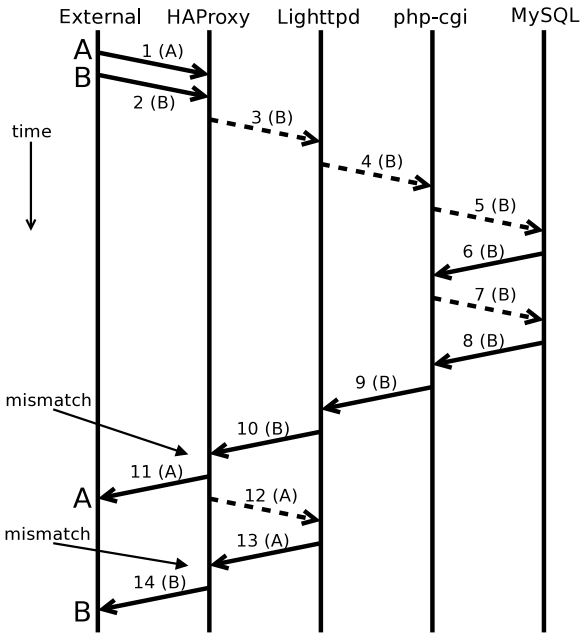


Figure 3: Process flow of events shown in Figure 2. Each vertical line represents a module. Time flows from the top to bottom. Each arrow represents a message sent, and it is labeled with the request ID of what BorderPatrol thinks it belong to. Messages linked to requests by immediacy are shown in dashed lines, and messages linked by witnesses are shown in solid lines. Two immediacy and message witness mismatches occur.

3.2 Backtracking

Once a false path is identified, BorderPatrol needs to know what portion of the path down to the point of mismatch is ambiguous. Starting from the mismatch event, BorderPatrol backtracks each event just processed until it is confident that an event belongs to a certain request. In Figure 3, immediacy thinks message 14 belongs to request **A**, and witness believes it belongs to request **B**. BorderPatrol stops backtracking when it reaches (*a*) an event belongs to request **B**, (*b*) the termination of a request, (*c*) an event which links to another event that happened before request **B** arrived by witness.

If BorderPatrol is certain that an event belongs to either request **B** or some other request, all events before it must be correct. The events between this breaking point and the mismatch event are the ones BorderPatrol is not confident about.

Stopping condition (a) is obvious that an event certainly belongs to **B**. Condition (b) says that an event can be linked with the beginning of a request by witness, such as message 11. So it is clear that the event certainly belongs to a request other than **B**. Condition (c) describes an event which can be confirmed by witness that it does not belong to request **B**, because it started before request **B** arrived. In our example, message 11 can be confirmed by witness that it is linked with message 1, which arrived before request **B** started. So backtracking ends here.

3.3 Ambiguity

Backtracking only tells BorderPatrol what fragment of a path is ambiguous. In order to find the correct path, BorderPatrol still needs to know what the ambiguous paths are by placing the ambiguous fragment differently.

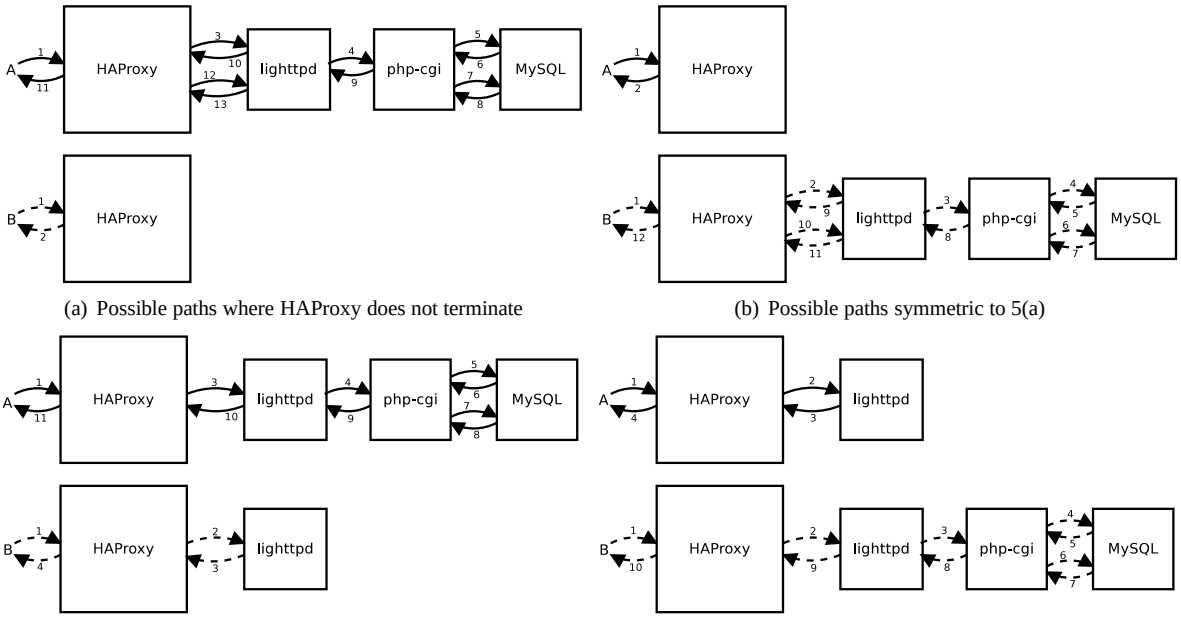
There are different types of feasible request paths in real-world applications, paths which are non-forking, paths which forks, paths which are non-terminating, and paths which terminates. Non-forking paths are the kind of paths which do not branch. They are linear. In contrast, forking paths have branches. They include black-box modules which fork a single input into multiple outputs. Non-terminating paths are paths which have branches left even after the request is being responded. On the other hand, terminating paths end when a request is being replied. Applications such as online shopping stores take orders and reply to customers immediately, then they process the orders in background. Notice that a non-forking path must also be terminating. Non-terminating path can occur if the path is non-forking.

