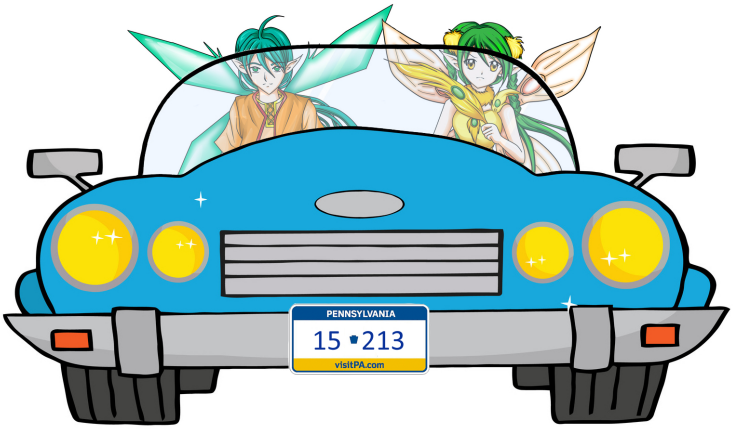

PXYDRIVE: A Proxy Testing Framework*

Randal E. Bryant
David R. O'Hallaron
Carnegie Mellon University

April 6, 2019

1 Overview

PXYDRIVE (pronounced "pixie drive") provides a comprehensive testing framework for web proxy programs. It enables testing and debugging of the basic operations of a proxy, as well as the ability to handle unexpected events, to support concurrent requests, to provide caching of retrieved files, and to avoid race conditions. An associated program PXYREGRESS runs a series of standard tests, using PXYDRIVE to perform each test. A second associated program PXYRACE runs a single test multiple times, with random delays inserted into the synchronization operations, in an attempt to expose race conditions. These three programs are designed to help students implement their proxies and to help graders perform automated testing. PXYDRIVE was developed to support the ProxyLab assignment, but it can be used on other proxies, as well.

Figure 1 illustrates the overall structure of PXYDRIVE. As shown, it includes a number of components:

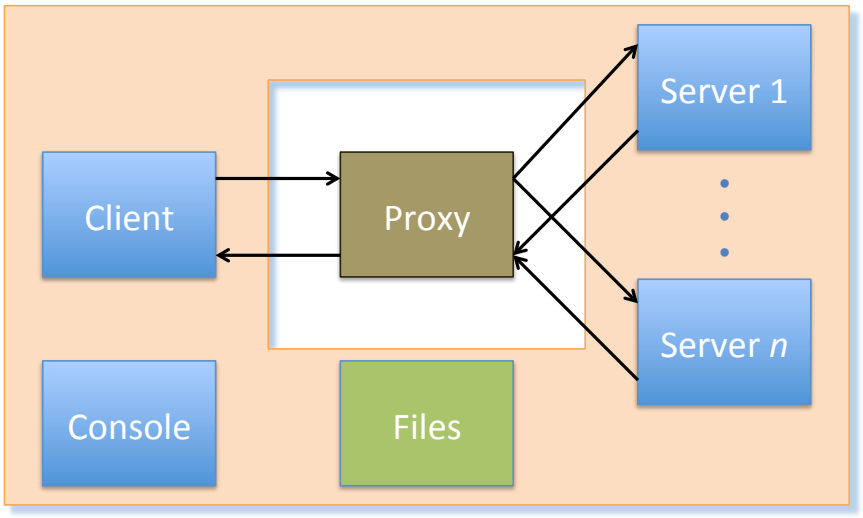

Figure 1: **Structure of PXYDRIVE** The program acts as both the client and the servers for a proxy program.

- An executing *proxy* program. This can either be an executable program run as a child of PXYDRIVE (an "internal" proxy), or it can be one invoked separately (an "external" proxy.)
- A web *client* that issues requests for files and compares the responses to original copies of the files.
- One or more *servers* that respond to requests for files
- A set of *files* shared by the client and the servers
- A *console* providing a simple command-line interpreter to perform tests and to observe results.

This structure provides a number useful features for testing a proxy program:

- By including all of these components in a single program, PXYDRIVE has very detailed control over the sequencing of interactions with the proxy.
- By having files shared between the client and the servers, the client can check that the results it receives match those sent by the servers.
- By observing a transaction from both the client's and the server's side, it can provide useful information to the user about all of the communications that take place.
- The ability to observe both server and client communications enables PXYDRIVE to check that the proxy complies with any formatting requirements for HTTP messages.

