Fixing File Descriptor Leaks

by

Octavian-Daniel Dumitran

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 30, 2007

Certified by .... [illegible]
Dorothy Curtis
Research Scientist, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Fixing File Descriptor Leaks

by

Octavian-Daniel Dumitran

Submitted to the Department of Electrical Engineering and Computer Science
on May 30, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

We design, implement and test a tool for eliminating file descriptor (FD) leaks in programs at run-time. Our tool monitors FD allocation and use. When the allocation of a new FD would fail because a process's entire pool of FDs is already in use, our tool "recycles" (forcibly closes) an FD. The FD to be recycled is chosen according to three simple rules: 1) do not recycle FDs used within the last second, 2) do not recycle "persistent" FDs (FDs allocated at sites that allocate few FDs) and 3) recycle the FD that was least recently used. We tested our tool with nine applications (HTTP/FTP servers, proxies and clients) that make extensive use of FDs. We first simulated massive leaks by disabling all calls to *close*. Without recycling (and in the presence of leaks), all applications malfunctioned. With FD recycling, applications continued to execute normally despite massive FD leakage. Throughout our tests, our tool has never attempted to close an FD that was still in use.

Thesis Supervisor: Dorothy Curtis
Title: Research Scientist, Department of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my research advisor, Martin Rinard, for his support and guidance during my undergraduate and short graduate experience at MIT.

I would also like to thank my academic advisor, Dorothy Curtis, for her help throughout my stay at MIT. She has gone beyond her duties as an advisor, and provided me with (among others) hardware for classes and help in writing papers.

I would also like to thank Anne Hunter for her patience and her help in dealing with departmental policies and the ISO.

I would also like to thank my family members for their support (moral and material). Special thanks go to my mother: without her constant nagging and help with writing my application, I would not have been admitted to MIT.

I would like to thank (in alphabetical order) Dorothy Curtis, Brittany Low, Martin Rinard and Jing Wang for the valuable feedback they gave me while I was writing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programs are becoming more complex each day. Tracking resource allocation and deallocation is continually more difficult as resource ownership is passed around among different components. Unless resources are properly tracked, programs can "forget" to free resources that are no longer needed, or programs can attempt to release the same resource multiple times.

The best known resource is memory. Memory leaks are the most common resource leaks. If memory is not freed when it is no longer needed, a process might be unable to allocate memory for other tasks. The effect of freeing the same memory location multiple times depends on the language and compiler, but in the case of C (and gcc) this leads to a segmentation fault. Some languages, such as Java and Scheme, lift the burden of memory management off the shoulders of the programmers by providing garbage collection.

Another resource that can be leaked is file descriptors (FDs). Like memory, the Linux kernel frees all FDs of a process upon process termination. Because, typically processes have a pool of 1024 FDs and most processes do not need many FDs, FD leaks might go unnoticed. Unlike memory leaks, when a program running out of memory crashes, programs that leak FDs usually do not crash. Instead, they stop performing normal tasks, which might not be immediately noticeable.

People are not as aware of FD leaks as they are aware of memory leaks. In the case where an FD leak is found in a program, the best solution is to fix the program.

In many instances, this is not possible because

- (a) the source code of the program is not available, or
- (b) the user does not have the technical expertise to fix the bug.

In these instances, a user would have to submit a bug report to the developer and wait for the developer to fix the bug.

The purpose of this project is to design, implement, and test a mechanism of automatically closing leaked FDs, thus allowing applications which leak FDs to continue to operate normally. This mechanism should be used only in emergencies while a patch that fixes the leak is being developed. The mechanism should not encourage developers to become negligent with managing FDs.

In chapter 2, we present related work and background information needed for a better understanding of this paper. We describe our approach in chapter 3 and outline our implementation in chapter 4. We present our experiments in chapter 5 and the results in chapter 6. Finally, we conclude in chapter 7 and offer suggestions for future improvements in chapter 8.

Chapter 2

Background Information

In section 2.1, we discuss related work in the field. Mechanisms used by our test applications for concurrent IO are presented in section 2.2. Our test applications use the HTTP and FTP protocols which are briefly described in section 2.3.

2.1 Related Work

We are not aware of any previous tools that attempt to correct at run-time the execution of programs leaking FDs.

The major problem in fixing FD leaks is detecting which FDs will never be used again, and which ones are still in use. A similar problem occurs when trying to garbage collect an unsafe language. In [11], Boehm analyzes conservative garbage collectors which treat each pattern of bits that could be a pointer as a pointer, in order to prevent garbage collecting data still in use.

Our approach will also have to be conservative because we do not want to close an FD that is still in use. However, unlike memory garbage collection, we cannot treat each integer in memory as an FD because this approach would be too conservative.

We are going to use allocation sites as a criterion for recycling FDs. In [13], Nguyen and Rinard manage memory leaks by setting a quota for each allocation site. When a site exceeds its quota, its least recently allocated memory buffer is reused. In our project, we will not set quotas. Instead we will monitor the number of FDs created at each site. FDs originating from sites with many allocations are more likely to be recycled than FDs originating from sites with few allocations.

2.2 Concurrent IO

HTTP and FTP servers and proxies must be able to handle multiple requests simultaneously. This can be difficult because Linux IO system calls are not asynchronous[15], and a single read operation can block the entire application, thus preventing it from handling any requests. Listening for data on multiple connections at the same time is a problem. Although there exist other methods of dealing with this problem, most applications use one of the following three techniques:

- (a) multiple processes,
- (b) multiple threads (or thread pools), or
- (c) select (or poll) to detect when data is available on an FD.

2.2.1 Multiple Processes

Applications using multiple processes for concurrent IO usually create one process for each data connection and have each process block on each IO operation. The main advantage of this method is simplicity. Some of the disadvantages are 1) the large overhead for creating new processes and 2) difficult inter-process communication (IPC).

A skeletal implementation of a server that uses *fork* to handle each connection is presented in figure 2-1. The server creates a socket and listens for connections. In a loop, the server uses *accept*[1] to create a socket for each new client connection. After the main process creates the connection, it forks a child process. The parent process closes the connection to the client, and goes to the beginning of the loop to wait for new connections. The child process closes the listening socket and deals with the client connection.

[1] *accept* is a blocking system call.

File: /mit/dumitran/thesis/code/fork.c   Page 1 of 1

```
int main() {
  int listening_fd, client_fd, pid;
  listening_fd = socket(...);
  bind(listening_fd, ...);
  listen(listening_fd, ...);

  for (;;) {
    client_fd = accept(listening_fd, ...);
    pid = fork();

    if (!pid) {   /* child process */
      close(listening_fd);

      /*

        Handle connection in client_fd

      */

      exit(0);
    }

    if (pid)     /* parent process */
      close(client_fd);
  }
}
```

Figure 2-1: Sample socket server that uses multiple processes

2.2.2 Multiple Threads

Using multiple threads for concurrent IO is very similar to using multiple processes; after all, threads are processes running within the same address space. As opposed to using multiple processes, using multiple threads is a little more difficult to implement and debug.[2] In addition, it introduces the problem of thread synchronization. However, creating threads has a significantly lower overhead than creating processes, and communication between threads is trivial.

A skeletal implementation of a server that uses threads for concurrent IO is presented in figure 2-2. The main thread creates a socket for listening and waits for client connections in a loop (the call to *accept*). When a client connects, the main thread creates a new thread and goes to the beginning of the loop to wait for connections. The new thread handles the connection and closes the connection FD at the end.

2.2.3 Thread Pools

Section 2.2.2 presents a server that creates a new thread for each connection. Once the connection is closed, the thread terminates. When a new connection is established, a new worker thread is created. In order to reduce the overhead of creating and terminating threads, a well-known technique is to create a pool of workers. The main thread listens for connections and puts the established connections in a queue. Idle worker threads pick up connections from the queue and handle them. After handling a connection, a worker thread does not die. Instead, it waits for another connection to be placed on the queue.

A skeletal implementation of a server that uses a thread pool is presented in figure 2-3. This implementation is more complicated than the implementation in section 2.2.2 and requires some thread synchronization. The main thread creates a listening socket and some worker threads. After that, the main thread enters a loop where it waits for connections and, whenever a connection is established, it puts the connection on a queue (this operation must be atomic, therefore it needs locks) and signals the

[2]Since threads share the same address space, they have access to one another's data and they can modify one another's data in ways that developers do not originally anticipate.