Forking modules are not common. However, they do fit into some use cases. A forking module can execute tasks of a single request simultaneously. In contrast, modules in real-world applications tend to be linear. They execute one task of a request at a time. In a single request, subsequent tasks may depend on previous tasks. Requests may be multiplexed, but tasks in a single request are sequentially executed. For example, application server has to read in the CGI file before it can make decision whether or not to access database. In the modules we have studied, including HAProxy, Lighttpd, php-cgi, and MySQL, tasks in a request are processed linearly.

The example shown in Figure 2 can result in four different possible versions of paths. If request **A** accesses the application server and the database, it takes the long path. Depends on whether the load balancer black box is terminating or not, the last access to the web server (messages 12 and 13) may or may not belong to request **B**, shown in Figures 5(a) and 5(c), respectively. On the other hand, if request **B** is the one that accesses the application server and the database, request **A** takes the short path. Similarly, the last access to the web server also depends on the load balancer's termination property, see Figures 5(d) and 5(b). These paths are ambiguous to BorderPatrol because it has no knowledge of the internal properties of the black boxes.

There are different approaches to resolve ambiguous paths, including marking modules as forking or non-forking, marking modules as terminating or non-

Figure 4: Possible paths from example shown in Figure 2. They are produced by placing the ambiguous fragments in different requests. Notice that 5(a) and 5(b), and 5(c) and 5(d) are symmetric pairs. There are two ambiguous fragments in Figure 3, messages 3 to 10 and messages 12 to 13. Which request messages 12 and 13 belong to depends on the termination property of HAProxy.



(a) Possible paths where HAProxy does not terminate

(b) Possible paths symmetric to 5(a)

(c) Possible paths where HAProxy terminates

(d) Possible paths symmetric to 5(c)

terminating, using witness to match black box inputs and outputs, or comparing with past unambiguous paths.

(a) BorderPatrol uses message witness like file descriptor and network end points to identify replies to requests. A similar idea can be used. A *weak witness* is data extracted from both input and output protocol messages, such as URI in HTTP protocol. It requires that the messages are in similar protocols so that the request identifier can be extracted from both protocol messages, like HTTP and FastCGI protocols. If the output of a black box is in the protocol as the input, they can also be linked by weak witness. For example, proxies usually forward messages without changing protocol.

(b) Users of BorderPatrol can mark each module as forking or non-forking in a configuration file. When BorderPatrol resolves ambiguous paths, it will discard the possible paths which conflict with the forking property of modules. Besides the forking property, users can also tell BorderPatrol which modules terminate and which do not. So when BorderPatrol resolves ambiguous paths, it can discard those which include non-terminating paths if the modules are marked as terminating, such as Figures 5(c) and 5(b).

(c) Sometimes not all paths are ambiguous. Clear paths can be recorded and used as guidance in resolving ambiguous paths later. If all clear paths show that a module is terminating, it is likely that a non-terminating ambiguous path is **not** what really happened. Similarly, the forking property of modules can also be indicated by clear paths.

Approach (a) works well on modules which speak the same protocol in both input and output messages, and they are one-to-one correspondences. An example shows how this approach restricts the types of requests it can resolve. For an HTTP proxy, it is normal to get multiple requests to the same URL concurrently. In this case, all output messages from the proxy will have the same URL embedded, which makes it impossible to distinguish them. However, weak witnesses can still be useful in telling which outputs are **not** linked with an input.