2 Running PXYDRIVE

PXYDRIVE is written in Python, and so is invoked as `./pxydrive.py` from its local directory. (You can also run it from another directory by giving an appropriate path.) PXYDRIVE has been tested on both Linux and Mac OS X. Its ability to run on other platforms is unknown.

2.1 Command-Line Arguments

PXYDRIVE supports the following command-line arguments:

`-h`
Print usage information

`-p` *PROXY*
Run specified proxy program internally. For more details, consult the documentation of the `proxy` command.

`-P` *HOST:PORT*
Use the proxy running externally on the specified host and port. For more details, consult the documentation of the `external` command.

`-f` *FILE*
Read commands from *FILE*. For more details, consult the documentation of the `source` command.

`-l` *FILE*
Write inputs and outputs to *FILE*. For more details, consult the documentation of the `log` command.

`-c (0-4)`
Run the proxy with different levels of checking for concurrency errors enabled. See Section 6.

`-d` *STRETCH*
Adjust all delays and timeouts by factor of $STRETCH/100.0$. For more details, consult the documentation of the `stretch` option.

PXYDRIVE creates two subdirectories of the directory in which it is invoked: `source_files`, in which generated files are stored, and `response_files`, in which files received by the client are stored. These files can be examined when some transaction generates an error. Old versions of these files are deleted every time PXYDRIVE starts.

Once started, the user directs PXYDRIVE to perform a series of tests, either by typing commands directly to the program or by reading the commands from a file.

2.2 Proxy Requirements

PXYDRIVE is designed to test proxy programs that meet the requirements specified in ProxyLab. The lab description specifies several important aspects of the request headers sent by the proxy to a server:

- There must be a `Host` header
- There must be `User-Agent`, `Connection` and `Proxy-Connection` headers with specific values.
- Any other request headers that are sent by the client should be forwarded unchanged.

The latter requirement is especially important, because PXYDRIVE uses information passed in headers to track the interactions between clients and servers.

PXYDRIVE enforces rules on the formatting of request message headers with different levels of strictness. The level, ranging from 0 to 4, can be set using command-line option `-S` or with the `strict` option. Beyond level 0, each level applies the following additional checks:

1: All headers are terminated properly, with both the carriage return ('`\r`') and the linefeed ('`\n`') characters. Each header must contain a key and value, separated by a colon ('`:`').

2: The GET request must be formatted correctly, and there must be a Host header.

3: The headers used by PXYDRIVE (included in the request from the client) must be present.

4: The `User-Agent`, `Connection` and `Proxy-Connection` headers must be present.

Error messages will be printed for request messages that do not comply. When running with strictness level less than three, PXYDRIVE does not require that the headers it provides from the client be included in the request to the server. Instead, it does its best to track the interactions heuristically. Even if this level of strictness is not enforced, the program works best when the proxy forwards information placed in the request headers to the server.

3 PXYDRIVE Operation

A *transaction* by PXYDRIVE consists of all of the steps required for the client to get a file from a server via the proxy and ascertain its correctness. A transaction has two different phases: the *request* by the client, followed by the *response* by the server. Each transaction can be divided into six steps—three for each phase, as is illustrated in Figure 2:

Request

1. The client sends a `GET` request to the proxy. This request includes a URL, identifying the host server and the file name.
2. The proxy forwards the request to the designated server.
3. The server receives the request, locates the file, and formulates a response message.