File: /mit/dumitran/thesis/code/thread.c Page 1 of 1

```
struct connection_data { int fd; };

void* handle_request(void *data) {
  int fd = ((connection_data*)data)->fd;

  /*

     Handle the connection in fd

  */

  /* clean connection state */
  close(fd);
  free(data);

  /* The thread is terminated when it exists this functions */
}

int main() {
  int fd, client_fd, pid;
  connection_data *data;
  pthread_t tid;

  fd = socket(...);
  bind(fd, ...);
  listen(fd, ...);

  for (;;) {
    data = malloc(sizeof(connection_data));
    data->fd = accept(fd, ...);
    /* create new thread to handle request */
    pthread_create(&tid, NULL, handle_request, data);
  }
}
```

Figure 2-2: Sample socket server that uses multiple threads

File: /mit/dumitran/thesis/code/pool.c Page 1 of 1

```
struct work_item { int fd; };
queue work_items;

void *worker_thread_main(void*) {
  work_item *data;
  for (;;) {
    pthread_mutex_lock(&mutex);
    while (queue_empty(&work_items))
      pthread_cond_wait(&cond, &mutex);
    data = queue_front(&work_items); queue_pop(&work_items);
    pthread_mutex_unlock(&mutex);

    /*

      Handle connection in data->fd

    */

    /* clean connection state */
    close(data->fd);
    free(data);
  }
}

int main() {
  int fd, client_fd, pid, i;
  work_item *data;
  pthread_t tid;
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  pthread_mutex_t mutex;

  fd = socket(...);
  bind(fd, ...);
  listen(fd, ...);

  pthread_mutex_init(&mutex, NULL);

  for (i = 0; i < number_of_worker_threads; ++i)
    pthread_create(&tid, NULL, worker_thread_main, NULL);

  for (;;) {
    data = malloc(sizeof(connection_data));
    data->fd = accept(fd, ...);
    pthread_mutex_lock(&mutex);
    queue_insert(&work_items, data);
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond);
  }
}
```

Figure 2-3: Sample socket server that uses a thread pool

worker threads. Worker threads enter loops where they wait for connections to be placed on the queue. When signalled, [3] a worker thread handles the connection, cleans connection state and then goes back to wait for more connections to be placed on the queue.

The implementation presented in figure 2-3 uses a constant number of threads, but a dynamic number (i.e., adapted to the load) can also be used. Although this approach is very suited for threads (sharing data is easy), a similar approach can be used with processes. One idea would be to allow the listening socket to be shared by multiple processes.

2.2.4 Synchronous I/O Multiplexing

Programs can handle concurrent IO in a single thread. The key is to make sure that data is available on an FD before an operation on that FD is issued; otherwise the thread will block, and all IO operations would be delayed. Linux provides the system call *select* that monitors multiple FDs and returns when at least one FD has changed status (i.e., data is available for reading, a write operation has completed, or an error has occurred). The *select* system call is blocking; however, if data is already available on one or more FDs when *select* is called then *select* completes immediately.[4]

A skeletal implementation of a server that uses *select* is presented in figure 2-4. The program creates a listening socket and a list of connections which is empty in the beginning. The program enters a loop where it waits for the change of status of one of its FDs: the listening socket or one of the connection sockets. If a change is detected on the listening socket, it means a new connection is available. Consequently, the connection is established, and the corresponding FD is added to the connection list. If a change of status is detected on a connection FD, it means data is available on that connection and the program takes necessary measures.[5] After a connection is

[3]When the main thread signals the condition variable, one worker thread that is waiting on the condition variable is awakened.

[4]The *select* system call has a timeout option and it can complete after a given period of time even if no FDs changed status.

[5]For simplicity, the code does not write to any socket. If the code used the *write* system call, then the target FDs should have been monitored with *select* to detect when the writes complete.

File: /mit/dumitran/thesis/code/select.c Page 1 of 1

```
int main() {
  int fd;
  int conn_fd[1024], len = 0, n, i;  /* list of connection FDs */
  fd_set rset;

  fd = socket(...);
  bind(fd, ...);
  listen(fd, ...);

  for (;;) {

    n = fd;
    FD_ZERO(&rset); FD_SET(fd, &rset);
    for (i = 0; i < len; ++i) {
      FD_SET(conn_fd[i], &rset);
      if (conn_fd[i] > n) n = conn_fd[i];
    }

    select(n, &rset, NULL, NULL, NULL);

    if (FD_ISSET(fd, &rset))
      conn_fd[len++] = accept(fd, ...);
    for (i = 0; i < len; ++i)
      if (FD_ISSET(conn_fd[i], &rset)) {

        read(conn_fd[i], ...);
        /*

          Handle read data from connection

        */
      }
  }}
```

Figure 2-4: Sample socket server that uses the select system call

finished, the FD must be closed and removed from the list of connections.

The main advantage for this method is efficiency: there is no overhead for creating threads or processes. Everything is implemented within the same process, so there is no need for IPC and, unlike threads, there is no need for thread synchronization.

2.3 Internet Protocols

2.3.1 HTTP

Traffic flow in the HTTP protocol [12] is straight-forward. The HTTP sever listens for connections on a port (traditionally port 80). A client connects to the server and sends a request for a page. The server sends the page on the same socket used by the client to make the request.

The early versions of the protocol required that the client connect to the server for each request. Starting with version 1.1, connections can be "kept alive". After responding to a request, the server can keep the connection open. If a client has other requests, it can send the requests using the same socket, thus eliminating the need to create additional connections. Both the server and the client must be able to keep the connection alive. If at least one of the server or client cannot keep the connection alive (i.e., the implementation dates from before the option was added to the protocol, or the developer did not implement the feature for simplicity), the connection is closed and new connections are created for each request.

HTTP proxies are applications that act as intermediaries between HTTP clients and HTTP servers. The main purpose of an HTTP proxy was caching. However, recently, proxies are being used for security reasons (i.e., restricting access from a network to HTTP traffic only). Like a server, a proxy listens for connections on a port. A client connects to the port and makes a request. The proxy reads the request and forwards it to the server. When the server replies, the proxy forwards the response to the client. To a server, a proxy behaves exactly like a client; in other words, the server cannot tell the difference between a proxy and a non-proxy client.

2.3.2 FTP

FTP [14] is slightly more complicated than HTTP. The server listens for connections (usually on port 21). A client connects to the server and opens a socket (*control stream*) used to send commands to the server.[6] File transfer does not occur on the control stream. Instead, for each file, a new TCP connection is created.

FTP has two modes : active mode and passive mode. By default, servers are in active mode. Servers switch to passive mode if clients issue the "PASV" command through the control stream.

In active mode, before a client downloads or uploads a file, it opens a port and sends the port number to the server over the control stream. The server connects to the client on the specified port, and then the data transfer begins.

In passive mode, the server opens a port and sends the port to the client over the control stream. The client connects to the server on the specified port, and then the data transfer begins.

Unlike HTTP, FTP has a higher overhead and it allows clients to upload files to the server.

[6]Although it happens rarely, the server can also send commands to the client.

Chapter 3

Our Approach

Our goal is to mask FD leaks of an application and to keep the application running normally (or at least with as few errors as possible) despite the fact that it is leaking FDs.

An FD leak will interfere with the normal execution of a program when the program will not be able to create a new FD because all the FDs in its pool are in use. FD creation is the only moment when FD leaks may impede a running program. Our basic approach will be to monitor FD creation. In the case where creating an FD fails because the entire FD pool is used, we will choose one FD and close it. The operation of forcibly closing an FD will be called *recycling.*

We want our recycling mechanism to be as conservative as possible. Therefore recycling will be delayed as long as possible, and it will only be performed when creating a new FD would fail because of the full occupancy of the FD pool.

The most difficult part of recycling is deciding which FD to close. We cannot really be sure if an application will ever use an FD in the future; therefore, we must devise a heuristic that will pick an FD that is least likely to be used.

There are three rules that we will use when deciding which FD to recycle:

- (1) Try to close an FD that was as least recently used as possible.
- (2) Do not close "persistent" FDs.
- (3) Never close an FD that was used within the last 1 second.

We are going to separate FDs into two categories : 1) persistent and 2) short-lived. Persistent FDs are used for the entire execution of the program. They are usually dormant and are used only sporadically. They are closed at the end of the program, or, sometimes, not explicitly closed at all. The first example of persistent FDs are listening sockets. For an HTTP server, a listening socket can be inactive for arbitrary periods of time (i.e. no clients are connecting to the server), but the socket is used for the entire lifespan of the process. FDs associated with log files are another example of persistent FDs: applications can log some events (that occur rarely) or they can write statistics after large periods of time. Short-lived FDs are used for a short period of time immediately after creation and then closed. Examples of short-lived FDs are 1) connection sockets used for HTTP traffic, 2) FDs used to read files requested by HTTP clients and 3) FDs used for name resolution (files for /etc/hosts or sockets for UDP datagrams to DNS servers).

Differentiating persistent FDs from short-lived FDs is a difficult task. First of all, our definition is not very accurate and it leaves plenty of room for interpretation. Secondly, determining if a program will ever use an FD again is an undecidable problem. For the rest of this paper, FDs allocated at sites that created fewer than five FDs will be considered persistent FDs. All other FDs will be considered short-lived. Therefore, our second rule[1] means that FDs created at sites with fewer than five FDs allocated are not to be closed.

Initially, we considered the allocation site of an FD to be the function that creates the FD. We found that some applications have their own wrappers for functions like *open, close, read, write*, and that, using the calling function as the definition for allocation site would cause (for some applications) most FDs to have the same allocation site. We decided instead to have the entire stack trace (prior to the system call) be the allocation site.

Our algorithm for choosing an FD to recycle is the following:

- (1) maintain a LRU (Least Recently Used) list of FDs

[1]Do not close "persistent" FDs.

- (2) whenever an FD is used or created, move it to the front of the list
- (3) whenever an FD is closed, remove it from the list
- (4) when an FD needs to be recycled, traverse the list backwards (least recently used FD is iterated first). If the current FD was used within the last second, then stop without recycling any FD. If the current FD is "persistent" then move to the next FD, otherwise recycle current FD.

Chapter 4

Implementation

We wanted to implement our tool as a kernel module. The advantage was that we would trap all system calls, but the disadvantages were that we had to duplicate state when *fork* was called (and, thus, keep state for multiple processes in the kernel) and write kernel code which is very difficult to debug. One additional disadvantage was that user would need superuser privileges to use our tool.

We decided against implementing a kernel module and opted for a shared library instead. More details about the shared library are given in section 4.1. Section 4.2 describes how our implementation can cause FD overloading.