Approaches (b) is easy to implement. However, it requires users have some knowledge about the modules in the applications to determine the properties. Often times, properties like forking and terminating are not stated clearly in the documentation of the modules. So the only way to find them out is to either ask an expert or read the source code, which is hard or even impossible to users who do not have access to these information.

We use approach (c) in BorderPatrol to resolve ambiguous paths. For each path which do not have any mismatch, it is simplified and stored in a catalog. The simplification shortens the path so that it only contains

the order of module occurrences. These simplified paths should indicate the forking and terminating properties of the modules. When BorderPatrol hits an ambiguous path, it compares all possibilities with the simplified paths in the catalog. The possible path that has an exact match in the catalog or more similar to paths in the catalog is likely to be the correct path.

4 Implementation

We added the capability of identifying ambiguous paths to the temporal joiner of BorderPatrol. Using this technique, BorderPatrol simplifies correct paths into the catalog. The catalog is then used as a guidance in resolving ambiguous paths.

4.1 Identifying Ambiguous Paths

The joiner is modified so that when a message can be confirmed by a witness, the result is compared with immediacy. If a mismatch happens, the normal operation pauses and BorderPatrol starts backtracking (see Section 3.2). Once backtracking is done, BorderPatrol resumes where it left off.

4.2 Building Catalog

Paths without mismatching immediacy and witnesses are correct. These paths are then simplified by only including module names in the order they appear in the paths. Simplified paths are added into a catalog for path resolution (See Section 4.3).

The way a single path is simplified is by only recording the module names associated with read events. If multiple contiguous read events occur, they are shortened and the module name is only recorded once. Take the actual path of request **A** in Figure 2 for example, the simplified path is shown in Table 1.

Process	Message	Simplified
HAProxy	Read request A	HAProxy
HAProxy	Write to Lighttpd	
Lighttpd	Read from HAProxy	Lighttpd
Lighttpd	Write to php-cgi	
php-cgi	Read from Lighttpd	php-cgi
php-cgi	Write to MySQL	
MySQL	Read from php-cgi	MySQL
MySQL	Write to php-cgi	
php-cgi	Read from MySQL	php-cgi
php-cgi	Write to MySQL	
MySQL	Read from php-cgi	MySQL
MySQL	Write to php-cgi	
php-cgi	Read from MySQL	php-cgi
php-cgi	Write to Lighttpd	
Lighttpd	Read from php-cgi	Lighttpd
Lighttpd	Write to HAProxy	
HAProxy	Read from Lighttpd	HAProxy
HAProxy	Reply to request A	

Table 1: Simplified path of request **A** in Figure 2. The simplified path is [HAProxy, Lighttpd, php-cgi, MySQL, php-cgi, MySQL, php-cgi, Lighttpd, HAProxy].

4.3 Resolving Paths

To recover all possible paths, we use *backtracking* (see Section 3.2) when a mismatch is identified. It takes the request ID given by witness as the new ID and backtracks the events before the mismatching event, adding the new ID to the list of request IDs they possibly belong to, until any one of the stopping conditions is satisfied.

All events being backtracked at the same time fall into the same fragment. They are assigned the same set of request IDs, meaning that they might belong to any one of the requests as a batch, but not more than one. All possible sets of paths are produced by including these fragments in different requests they were assigned to one at a time.

Each possible path is then simplified using the same procedure discussed in Section 4.2. The simplified path is compared to the ones in the catalog. A score is calculated based on the smallest difference between the simplified path and the ones in the catalog. Difference is recorded as negative value. If there is an exact match in the catalog, a high positive value is given. The score is assigned to the path fragments in the current possible path as weights.

After calculating the weights of all path fragments, all possible paths are presented to the user with path fragments colored based on their weights. This can guide the user in determining the correct paths.

4.4 MySQL Protocol Processor

The only database protocol BorderPatrol supported was PostgreSQL. We added MySQL protocol processor because it is widely deployed and popular among web applications. It is implemented in 156 lines of code in total for all functions. The MySQL protocol processor also logs SQL queries to the corresponding events. They can be used to show which requests account for what queries.

4.5 Multi-machine Support

Although BorderPatrol was designed with multi-machine support in mind, it did not have all the proper pieces in place. To make it work across network on multiple machines, we made the following changes.

In the original BorderPatrol paper [2], the logging daemon only accepted local unix domain sockets for connections. This obviously does not work across network. We modified the logging daemon and the tracing library to use TCP connections instead.