Response

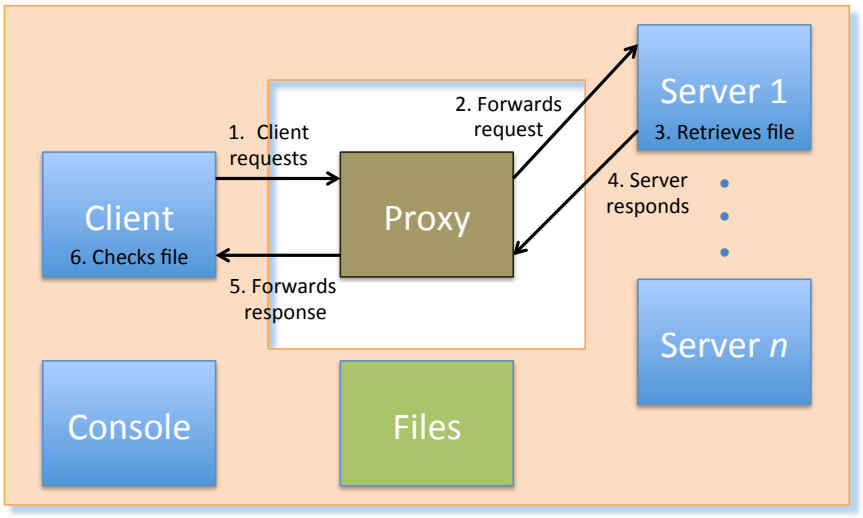

Figure 2: **Steps in a transaction** A transaction involves a *request* (steps 1–3) and a *response* (steps 4–6)

```
1 serve s1                          # Set up server
2 generate random-text.txt 10K      # Generate text file
3 request r1 random-text.txt s1     # Initiate request r1 from server s1
4 wait *                            # Wait for request to propagate
5 respond r1                        # Allow server to respond
6 wait *                            # Wait for response to propagate
7 check r1                          # Compare retrieved file to original
```

Figure 3: **Using PXYDRIVE to perform transaction with separate request and response.** The `wait` commands are included to make the test program wait while the proxy, server, and network operate.

4. The server sends the response message to the proxy.
5. The proxy forwards the response to the client.
6. The client receives the requested file and compares it to the original one.

PXYDRIVE gives fine-grained control over transactions, providing separate commands to trigger the request phase and the response phase. This feature makes it possible to test important features of the proxy, including its ability to handle concurrent transactions and to cache previously retrieved files.

The following simple examples illustrate how the user can control PXYDRIVE operation. These examples all consist of commands that can be supplied to the command-line interface. Observe that '`#`' is a comment character—it and anything to the right on the same line are ignored.

Figure 3 shows a session having a single transaction, with separate commands to trigger the request and the response. (For this session, the proxy was specified as a command-line option.) The lines are as follows:

1. A server is set up and given name `s1`. (It is possible to have multiple servers, each with a unique name.)

```
1 serve s1                          # Set up server
2 generate random-data.bin 10K      # Generate binary file
3 fetch f1 random-data.bin s1       # Fetch file
4 wait *                            # Wait for request and response
5 check f1                          # Compare retrieved file to original
```

Figure 4: **Using PXYDRIVE to perform transaction with immediate response.** Only one `wait` command is required.

2. A file, consisting of 10,000 random ASCII characters is generated. (The extension `.txt` indicates that the file should contain only ASCII characters.) The suffix '`K`' indicates that the number of bytes should be multiplied by 1,000.

3. The client makes a request for this file from the server. The transaction is given the identifier '`r1`.' Transaction identifiers must be unique, because they provide the mechanism by which PXYDRIVE tracks the progress of transactions.

4. The program waits for the request to propagate to the server, allowing steps 1–3 of the transaction to complete. The notation `wait *` specifies that all outstanding actions should complete. Alternatively, we could have used the command `wait r1` to specifically wait for request `r1`. Waiting via the `wait` command must be included when operating in batch mode whenever the proxy or server must complete some operation(s) before the next command can be performed.

5. The server is instructed to respond to the request. Note the use of the transaction identifier to indicate the specific request.

6. The program waits for the response to propagate to the client, allowing steps 4–6 of the transaction to complete. Again, we could have used the command `wait r1` to specifically wait for response `r1`.

7. The program checks that the file was received correctly by the client. An error message will be printed if either the server could not find the file, the file was corrupted, or the client did not receive the response.

Figure 4 also performs a single transaction, but it illustrates several additional features of PXYDRIVE. In line 2, the generated file is given the extension `.bin`. This causes the generated file to contain arbitrary bytes. Such a file will test the ability of the proxy to properly handle binary data. The `fetch` command of line 3 indicates that PXYDRIVE should perform a complete transaction, with the server responding immediately to the request. Only one `wait` command (line 4) is required, since all six steps of the transaction will take place due to the `fetch` command. Running a session consisting of a mix of fetches and requests requires the proxy to handle a variety of event timings.