4.1 Shared Library

We ended up implementing our tool as a shared library that is loaded when applications start. This approach has its features, but it has serious limitations. It is easier to implement and debug than a kernel module and state has to be kept for only one process. The state is automatically replicated by the kernel on forks. One big disadvantage is that our tool shares the fate of the monitored application. If the application crashes, our library (along with all statistics) dies as well. In order to make up for this inconvenience, we also implemented a small statistics server to which our library connects to report statistics. The statistics server has two roles: (1) it makes sure that data survives if an application crashes and (2) it collects data from more processes (that would otherwise act independently and not share statistics).

The mechanism we used to trap functions is presented in figure 4-1. The sample code traps only calls to "read". Our library is preloaded, which means that the dynamic linker will forward the calls to "read" to our code, not to the original *libc* "read" function. When the library is loaded, we initialize the pointer to the real "read" function to NULL. The library containing "read" might not be loaded at this time, so asking the loader to find symbol "read" at this point could be fruitless. When "read" is called, we ask the linker to find the address for symbol "read" (the pointer to the libc "read" function). We perform our monitoring tasks and call the original "read" function in the end. For performance, the pointer to the original function is saved, so the loader is queried only once. More details about shared libraries can be found in [5].

Libc provides access to system calls via wrappers. Our hope was that by trapping the calls to the *libc* wrapper functions, we would trap all system calls. For various reasons, we were not able to trap all system calls, but we were able to trap all system calls that create or close FDs and most FD system calls used by our test applications. Because we are missing some system calls, our LRU list is not exact, so our heuristic has to make decisions based on an approximation of the real LRU list. Our results show that our heuristic functioned properly despite our library missing system calls.

The main reasons for missing systems calls are

- (a) there are *libc* functions that we do not trap
- (b) users issue system calls directly (without using *libc* wrappers)
- (c) user programs link the *libc* library statically
- (d) we cannot trap *libc* function calls made within *libc* itself (because of the way *libc* is linked)

Table 4.1 lists the functions that our tool traps.

File: /mit/dumitran/thesis/code/shared.c   Page 1 of 1

```
/* demo.c */
#include <unistd.h>
#include <stdio.h>

ssize_t (*orig_read)(int, void*, size_t);

void lib_init(void) __attribute__((constructor));
void lib_shutdown(void) __attribute__((destructor));

void lib_init(void) {
  orig_read = NULL; /* no value for the original function */
}

void lib_shutdown {
  /* nothing needs to be cleaned up */
}

ssize_t read(int fd, void* buffer, size_t len) {
  ssize_t result;

  if (!orig_read)
    /* find address for symbol "read" (the pointer for the
       original read function */
    orig_read = ssize_t(*)(int, void*, size_t)
      dlsym(RTLD_NEXT, "read");

  /* Perform "pre-call" tasks */
  printf("Starting a read operation\n");

  /* Make the original call */
  result = orig_read(fd, buffer, len);

  /* Perform "post-call" tasks */
  printf("Read operation ended\n");

  /* Return nothing to the called */
  return res;
}

/*
   Generating the library

   gcc -fPIC -g -c -Wall demo.c
   gcc -shared -Wl,-soname,libdemo.so.1 -o libdemo.so.1.0.1 demo.o -lc

*/

/*
   Running a test program (using bash as a shell)

   LD_PRELOAD=libdemo.so.1.0.1 <test_program>

*/
```

Figure 4-1: Writing wrappers for system functions using a shared library

ID	Function	Description
1	open	opens or creates a file
2	creat	creates a file (particular case of "open")
3	close	closes an FD
4	socket	creates a socket
5	accept	accepts a connection and creates a socket for it
6	socketpair	creates a pair of connected sockets
7	dup	duplicates an FD
8	dup2	duplicates an FD; the target FD is specified
9	pipe	creates a FIFO pipe
10	fopen	opens a file (the result is a stream, not an FD)
11	fclose	closes a steam (also releases the associated FD)
12	bind	binds a socket to a specific address and port
13	listen	makes a socket to listen for connections
14	connect	connects a socket to a specified address
15	read	reads data from an FD
16	write	writes data to an FD
17	readv	reads data from an FD to multiple buffers
18	writev	writes data from multiple buffers to an FD
19	recvfrom	reads data from a socket (more functionality than read)
20	sendto	writes data to a socket (more functionality than write)
21	send	a special case of sendto
22	recv	a special case of recvfrom
23	select	waits for the change of status of a set of FDs
24	printf	prints a formatted text to the "stdout" stream
25	fflush	flushes the buffers of a stream
26	fprintf	prints a formatted text to a given stream
27	vprintf	similar to printf
28	fvprintf	similar to fprintf
29	fread	reads data from a stream
30	fwrite	writes data to a stream
31	fputc	writes a character to a stream
32	fputs	writes a string to a stream
33	sendfile	copies data between a file and a socket
34	sendfile64	similar to sendfile
35	pread	reads data from a file at a given offset
36	pwrite	writes data to a file at a given offset
37	poll	similar to select

Table 4.1: Trapped functions

4.2 FD Overloading

Our strategy of recycling FDs can potentially close an FD that is still in use and lead to FD *overloading*. Overloading occurs when an application uses the same physical number to represent two or more FDs. Figure 4-2 shows a scenario that can lead to FD overloading. Thread A executes instructions *a1*, and obtains value x for fd. Thread A than executes *a2* and goes on performing computations that do not use FDs. Meanwhile, all FDs are being allocated and x is the last FD in the LRU list. Thread B starts and tries to execute *b1*. Since there are no FDs available, our library recycles an FD which happens to be x. The call in instruction *b1* will return x. Right now, x is an overloaded FD. Both thread A and thread B think they each have an FD opened and mapped to x. In reality, only thread B has an FD mapped to x. When thread A uses FD x, it will use thread B's FD. If, after *b1*, the kernel decides to interrupt thread B and run thread A, then thread A will actually corrupt B's file by writing inappropriate information. In addition, thread A will also close B's FD without B knowing it.

Overloading can have unpredictable consequences. In order to avoid overloading in our tests, we used the fact that our test applications do not actually leak FDs: their leaks are just simulated by our library. Whenever an application calls *close*, we mark the FD as *closable* and do not release the FD. When we recycle an FD, we first check if it is closable. If it is not, we exit with an error message.

File: /mit/dumitran/thesis/code/overload.c Page 1 of 1

```
void thread_a() {
  int fd;
  /* a1 */ fd = open(some_file, O_WRONLY);
  /* a2 */ write(fd, "something", 9);

  /*

       Wait

  */

  /* a3 */ write(fd, "bad string", 10);
  /* a4 */ close(fd);
}

void thread_b() {
  int fd;
  /* b1 */ fd = open(some_other_file, O_WRONLY);
  /* b2 */ write(fd, "good string", 11);
  /* b3 */ close(fd);
}
```

Figure 4-2: Code fragment that can lead to FD overload

Chapter 5

Experimental Design

Section 5.1 presents the applications that we used in our tests and section 5.2 discusses the environment for the tests. Finally, section 5.3 describes our experiments.

5.1 Applications

We used HTTP/FTP applications (servers, proxies and clients) to test our library because these applications make extensive use of FDs, and a leak in such an application will most likely cause malfunctions. The applications that we used are:

- (1) Apache version 2.2.0.[1] Apache is a very popular HTTP server. It is maintained and developed by an open community of developers working together with the Apache Software Foundation. Apache is available for a variety of operating systems including Windows NT, Linux and MacOS. Apache is free and open software released under the Apache License.

- (2) OOPS version 1.5.23.[3] OOPS is an HTTP proxy server comparable to Squid. The project was started in order to offer users an alternative to Squid which (according to the authors of OOPS) has several shortcomings.

- (3) Squid version 2.5.6.[6] Squid is a very popular proxy server and web cache daemon. Although it offers a lot more functionality, in our tests, we have used Squid as an HTTP proxy server.

- (4) TinyProxy version 1.6.3.[8] TinyProxy is a lightweight HTTP proxy that comes under GPL license. It is designed to be fast and small.

- (5) WGET-1.10.2.[10] WGET is a very popular download manager that is part of the GNU project. Its features include recursive download which means that WGET can also be used as a web crawler.

- (6) Hydra version 0.1.8.[2] Hydra is a small multi-threaded HTTP server. Hydra is based on Boa version 0.94.13. Hydra is covered by the GNU GPL License.

- (7) Pure-FTPd version 1.0.20.[4] Pure-FTPd is a free, secure and ease to use FTP server. It is popular and it can be compiled and run on most UNIX-like operating systems including Linux.

- (8) Thttpd version 2.25b.[7] Thttpd is an open source HTTP server. The "t" before HTTP stands for tiny, turbo or throttling. Thttpd is small and fast. It can be compiled and run under most UNIX-like operating systems including Linux.

- (9) WebFS version 1.20.[9] WebFS is a simple HTTP server that is used mostly for static content. It is meant to be a quick way to share information. WebFS does not have a configuration file; it only has a few command line options.

To the best of our knowledge, none of the applications used in our tests has bugs that can lead to FD leaks.

5.2 Experiment Environment

We ran our experiments on a computer with two 3.20Ghz Intel(R) Pentium(R) 4 CPUs and a 300Gb hard disk. The operating system was a default Fedora Core release 4 (Stentz) full installation (all packages installed).