BorderPatrol was using kernel cycle count as time stamp. It worked well on a single machine with high precision. However, it did not work on multiple machines, since kernel cycle count is different on each machine. Depends on the architecture of the systems, 32-bit or 64-bit, the precisions of the cycle counts are also different. Some machines may have power management features which will scale CPU speed down when not needed. This changes the cycle count as well. So we modified the logging daemon and the tracing library to use the time of day given by **gettimeofday()** as time stamp. The clocks do not need to be synchronized precisely. However, the differences have to be within a small constant. In order to accommodate one-way network delays, we use a simple method to adjust the time stamps of the events received at the logging daemon. The logging daemon records the time differences between each machine and the machine it runs on based on the first event from them. Then it applies the corresponding recorded difference to all events from each machine.

Applications that span multiple machines use TCP or UDP sockets for module communication. In order to recognize what protocol a module uses to talk to another module, we use the same approach as the Internet network service list. When a connection establishes, both the host end port and the destination end port are searched in */etc/services*. If the user wants to specify custom service ports, a custom service-port mapping file can be provided to BorderPatrol. The custom mappings will override those in */etc/services*.

4.6 Dark Corners of the Linux API

Most real-world applications use non-blocking system calls in modules, especially event-driven applications. Non-blocking connect() returns immediately with error code EINPROGRESS to indicate that the connection is in progress. Then the programmer has to call poll() or select() on the socket file descriptor to check if the connection succeeded. On the destination end, accept() may return before the host calls poll()/select(). This confuses BorderPatrol as if accept() happened before connect() takes place, because BorderPatrol records the CONNECT event at the time of success. We modified the interposed connect() and poll()/select() so that when connect() returns EINPROGRESS, it records a CONNECT_DELAYED event. Later when poll()/select() returns the file descriptor, it is recorded as the actual CONNECT event with the time stamp of the corresponding CONNECT_DELAYED event. This guarantees that the CONNECT event appears before the ACCEPT event to the joiner.

BorderPatrol greatly relies on the effectiveness of library interception. Modules which support plugins usually use dynamic linking loader to load them into memory at run-time. On Linux, the flag RTLD_DEEPBIND to dlopen() puts the symbols in the loaded plugins ahead of the global scope, making them untraceable to BorderPatrol. As a result, we have to interpose on dlopen() to remove RTLD_DEEPBIND from the flag argument.

Some applications, such as Lighttpd, use ioctl() to check if a file descriptor has data ready to be read. If the file descriptor has data ready, the application calls read() and compares the bytes read with what ioctl() reported. This did not work in BorderPatrol because protocol processors may decide to isolate multiple messages in a single packet and present one message at a time. We interposed on ioctl() to report the exact number of bytes which will be returned by the interposed read().

5 Evaluation

The evaluation is divided into two subsections. The first subsection shows the correctness and effectiveness of ambiguous path resolution. The second section shows the overhead of the tracing library under realistic workload on real-world deployment. Our experiments were conducted on two servers, one running 2.2GHz Athlon Dual Core CPU and 2GB of RAM, and the other one running two 2.8GHz Pentium 4 CPUs and 1GB of RAM. The servers are of different architectures, 32-bit on the faster machine and 64-bit on the slower one. MySQL database was run on the 32-bit machine, with the logging daemon running locally. Lighttpd and HAProxy were both run on the other machine, sharing the logging daemon with MySQL. All modules communicated through TCP connections.

We worked with the maintainers of passiveaggressivenotes.comso that we set up a similar environment as their website. All modules were configured using the configuration file they provided. Lighttpd was set up to accept 2048 simultaneous connections and with 16 php-cgi backends. They were not using HAProxy as load balancer, we added HAProxy in front of the web server because it uses internal work queues. A single services file was provided with custom port numbers included for the protocols. The blog platform they used was wordpress. We installed wordpress 2.8.4 and submitted several simple posts. The deployment is similar to that shown in Figure 2, but with more php-cgi modules.

5.1 Ambiguous Path Resolution

The experiment was conducted on application set up the same as in real-world deployment. All modules were traced using BorderPatrol tracing library. We took the log file of passiveaggressivenotes.comand replayed 20 seconds of the requests. The 20-second request fragment was extracted from the access log of 3pm of a Monday. There were 402 requests in the fragment, mostly static file requests. 28 of the total requests were to PHP scripts.

All requests were sent out from a dedicated thread, so that no request wait for the other to finish. Although in realistic scenarios, static files in a single page always follow the request to the page. The static files themselves may or may not be requested concurrently depending on the behavior of the browsers. Since the precision of the time stamps in the log file is second, we made requests with the same time stamp to happen at random time within the second.