Figure 5 illustrates how PXYDRIVE can be used to test the ability of the proxy to handle transactions involving exceptional conditions. It shows an attempt to retrieve a file that does not exist. The server will respond to this request with a message having status code 404, indicating that the file was not found. The proxy should forward this message to the client. The optional argument `404` on line 4 indicates that the program should check for this status code in the response to the client.

```
1 serve s1                        # Set up server
2 fetch f1 nonexistent.file s1    # Fetch file
3 wait *                          # Wait for request and response
4 check f1 404                    # Check for not-found status
```

Figure 5: **Using PXYDRIVE to perform transaction with missing file.** The check confirms status code 404.

These three examples illustrate some of the capabilities of PXYDRIVE to run tests of a proxy. Section 5 provides additional examples showing how to test different features of a proxy.

4 Commands

The following documents all of the commands supported by PXYDRIVE. These are divided into different categories, according to when and how they get used. Commands labeled as "seldom used" are ones that you are unlikely to use in writing your own tests.

In the following, square brackets enclose optional arguments, while the notation *ARG*+ means one or more repetitions of argument *ARG*. The notation (*ARGA*|*ARGB*) indicates a choice between arguments *ARGA* and *ARGB*.

4.1 Set Up

There are two ways to indicate which proxy to run. With an *internal* proxy, the user provides the path name to an executable proxy program. PXYDRIVE executes this program as a child process. With an *external* proxy, the user starts the proxy separately and indicates to PXYDRIVE the identity of the host name and port on which the proxy is running. The following commands are used at the beginning of a session to establish the environment for a set of transactions.

`proxy` *PATH* **[*ARG*+]**

Execute proxy program stored as executable file *PATH*. Any remaining arguments are passed to the proxy as command-line arguments. (There is no need to pass a port number—this is supplied by PXYDRIVE.) If an internal proxy is already running, it will be terminated before starting the specified proxy.

Any results printed by the proxy to either standard output or standard error will be echoed to the PXYDRIVE standard output and log files. **Note:** On many machines, you will find that printing debugging information on standard error works better. These get printed immediately. When printing on standard output, you may need to include calls to `fflush()` to force the output to appear.

`external` *HOST:PORT*

Use the proxy already running with the specified host and port. The proxy can be running on a different machine.

serve *SID*+

Set up server(s) with named identifier(s). These will be assigned ports chosen by PXYDRIVE. Server identifiers must be unique.

When *SID* starts with a hyphen ('−') character, the server will be set up but not activated. This will cause connection attempts by the proxy to be refused. Having an inactive server provides a way to test the proxy's ability to deal with URLs containing invalid host addresses. The proxy will not be able to perform the transaction, but it should still be able to process other transactions.

generate *NAME*`.`(**txt**|**bin**) *BYTES*

Generate file with random content. Files with names of the form *NAME*`.txt` will contain random ASCII characters, formatted into lines of length 80. Files with names of the form *NAME*`.bin` will contain random, unformatted bytes. Argument *BYTES* indicates the number of bytes in the file. As a shortcut, suffix '`K`' multiplies the number of bytes by 1,000, and suffix '`M`' multiplies the number of bytes by 1,000,000.

4.2 Transactions

The following commands control the progress of transactions and observe their results. Some are labeled "seldom used" to indicate that they only occur in specialized tests of corner conditions.

get *URL*

Retrieve the web object specified by the URL. One copy will be retrieved via the proxy and one will be retrieved directly. If the two copies are not identical, an error message will be generated. The URL must start with '`http://`,' but it can be to any web server.

fetch *ID FILE SID*

Fetch file from server (i.e., the request will be passed to server, and the server will be allowed to respond immediately.) *ID* is an identifying name for the transaction. It must be unique. Unless testing for a "not found" response, *FILE* should be one of the files accessible to the server (e.g., one generated with the `generate` command). *SID* identifies which server to use.

request *ID FILE SID*

Request file from server (i.e., the request will be passed to server, but the server will be not be allowed to respond.) *ID* is an identifying name for the transaction. It must be unique. Unless testing for a "not found" response, *FILE* should be one of the files accessible to the server (e.g., one generated with the `generate` command). *SID* identifies which server to use.

respond *ID*+

Allow the server(s) to respond to request(s) specified by the identifier(s). An error message will be printed if any of these requests has not been received by its server.

wait (***** | *ID*+)

Wait until the specified requests, responses, and fetches have completed. Explicit waiting is required when running in batch mode to make sure the client, proxy, and servers have time to complete their activities. The argument '`*`' indicates that the program should wait until all outstanding operations

have completed. If the specified actions do not complete within the timeout limit (determined by the `timeout` and `stretch` options), an error message will be generated.