During our experiments, we used our libraries in three different modes:

- (1) Monitor Mode. In Monitor Mode, our library monitors FD creation, closure and use, but does not alter the original behavior of the program. Calls to *close* are forwarded to the *libc* function *close*. The purpose of Monitor Mode is to study programs' FD allocation and use.

- (2) Leak Mode. In Leak Mode, our library monitors FD creation and use and it disables FD closure. In Leak Mode, our library does not attempt to recycle FDs. The purpose of Leak Mode is to see how applications behave in the presence of FD leaks.

- (3) Fix Mode. In Fix Mode, our library monitors FD creation and use and it disables FD closure. Furthermore, our library recycles FDs when a process exhausts its entire pool. In order to prevent FD overloading and unpredictable behavior, our library monitors calls to *close* and exits with an error when it tries to recycle an FD that was not closed.[1]

In order to trap as many system calls as possible, we recompiled our test applications and had the linker link everything dynamically.

We created a directory with 1024 small (14 byte) files, 2 big (64Mb) files and one index file that contained links to all other files. In our tests, we had WGET create a mirror of this directory by having WGET recursively download the contents of an HTTP or FTP server serving files from out test directory.

5.3 Experiments

For each of our HTTP servers (Apache, Hydra, Thttpd and WebFS) we ran 24 experiments. We ran 10 experiments with 1 through 10 WGET clients downloading the site serially with the library in Monitor Mode. We then ran 10 more experiments with the library in Monitor Mode, but this time the 1 through 10 WGET clients were downloading the site concurrently. We ran 2 experiments with our library in Leak

[1]The FDs are not closed per say because our library disables FD closure. Instead the FDs are closable meaning our library detected the program tried to close the FD.

Mode: 1) 10 WGET clients downloading serially and 2) 10 WGET clients downloading concurrently. We repeated the two experiments above with our library in Fix Mode.

For each of our HTTP proxies (OOPS, Squid and TinyProxy), we ran the same 24 experiments, but we had the clients download the site from an Apache server (running without our library loaded) using a test proxy (running with our library loaded).

For our FTP server (Pure-FTPd), we ran the same 24 experiments with WGET downloading from a Pure-FTPd server using active mode.

For all the experiments above, only the application being tested had our library loaded. Other applications used in the experiments (WGET and, for proxy servers only, Apache) did not have our library loaded.

For WGET, we had WGET with our library loaded download the site from an Apache server running without our library. We ran three experiments for WGET: one in Monitor Mode, one in Leak Mode and one in Fix Mode.

Our hope is that most FD accesses occur to FDs that are towards the front of the LRU list (this is another way of saying that a process does not need many FDs at any given time). We built histograms that count how many times accesses have been made to each position on the LRU list. These histograms are called *backlists*. Ideally, most accesses occur at small positions in the LRU list, and the accesses that occur at high position refer to persistent FDs.

Because we will use backlists to present our results, we would like to spend more time explaining the concept. When building a backlist, we are not interested in knowing what FD was used; we only need to know the location of the FD in the LRU list. Imagine that at the beginning of the execution (when the LRU list is initially empty), the following sequence of FDs is used : 0, 0, 1, 1, 2, 2, 0, 3, 3, 4, 4, 2, 5, 5. The LRU lists and backlists after each FD access are presented in table 5.1. In the first step, 0 is inserted in the LRU list, but since it did not exist in the LRU list before it was accessed, the backlist is not changed. In step 2, 0 is located on position 0 of the LRU list, so the count of 0 is increased in the backlist. 0 is moved to the front of the LRU list (it is not actually moved, because it was already on the first position). Steps 3, 5, 8, 10 and 13 insert previously non-existent elements in the LRU list and they do not modify the backlist. Steps 4, 6, 9, 11 and 14 access the first element of the LRU list, so the count for position 0 gets incremented in the backlist. Step 7 accesses FD 0 which is on position 2 of the LRU list. The count for position 2 is incremented and FD 0 is moved to the front of the LRU list. Step 12 accesses FD 2 which is on position 3 of the LRU list, so the count for position 3 is incremented and FD 2 is moved to the front of the LRU list.

We define the size of a backlist[2] to be the largest position in the LRU list that has a non-zero count in the backlist. Notice that FDs that are not used tend to slide back in the LRU list, but unless they are used again in the future, they do not affect the length of the backlist.[3]

Another note worth mentioning is that backlists tend to spread out more in Fix Mode than in Monitor Mode. The reason behind this phenomenon is the fact that when an FD is closed, it is removed from the LRU list (so FDs move towards the front). In Fix Mode, calls to close are ignored, so FDs are not removed from the LRU list.

[2]Backlist is a poor choice for this term because our backlists are actually maps, so backmap would be a better term. Originally, we wrote backlists as a list; for example (0:6; 2: 1; 3: 1) was written as (6 0 1 1). We adopted the map notation because our backlists may contain isolated entries for large positions. The definition for the length of a backlist makes more sense if we think about the list notation as opposed to the map notation.

[3]A good example is FD 1 in table 5.1.

Step	FD used	LRU list	backlist
1	0	0	()
2	0	0	(0: 1)
3	1	1 0	(0: 1)
4	1	1 0	(0: 2)
5	2	2 1 0	(0: 2)
6	2	2 1 0	(0: 3)
7	0	0 2 1	(0: 3; 2: 1)
8	3	3 0 2 1	(0: 3; 2: 1)
9	3	3 0 2 1	(0: 4; 2: 1)
10	4	4 3 0 2 1	(0: 4; 2: 1)
11	4	4 3 0 2 1	(0: 5; 2: 1)
12	2	2 4 3 0 1	(0: 5; 2: 1; 3: 1)
13	5	5 2 4 3 0 1	(0: 5; 2: 1; 3: 1)
14	5	5 2 4 3 0 1	(0: 6; 2: 1; 3: 1)

Table 5.1: Backlist example

Chapter 6

Experimental Results

Each of the nine sections in this chapter describes the experimental results of one application.

6.1 Apache

We first ran Apache in Monitor Mode to observe file descriptor usage patterns. After that, we ran Apache in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran Apache in Fix Mode to see if our library could improve the performance in the presence of FD leaks.

6.1.1 Apache in Monitor Mode

Even before running our experiments, we noticed that Apache is a multi-process application that has one thread per process. We observed that the first two processes started by the application do not serve clients; instead they are just used to read the configuration files and start worker processes.

We ran repeated experiments with different numbers of WGET clients serially downloading our test site. We found that a total of 8 processes (2 starter processes and 6 worker processes) are used regardless of the number of clients used. Obviously, as the number of clients increases so does the number of FDs created by the worker

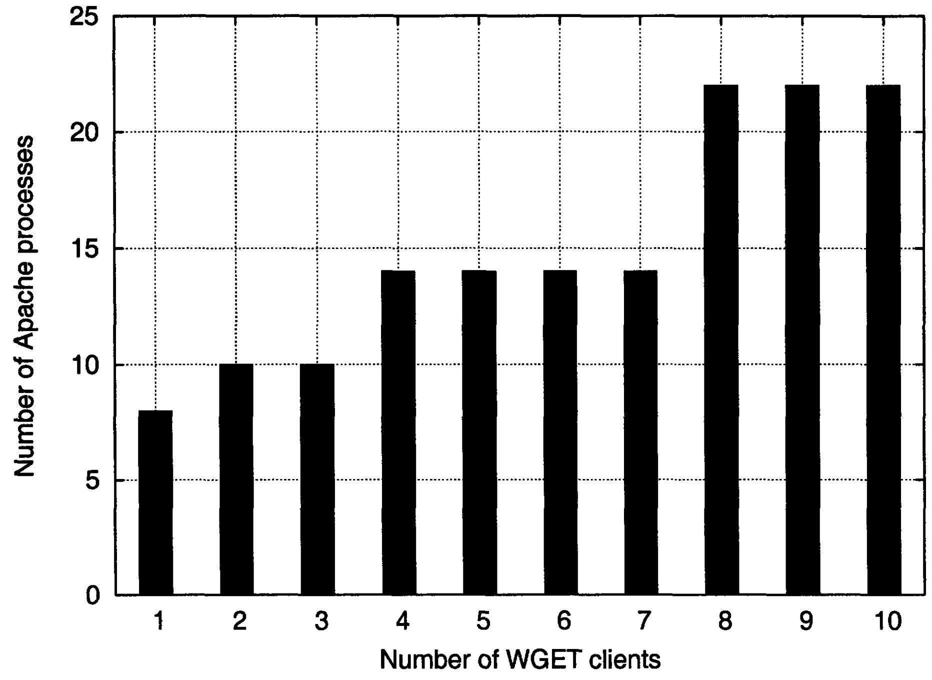

Figure 6-1: Number of Apache processes

processes.

We ran repeated experiments with different numbers of WGET clients concurrently downloading our test site. We found that the number of processes created by Apache increases as the number of clients increases. Furthermore, FD creation is evenly distributed among the worker processes. Figure 6-1 shows the number of processes as a function of the number of clients.

Table 6.1 presents a list of FD allocation sites together with the function used to allocate FDs, the number of FDs created at the site and a brief description. The data was obtained using 10 parallel WGET clients.