Stretch factor	0.5	1	2
Total requests	402	402	402
Clear paths	172	91	299

Table 2: Results of replaying realistic requests from passiveaggressivenotes.comwith different stretch factors. There were 402 requests in total, mostly static file requests. Clear paths show the number of requests with no ambiguous paths. These paths were simplified and inserted into the catalog.

The result of the experiment is shown in Table 2. We replayed the same requests in the same order with different stretch factors, twice the normal speed, normal speed, and half the normal speed. The clear paths did not have ambiguous fragments. They were simplified and inserted into the catalog. We were able to obtain the following four types of paths in the catalog in all three experiments. The first type is requests to static files, which only involve HAProxy and Lighttpd. The second type is requests to PHP scripts which do not access the database. It includes php-cgi in addition to HAProxy and Lighttpd. The third type is requests to PHP scripts which also access the database, so MySQL is also involved. The last type is requests which were rejected by HAProxy due to overload. The simplified paths of all four types are shown in Table 3. These four types of paths cover all types of scenarios the blog can produce. All four types of paths indicate that all modules are terminating.

The joiner was run twice to resolve the paths. The first run found all clear paths and insert them into the catalog. The second run was to resolve the ambiguous paths using the catalog. Each run took about half an hour to finish.

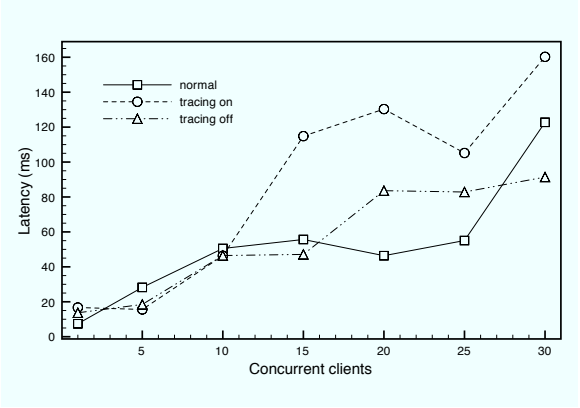


Figure 5: Latency overhead for realistic workload. The workload was requests to a wordpress blog. The benchmarks were run without BorderPatrol tracing library, with the tracing library turned on and off at run-time.

5.2 Overhead Benchmarks

The tracing library of BorderPatrol can be turned off at run-time. We now show the run-time overhead of the tracing library on a real-world deployment. The same set up is used in the benchmarks. We used the same 20-second requests extracted from the access log of passiveaggressivenotes.com. However, requests were not replayed. They were randomly chosen from the same list of URLs and sent out as soon as possible by closed-loop feedback clients. So there were more static file requests than CGI requests.

The result of the benchmarks is shown in Figure 5. The concurrent clients were run on the local machine to reduce network latency. Note that MySQL forks a new process each time a new client connects to it. Once the static files were read, they were loaded into memory and subsequent accesses to the same files represent overhead of making system calls. With BorderPatrol tracing library turned on, the mean overhead was 60.97%. With BorderPatrol tracing library loaded but turned off at run-time, the overall latency overhead was 4.80%.

6 Conclusions

We improved BorderPatrol to be able to work on real-world deployments. We also modified BorderPatrol so that it knows when it makes mistakes. Beyond this, we also showed possible ways of solving the ambiguous paths when mistakes are made. We implemented one of the ambiguity resolution algorithms which requires no user interruption in BorderPatrol so that it can handle applications which make use of internal work queues and user-level scheduling.

BorderPatrol with all the improvements and the ambiguity resolution algorithm is available from

Static	haproxy, lighttpd, haproxy
PHP w/o database	haproxy, lighttpd, php-cgi, lighttpd, haproxy, lighttpd, haproxy, php-cgi
PHP with database	haproxy, lighttpd, php-cgi, mysqld, ..., php-cgi, lighttpd, haproxy, php-cgi
Rejected	haproxy

Table 3: Four types of paths in the catalog built from the clear paths.

`http://cs.brown.edu/research/borderpatrol/`.

Acknowledgements

The authors would like to thank Eric Koskine for the help at the initial stage of the project, and C. Chris Erway for the kindness of providing configuration files of passiveaggressivenotes.com.

References

[1] HAProxy - event-driven http load balancer. `http://haproxy.1wt.eu/`.

[2] Eric Koskinen and John Jannotti. Borderpatrol: isolating events for black-box tracing. *SIGOPS Oper. Syst. Rev.*, 42(4):191–203, 2008.

[3] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server.