`check` *ID* [*CODE*]
Check that transaction *ID* generated the proper status code. The default code is 200, indicating that the file was successfully retrieved. Additional status codes include 400 (bad request), 404 (not found), 501 (not implemented), and 505 (unsupported HTTP version), as are documented in Figure 11.25 of CS:APP.

`trace` *ID*+
Trace the histories of the specified transactions. This will print information about the six transaction steps illustrated in Figure 2, including copies of the HTTP headers. This command can be very useful when debugging a proxy or a test sequence.

`disrupt` (`request`|`response`) [*SID*]
(**Seldom used**) Disrupt the next request or response message between the proxy and either 1) the client (default) or 2) the server (indicated by server ID *SID*). This command allows testing the proxy's ability to handle an unexpected failure in its connection to the client or server. Referring to Figure 2, the possible disruptions cause a failure at step 1 (request/client), step 2 (request/server), step 4 (response/server), or step 5 (response/client). When a disruption occurs, the proxy will not be able to complete the transaction, but it should be able to continue processing other transactions.

4.3 Runtime Control

The following commands control the operation of the program.

`log` *FILE*
Copy outputs to *FILE*. Inputs are also echoed to the file if the `echo` option is set. (This is the default.)

`source` *FILE*
Read commands from *FILE*.

`option` *OPTION VALUE*
Set a runtime option. Supported options are documented below.

`delete` *FILE*
Delete file *FILE*. Files generated during previous runs of PXYDRIVE are automatically deleted every time the program starts, and so explicit deletion of files is only required for running special tests, or to avoid leaving large files around after a test has completed.

`help`
Print a summary of the commands and options.

`quit`
Exit the program.

`delay` *MS*
(**Seldom used**) Delay for *MS* milliseconds. This command can be used to allow the client, proxy, and servers to complete any outstanding requests, fetches have completed. It is an alternative (generally less desirable) to using the `wait` command.

`signal` [*SIGNO*]
(**Seldom used**) Send a signal with numeric value *SIGNO* to the proxy process. The default value is 13, corresponding to signal type SIGPIPE. This provides a way to test the proxy's ability to handle unexpected signals.

4.4 Options

The following options are supported. For Boolean options, 1 indicates true, and 0 indicates false.

autotrace (Boolean, default = 0)
Trace every request for which a check fails. This mode is helpful for debugging, because it causes detailed information about each failed check to be printed.

echo (Boolean, default = 1)
Echo inputs to log files. This makes it possible to keep track of which commands were being executed.

error (Integer, default = 5)
The maximum number of errors that can occur before the program aborts.

linefeed (Percent, default = 5)
The fraction of bytes in a randomly generated binary file having value `0xA` (linefeed character). The presence of lots of line-feeds will slow the response of a proxy that uses line-oriented reading and writing when receiving and sending binary data.

locking (Boolean, default = 0)
When enabled, the **locking** concurrency check will be performed, as documented in Section 6.

quiet (Boolean, default = 0)
Minimize the output generated by the program.

semaphore (Boolean, default = 0)
When enabled, the **semaphore** concurrency check will be performed, as documented in Section 6.

stretch (Integer, default = 100):
Adjusts the length of each delay and of the timeout for the `wait` command. Given a command "**delay** d," and given a stretch value of s, the actual delay will be $d \cdot s/100$ milliseconds. Setting the stretch to a value greater than 100 lengthens all delays, while setting it to a value less than 100 shortens them. Similarly, given a nominal timeout value of t, the actual timeout will be $t \cdot s/100$ milliseconds. This provides a way to adjust all of the delays and the timeout in a test sequence without editing the delay commands in a file or setting the timeout option. It can be useful in detecting cases where tests are just on a border between success and failure due to event timings.

```
1 serve s1
2 generate random-text1.txt 2K     # File 1
3 generate random-text2.txt 4K     # File 2
4 request r1 random-text1.txt s1   # Request r1
5 request r2 random-text2.txt s1   # Request r2
6 wait *
7 respond r2                       # Respond out of order
8 wait *
9 check r2
```