We created a backlist in order to see how Apache uses file descriptors. The backlist was obtained using 10 parallel WGET clients. As figure 6-2 shows, most of the accesses refer to recent file descriptors. All FDs that were accessed on positions 3 or more in the backlist were either

- (a) a pipe,
- (b) a listening socket, or

ID	Function	Number	Description
1	open	1	opened httpd.conf
2	socket	2	sockets for listening
3	fopen	1	opened /etc/passwd
4	fopen	1	opened /etc/group
5	fopen	1	opened /etc/hosts
6	fopen	1	opened /etc/hosts
7	fopen	1	opened /etc/hosts
8	pipe	2	pipe for IPC
9	open	1	opened logs/error_log
10	dup2	1	dup for logs/error_log
11	open	1	opened logs/access_log
12	open	1	opened conf/mime.types
13	open	1	opened conf/httpd.conf
14	fopen	1	opened /etc/passwd
15	fopen	1	opened /etc/group
16	fopen	1	opened /etc/hosts
17	fopen	1	opened /etc/hosts
18	fopen	1	opened /etc/hosts
19	pipe	2	pipe for IPC
20	open	1	opened logs/error_log
21	dup2	1	dup for logs/error_log
22	open	1	opened logs/access_log
23	open	1	opened conf/mime.types
24	open	1	opened logs/httpd.pid
25	accept	65	sockets for handling connections
26	open	5310	reading user-requested files
27	accept	57	sockets for handling connections
28	open	4962	reading user-requested files
29	socket	2	sockets for self-connecting (used to unblock accept)

Table 6.1: FD Allocation Sites for Apache

- (c) a log file (access_log or error_log).

All three categories of file descriptors above are allocated in sites that create 1 or 2 file descriptors.

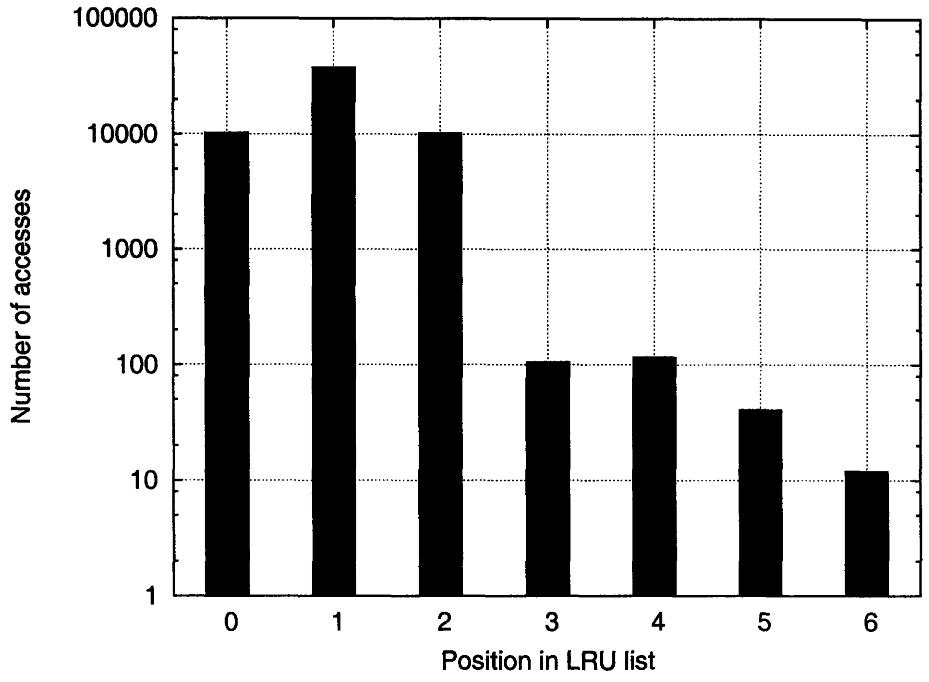

Figure 6-2: Backlist of Apache in Monitor Mode

6.1.2 Apache in Leak Mode

We ran Apache in Leak Mode with 10 parallel clients, and, to our surprise, all clients have successfully downloaded the entire site. The reason behind this is that Apache creates many processes when receiving a lot of simultaneous connections. Apache created 20 worker processes (each with a pool of around 1000 file descriptors) in order to server around 10,000 requests. Overall, Apache had twice as many FDs as it needed for the requests.

We tried to limit the total number of FDs that Apache can use by having clients download serially, rather than in parallel. This way, Apache does not create a large number of processes, so it does not have as many FDs at its disposal. We ran 10 serial clients, and Apache managed to perform admirably. All but 14 files (out of a total of 10270) were downloaded successfully. Figure 6-3 shows how many files each client managed to download. We noticed that some processes stopped functioning (because no more FDs could be allocated), but Apache adapted and automatically forked new processes. Instead of 6 worker processes (that were created for a similar load under Monitor Mode), Apache needed 13 worker processes.

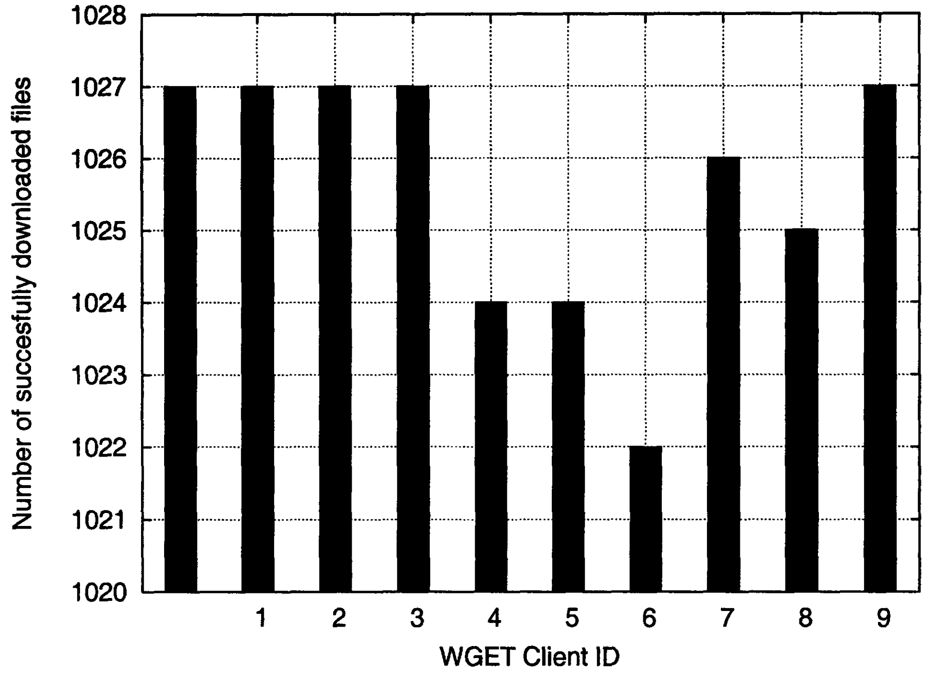

Figure 6-3: Succesful downloads with Apache under Leak Mode

6.1.3 Apache in Fix Mode

We tested Apache in Fix Mode in two situations: 10 parallel clients and 10 serial clients. In both cases, the clients have successfully managed to download all files, and the number of worker processes created equaled the number of processes created under Monitor Mode.

A backlist obtained from the experiment with parallel clients is presented in figure 6-4. Most accesses are recent (positions 0, 1 or 2 in the LRU list), however, some accesses go as far as 722. Fortunately, all three accesses going to positions 722, 448 and 212 were FDs mapped to logs/error_log, which come from allocation sites that do

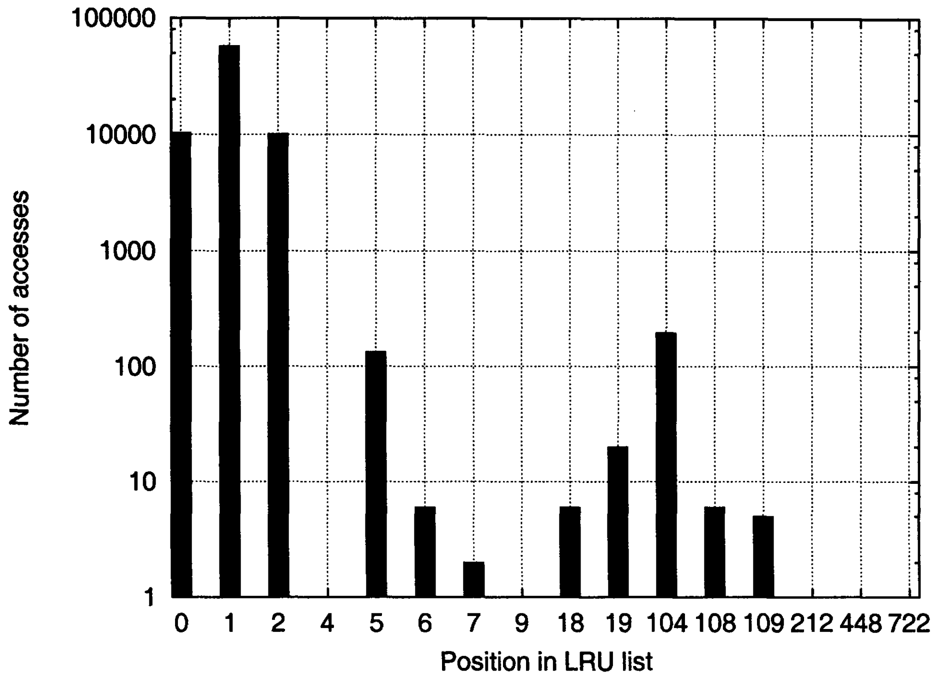

Figure 6-4: Backlist of Apache in Fix Mode

not open a lot of file descriptors. For this reason, our heuristic does not close these FDs. The FDs accessed on positions 4 though 109 are either:

- (a) pipes
- (b) listening sockets, or
- (c) log files (access_log or error_log).

These FDs are not closed either because of their allocation site.

6.2 Squid

We first ran Squid in Monitor Mode to observe FD usage patterns. After that, we ran Squid in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran Squid in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.2.1 Squid in Monitor Mode

We ran Squid in Monitor Mode with different numbers of clients downloading our site using an Apache server and a Squid proxy. We ran the tests using clients running concurrently and clients running serially. We noticed that Squid always creates four processes, each process having one thread. One of the four processes serves all the requests, while the other three are used just for reading configuration files and writing statistics. The single-threaded process that handles all requests uses the libc function *poll* for asynchronous IO.

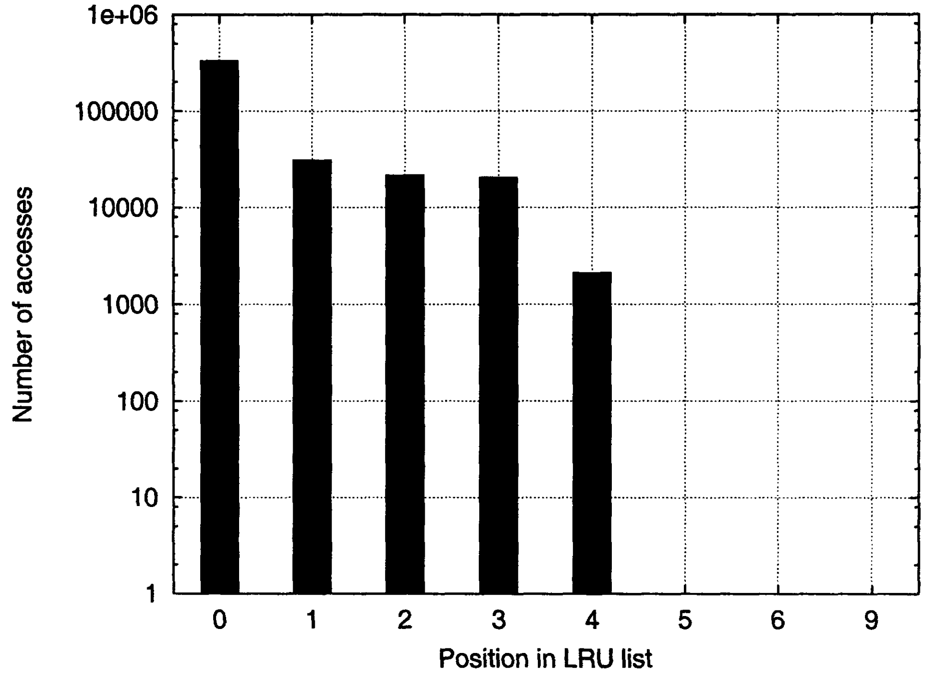

Figure 6-5: Backlist of Squid in Monitor Mode with 10 serial clients

A backlist obtained for 10 serial clients is presented in figure 6-5. Almost all accesses are made to positions 0, 1, 2, 3 or 4. There is one access for each of the following positions: 5, 6 and 9. These accesses, however, use file descriptors associated with log files. While serving serial requests, Squid does not go back beyond position 4 for active connection FDs.

The backlist obtained for 10 parallel clients, presented in figure 6-5 is more spread out than the backlist for serial clients. The access made for position 27 refers to an

FD bound to access.log, however, active connections can be found up to position 24.

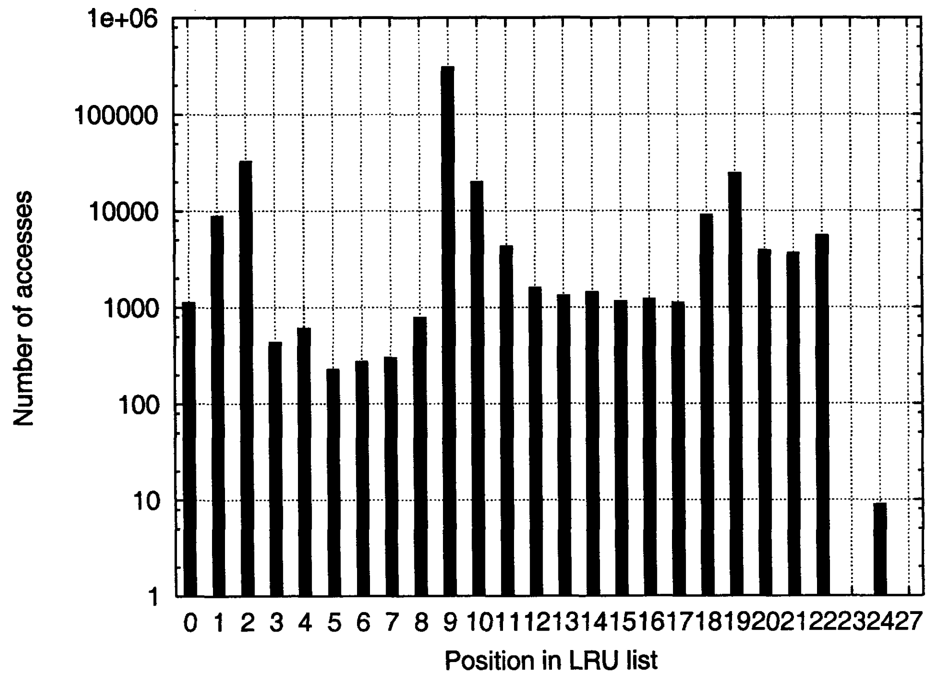

Figure 6-6: Backlist of Squid in Monitor Mode with 10 parallel clients

We looked at how far back in the LRU list active connections occur. Figure 6-7 shows the furthest occurrence of an active connection as a function of the number of parallel clients. It is easy to observe that active connections tend to spread out into the LRU list as a linear function of the number of clients.

Table 6.2 shows a list of allocation sites for Squid. The number of FDs created by each site were recorded while Squid was used by 10 clients concurrently. Most allocation sites only create one file descriptor (or two in the case of pipes). Allocation site 27 reads 29 HTML pages, each page corresponding to an error message. When an error occurs, Squid returns one of its 29 error pages. Allocation site 44 reads 25 icon files which Squid uses to embed in its pages. All HTML error pages and icon files are opened only once, and their content is loaded into main memory. The FDs allocated at sites 27 and 44 are used only immediately after their creation, so in the case of a leak, it would be acceptable to close them. The other allocation sites that create many file descriptors are 49, 50, 51, 52 and 54; all of these are short-lived FDs that handle user requests.

ID	Function	Number	Description
1	fopen	1	opened squid.conf
2	fopen	0	used by gethostbyname
3	fopen	0	used by gethostbyname
4	fopen	1	opened /etc/resolve.conf
5	socket	1	socket used by gethostbyname
6	socket	1	socket used by gethostbyname
7	fopen	1	opened /etc/hosts.conf
8	fopen	1	opened squid.pid
9	open	0	missing optional configuration file
10	open	1	opened /dev/null
11	dup	1	dup of /dev/null
12	dup	1	dup of /dev/null
13	dup	1	dup of /dev/null
14	fopen	1	opened squid.conf
15	fopen	0	used by gethostbyname
16	fopen	0	used by gethostbyname
17	fopen	1	opened /etc/resolv.conf
18	socket	1	used by gethostbyname
19	socket	1	used by gethostbyname
20	fopen	1	opened /etc/hosts
21	fopen	1	opened squid.pid
22	fopen	1	opened cache.log
23	socket	1	used by gethostbyname
24	fopen	1	opened /etc/hosts
25	socket	1	listening socket
26	fopen	1	opened /etc/resolv.conf
27	open	29	files used for reading error messages
28	open	1	opened access.log
29	pipe	2	pipe for IPC
30	pipe	2	pipe for IPC
31	open	1	opened /dev/null
32	dup	1	dup for reading end of a pipe
33	dup	1	dup for writing end of a pipe
34	dup	1	dup of cache.log
35	dup	1	dup for reading end of a pipe
36	dup	1	dup for writing end of a pipe
37	dup	1	dup of cache.log
38	open	1	opened store.log
39	open	1	opened /dev/null
40	open	1	opened swap.state
41	open	1	opened swap.state.new
42	fopen	1	opened swap.state
43	fopen	1	opened mime.conf
44	open	25	icon files
45	socket	1	listening socket
46	socket	1	listening socket
47	fopen	1	opened /etc/services
48	open	1	opened squid.pid
49	accept	5129	client connections
50	socket	104	connections to HTTP servers
51	accept	4961	client connections
52	accept	169	client connections
53	open	1	opened swap.state
54	accept	21	client connections