Figure 6: **Using PXYDRIVE to perform transaction requiring concurrent proxy.** A serial proxy would be held up waiting for the server to respond to request `r1`.

strict (Integer, range 0–4, default = 0):
Apply specified level of strictness to the formatting rules of the HTTP request headers. The different levels are described in Section 2.2.

timeout (Integer, default = 5000):
Sets the timeout used by the `wait` command. If the specified actions do not complete within the designated time, an error message will be generated.

timing (Boolean, default = 0)
When enabled, the **timing** concurrency check will be performed, as documented in Section 6.

unsafe (Boolean, default = 0)
When enabled, the **unsafe** concurrency check will be performed, as documented in Section 6.

verbose (Integer, default = 0)
Sets the amount of detailed reporting generated. Setting it to a nonzero value will cause more information to be generated, but the results generated by multiple threads will be interleaved arbitrarily, making it difficult to interpret the results.

5 Testing Proxy Capabilities

The following are some examples demonstrating how the fine-grained control provided by PXYDRIVE allows testing of different features of a proxy. Additional examples can be found in the standard tests described in Section 8.

Figure 6 illustrates the ability of PXYDRIVE to ascertain that a proxy supports concurrent transactions. In this test, two requests—`r1` and `r2`—are made (lines 4–5), but the server only responds to `r2` (line 7). A sequential proxy will hang up waiting for the response for `r1`, never forwarding `r2` to the server. The `wait` command on line 6 will therefore fail, because request `r2` never completes. A concurrent proxy can handle multiple transactions simultaneously, and so it will forward `r2` to the server and send the response back to the client, even while waiting for the server to respond to `r1`

```
1 serve s1
2 generate random-text.txt 10K
3 fetch f1 random-text.txt s1    # Handled by server
4 wait *
5 check f1
6 request r1 random-text.txt s1  # Handled by proxy
7 wait *
8 check r1                       # No action required by server
```

Figure 7: **Using PXYDRIVE to perform transaction requiring caching.** A caching proxy will respond to `r1` without any action by the server.

Figure 7 illustrates the ability of PXYDRIVE to ascertain that the proxy caches previous results. It first performs transaction `f1`, retrieving file `random-text.txt`. It then issues a request for the same file (transaction `r1`). If the proxy has the file it its cache, it will short circuit the transaction and return the file directly to the client, skipping steps 2–4 of Figure 2. The server never receives the request and therefore need not be instructed to respond to it.

6 Concurrency Checking

PXYDRIVE can apply enhanced checking to increase its ability to detect several classes of concurrency errors. First, we cover the case where PXYDRIVE uses an internal proxy.

The checks are selectively enabled either by using the command line option `-c` with a value between 1 and 4, or by selecting the individual checks via the `option` command. (In the latter case, these options must be set before the `proxy` command is given.) PXYDRIVE performs these checks by using *run-time interpositioning* to modify the calls to system functions, including synchronization operations and file reads and writes. (Runtime interpositioning is described in slides 56–58 of the lecture on linking.)

The following checks can be performed. In the following, let L be the checking level specified with the `-c` command-line option.

timing: Random delays are inserted before synchronization operations, as well as socket reads and writes. The resulting set of event orderings will vary over the different tests, enhancing the detection of race conditions. This is not a foolproof way to detect races, but it can help. Enabled when $L \geq 1$.

unsafe: Warning messages are printed when the proxy calls a function that is known to be thread-unsafe. This does not necessarily indicate an error. For example, such calls are safe if the program uses a lock to ensure that they are mutually exclusive. Enabled when $L \geq 2$.

locking: Checks are performed before each read or write operation to a socket. If the thread performing the operation is also holding a lock, this is considered a violation of the required locking protocol. The read or write operation will fail, returning value -1. Enabled when $L \geq 3$.

semaphore: Calls to semaphore operations will be detected and fail, since semaphores should not be used in your proxy implementation. Enabled when $L \geq 4$.