Table 6.2: FD Allocation Sites for Squid

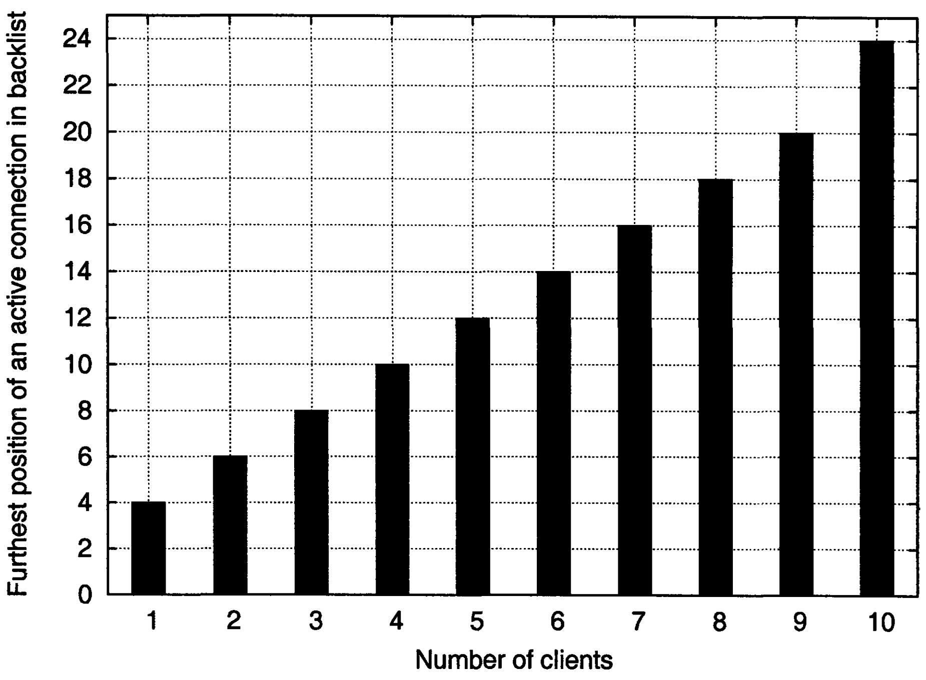

Figure 6-7: Number of FDs needed by Squid

6.2.2 Squid in Leak Mode

We ran Squid in Leak Mode with 10 parallel clients, and then with 10 serial clients. In the case of parallel clients, most clients managed to download the first 92 files (one client downloaded 90 files, another client downloaded 91 files, and the remaining 8 clients downloaded 92 files each). For the serial clients, the first client downloaded 926 files and exhausted the entire file descriptor pool of the server. After Squid exhausted its FD pool, clients would hang waiting for a response from Squid and Squid would not return any reply because it could not create sockets to connect to the HTTP server.

6.2.3 Squid in Fix Mode

We ran Squid in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to download all the files.

A backlist with 10 parallel clients is presented in figure 6-8. The accesses made to positions 53, 59, 300, 978 and 980 refer to log files. Active connections spread out only up to position 34 in the LRU list.

6.3 OOPS

We first ran OOPS in Monitor Mode to observe FD usage patterns. After that, we ran OOPS in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran OOPS in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.3.1 OOPS in Monitor Mode

We ran OOPS in Monitor Mode with different numbers of clients downloading our site using an Apache server and an OOPS proxy. We ran the tests using clients running concurrently and clients running serially. OOPS always creates a single process, regardless of the workload. Moreover, in the case of serial clients, OOPS uses 9 threads. When multiple clients make requests simultaneously, the number of threads increases. As figure 6-9 shows, the number of threads is a linear function in terms of the number of clients.

A backlist obtained for 10 serial clients is presented in figure 6-10. Active connections use FDs only on positions 0, 1 and 2. Accesses to positions 4 and 8 are made by FDs linked to *oops.log*. The access to position 12 is made by an FD linked to *access.log*.

The backlist obtained for 10 parallel clients, presented in figure 6-10 is more spread out than the backlist for serial clients. The access made for position 20 refers to a FD bound to *access.log*, however, active connections spread until position 18.

We looked at how far back in the LRU list active connections occur. Figure 6-12 shows the furthest occurrence of a active connection as a function of the number of parallel clients. It is easy to observe that active connections tend to spread out into the LRU list as a linear function of the number of clients.