A). Running proxy under GDB

```
[catshark]$ gdb ./proxy
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-110.el7
(gdb) break open_clientfd
Breakpoint 1 at 0x40481c: file csapp.c, line 955.
(gdb) run 15213
```

B). Running PXYDRIVE with external proxy

```
[carpetshark]$ ./pxydrive.py -P catshark.ics.cs.cmu.edu:15213
Using proxy at catshark.ics.cs.cmu.edu:15213
>generate r1.txt 1k
>serve s1
Server s1 running at carpetshark.ics.cs.cmu.edu:30056
>fetch f1 r1.txt s1
Client: Fetching '/r1.txt' from carpetshark.ics.cs.cmu.edu:30056
```

Figure 8: **Session demonstrating proxy running under GDB.** Parts A and B show sessions running on different machines.

The concurrency checks can also be performed while running an external proxy, although it's a bit tricky. See Appendix A.

7 Useful Debugging Techniques

This section describes ways you can use some popular debugging tools with a proxy program in conjunction with PXYDRIVE.

7.1 Running under GDB

The Gnu Debugger (GDB) enables the user to interactively control a program and observe the program state. To test a proxy running under GDB with PXYDRIVE, you can set it up as an external proxy. Figure 8 shows an example session running GDB on one machine (A) and PXYDRIVE on another (B). As the example shows, the user can set breakpoints and examine the state of the program with GDB. By indicating to PXYDRIVE the location (host and port) of the proxy, the proxy will be driven by requests made by PXYDRIVE to server(s) operated by PXYDRIVE.

7.2 Running under VALGRIND

VALGRIND is a powerful tool for debugging and profiling programs. It is especially helpful for identifying memory referencing bugs and memory leaks. (See `www.valgrind.org`.) PXYDRIVE can use either an external or an internal proxy running under VALGRIND. The following command sequence shows how to set up a test with an internal proxy:

```
[carpetshark]$ ./pxydrive.py
>proxy valgrind --leak-check=yes ./proxy
>source ../tests/A01-simple-fetch.cmd
```

Observe that we tell PXYDRIVE to execute the program `valgrind`, but we also pass the command-line arguments that cause VALGRIND to invoke the proxy program.

Running VALGRIND with an external proxy is similar to running the proxy under GDB, as shown in Figure 8A. Simply start the proxy under VALGRIND and indicate to PXYDRIVE the host and port number it is using.

8 Standard Tests

The directory `tests` contains a set of 51 command files testing many aspects of a proxy. These are divided into five *series*, named "A", "B", "C," "D," and "E," based on the first letter of their file names.

The A series (12 files) tests attributes required of all proxies, including ones that operate sequentially. These include the ability to handle both text and binary files, to correctly handle exceptional cases, and to support sequences of transactions involving multiple servers.

The B series (12 files) consists three groups. The first group (6 files) tests the ability of the proxy to handle unexpected errors, such as an invalid host address or a closed connection to a client or server. The second group (4 files) tests the compliance of the proxy to the request header requirements specified in Section 2.2. These tests can also be passed by a sequential server. The third group (2 files) tests the ability of the proxy to retrieve data from real-world web servers (`www.cs.cmu.edu` and `www.ece.cmu.edu`.)

The C series (10 files) tests attributes required of a concurrent proxy. These include cases where the responses occur in a different order than the requests, and ones with many transactions, having the potential to cause race conditions.

The D series (17 files) tests attributes required for a caching proxy. These include the ability to cache text and binary files, as well as responses for nonexistent files. It tests the cache policies on maximum file size, maximum total cache size, and eviction rules. The first five of them can be passed by a serial proxy, as long as it supports caching. The other tests require a concurrent proxy.

Of course, even passing these 51 tests does not ensure complete correctness of a proxy program. Errors involving incorrect synchronization are especially difficult to detect in any testing regime.

8.1 Regression Testing with PXYREGRESS

The program PXYREGRESS automates the process of running a subset of the standard tests using PXYDRIVE. From its local directory, it is invoked as: `./pxyregress.py -p` *PROXY*, where *PROXY* is the path to the file containing the executable proxy program. (PXYREGRESS cannot use an external proxy.) Command-line options include

`-h`
: Print usage information

-p *PROXY*
Run specified proxy program internally.

-s *SERIES*
Specify which series of tests to perform. By default, the four series A, B, C, and D are tested. This argument allows finer control, specified as a string consisting of some combination of the characters 'A', 'B', 'C', and 'D.' For example, the command-line option `-s AB` will run only the tests in series A and B.

-a *LIMIT*
Set a limit on the number of tests that can fail within a series before testing is aborted. If, while running the tests for a particular series, the total number of failures exceeds this limit, testing of that series will terminate. In most cases, this will also cause all testing to terminate. The exception is for series C—upon failure, testing will proceed with series D, since a sequential proxy will fail all tests in series C, but it can possibly pass the first five tests in series D. The default value for the limit is 3.

-c (0-4)
Set the level of checking for concurrency errors.

-t *SECS*
Set a timeout limit of *SECS* seconds for each test. Any test exceeding that limit will be marked as having failed. The default limit is 60 seconds. Setting the limit to 0 indicates that the test should be allowed to run indefinitely.

-l *FILE*
Copy program outputs to *FILE*.

-L *FILE*
Like **-l**, but also copies the log files for all failed tests to *FILE*.

-d *STRETCH*
Adjust all delays and timeout by factor of $STRETCH/100.0$.

PXYREGRESS stores the outputs generated by each of the tests in the `logs` subdirectory.

9 Additional Notes

- The results of PXYDRIVE (and therefore PXYREGRESS) are nondeterministic, due to the variable delays of network communications and proxy actions. From one run to another, a test may pass one time and fail another. When running one of the standard tests, such an inconsistency most likely indicates a synchronization error in the proxy, since the default timeout of 5000 milliseconds should provide ample time for activities to take place. Keep in mind that a correct proxy should pass all tests every time. If tests succeed in some runs but fail in others, it is highly likely that the proxy is incorrect, and it is only by chance that the bugs are not always triggered.

A Performing Concurrency Checks with an External Proxy

Running the concurrency checks with an external proxy is possible, although it is a bit awkward and has some limitations. Here we describe how to do it and some of the pitfalls.

The key is to set an appropriate set of *environment variables* before invoking the proxy program. It is best to do this in a newly created terminal window, and then exit this window once you are done, because some of these environment variable settings can interfere with other programs you may use.

The methods for setting and unsetting environment variables depends on which shell you are using. If you aren't sure, execute the following command:

```
linux> echo $SHELL
```

The two most common responses will be `/bin/csh`, indicating that you are running the C shell (`csh`) or `/bin/bash`, indicating that you are running the "Bourne-again shell" (`bash`.)

With `csh`, an environment variable `VAR` can be set to value 1 and then unset as follows:

```
linux> setenv VAR 1
linux> unsetenv VAR
```

With `bash`, the same actions use the commands:

```
linux> export VAR=1
linux> unset VAR
```

(Make sure there are no spaces before or after the equal sign in the first line.)

It is possible to put these commands in a file, e.g., `settings.sh`, and then batch process them with the command

```
linux> source settings.sh
```

Here are the relevant environment variables.

`LD_PRELOAD`: Assuming you are running in a directory having `pxy` as a subdirectory, set this to `pxy/wrapper.so` to enable the runtime interpositioning that supports the checks.

`CHECK_TIMING`: Set to 1 to enable the **timing** checks.

`CHECK_UNSAFE`: Set to 1 to enable the **unsafe** checks.

`CHECK_LOCKING`: Set to 1 to enable the **locking** checks.

`CHECK_SEMAPHORE`: Set to 1 to enable the **semaphore** checks.

Note: It is important that you unset the environment variable `LD_PRELOAD` when you are done running the proxy. Otherwise, every program you run, including `gcc`, `make`, and you favorite editor will run with this interposition library, slowing them down and possibly impeding their operation.

It *is* possible to run your proxy under GDB while performing these checks, as is described in Section 7.1, but there are a few caveats:

- Don't try the **unsafe**, **locking**, or **semaphore** checks. GDB violates all of these checks.
- It takes a *long* time to start up the shell under GDB, since the debugger makes many system calls while starting a process, and these all incur added delays. You might want to have your proxy print a message when it first starts up, so that you can determine when it's actually started.
- In our experience, you must restart the proxy for every invocation of PXYDRIVE.

Although all of this may seem a bit daunting, you will find this a useful experience in honing your debugging skills.