Table 6.3 shows a list of allocation sites for OOPS. The number of FDs created

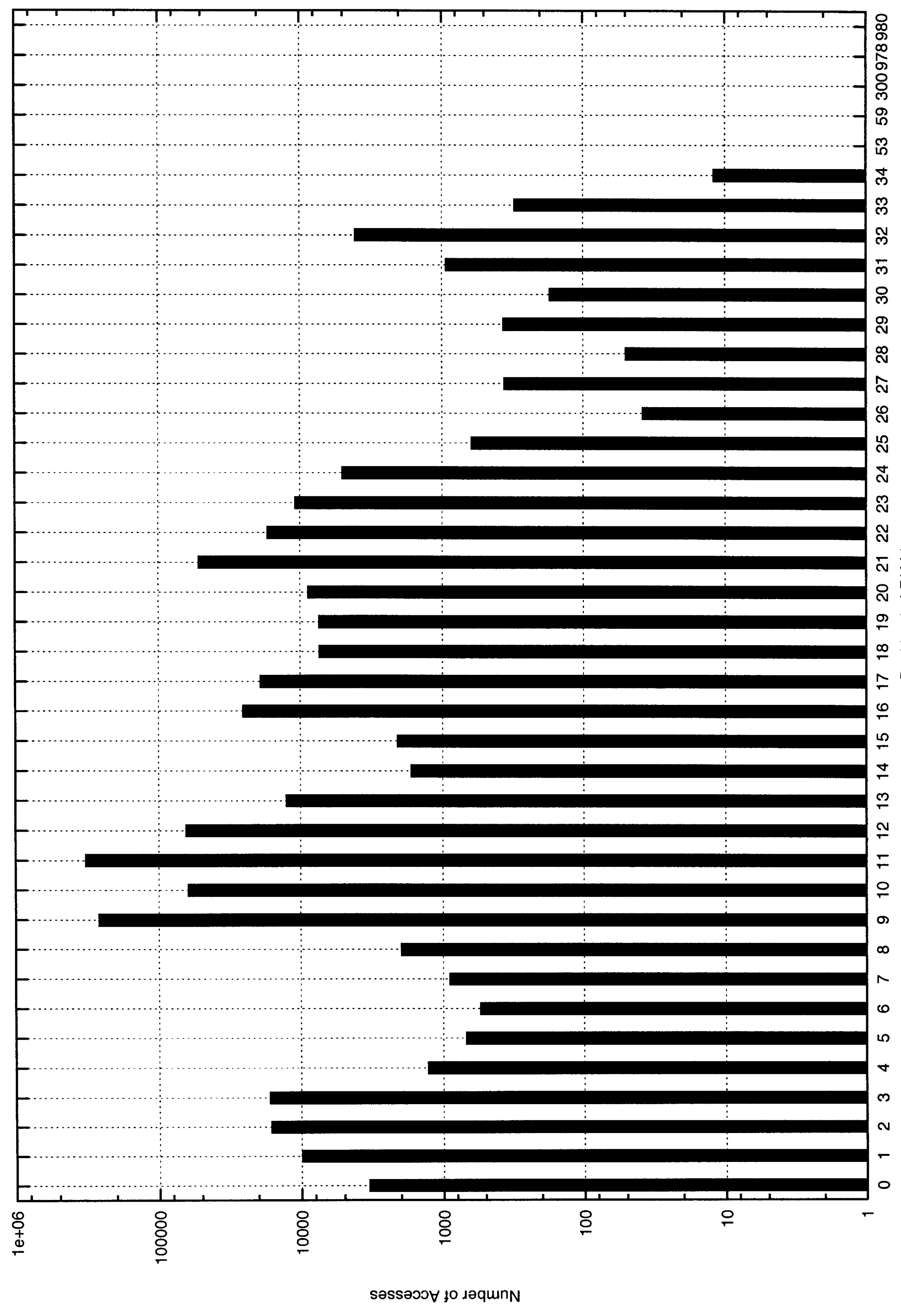

Figure 6-8: Backlist of Squid in Fix Mode with 10 parallel clients

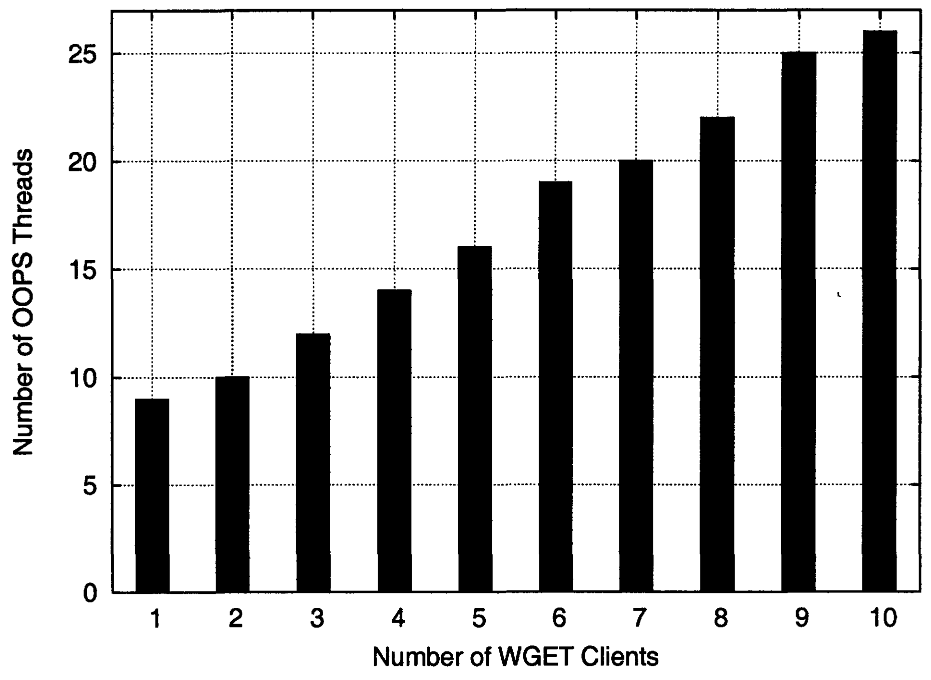

Figure 6-9: Number of OOPS threads

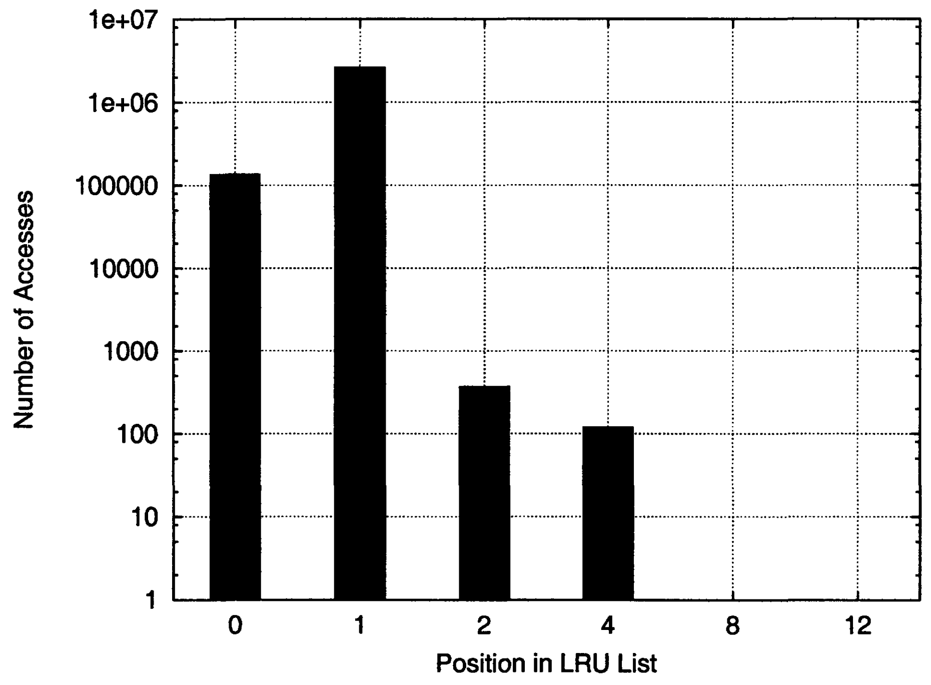

Figure 6-10: Backlist of OOPS in Monitor Mode with 10 serial clients

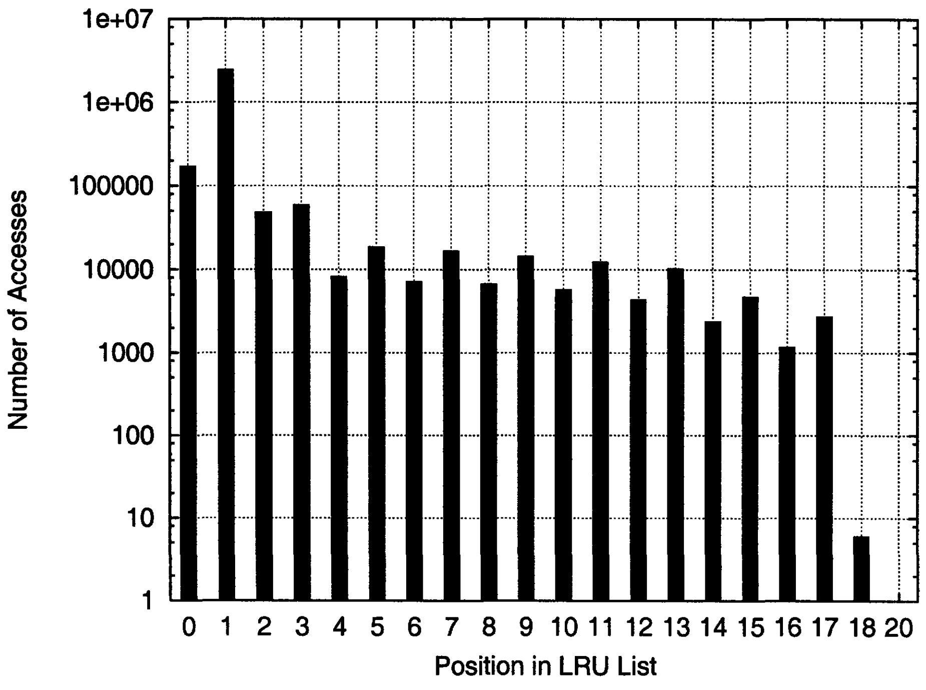

Figure 6-11: Backlist of OOPS in Monitor Mode with 10 parallel clients

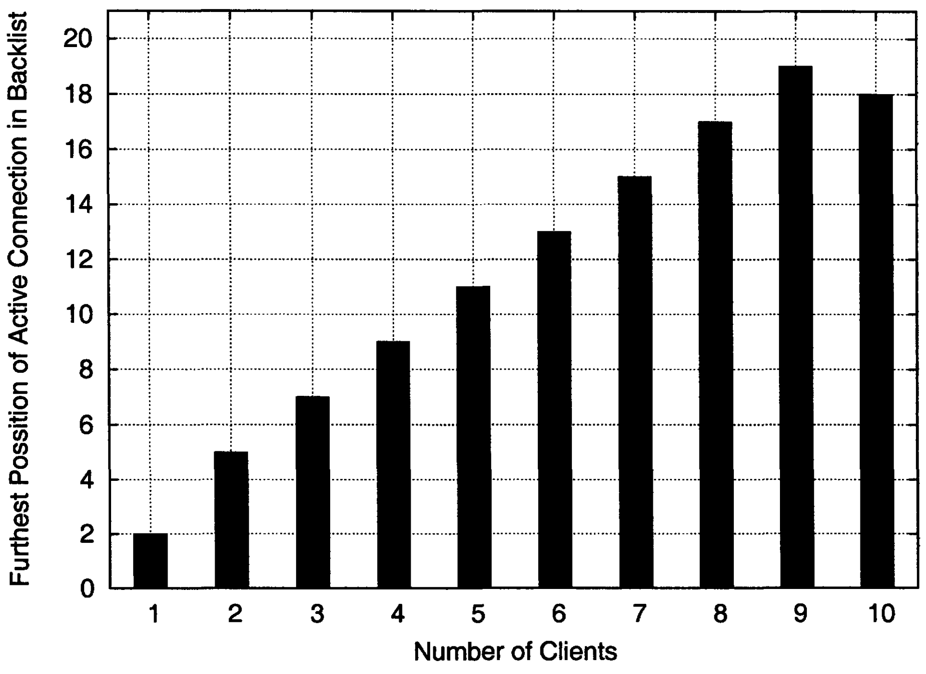

Figure 6-12: Number of FDs needed by OOPS

ID	Function	Number	Description
1	open	5	reserved FDs (opened /dev/null)
2	fopen	1	opened oops.cfg
3	fopen	3	opened configuration files
4	open	1	opened error template file
5	open	1	opened oops/passwd
6	open	1	opened authentication template file
7	open	1	opened authentication template file
8	fopen	1	opened file with redirection rules
9	open	1	opened file with redirection templates
10	open	1	opened oops/logs/oops.pid
11	fopen	1	opened /etc/passwd
12	open	1	opened oops/logs/oops.log
13	open	1	opened oops/logs/access.log
14	fopen	1	opened /etc/passwd
15	open	0	FD for storage (not used)
16	fopen	1	opened /etc/passwd
17	socket	1	listening socket
18	socket	1	listening socket
19	open	5	reserved FDs (opened /dev/null)
20	socket	1	listening socket
21	socket	1	listening socket
22	fopen	1	opened /etc/passwd
23	accept	10280	accepted connections
24	socket	1102	connections to HTTP servers
25	socket	9178	connections to HTTP servers
26	fopen	169	opened oops/logs/oops_statfile

Table 6.3: FD Allocation Sites for OOPS

by each site was recorded when OOPS was used by 10 clients concurrently. Most of the sites open only one file descriptor which is associated with a configuration file. Sites 23, 24 and 25 are used for active connections. Sites 17, 18, 20 and 21 create sockets for listening for connections. Site 26 is always used to write statistics. Sites 1 and 19 are interesting. OOPS reserves a few FDs by opening */dev/null*. Initially, OOPS reserved 25 FDs at sites 1 and 19 (FDs are allocated at a site after the FDs from the previous site are closed), but we had to change the value from 25 to 5 in order for our heuristic to work. This issue will be discussed in more detail when we analyze OOPS in Fix Mode.

6.3.2 OOPS in Leak Mode

We ran OOPS in Leak Mode with 10 parallel clients, and then with 10 serial clients. In the case of parallel clients, 8 clients downloaded 48 files and 2 clients downloaded 49 files. The total number of files successfully downloaded was 482. For serial clients, the first client downloaded 491 files, and the other clients were not able to download any files.

In Leak Mode, after the entire pool of file descriptors is used, the system calls to *accept* fail and connections are dropped.

6.3.3 OOPS in Fix Mode

When we first ran OOPS in Fix Mode, our safety check caused the program to exit because our heuristic was closing a file descriptor that was still in use. We traced the file descriptor to allocation site 1. We analyzed the code of OOPS, and we concluded that OOPS reserves FDs by mapping them to */dev/null*. When OOPS needs some FDs (for accessing logs), it closes its pool of reserved FDs. Basically, this measure is taken in order to ensure that OOPS always has some file descriptor available. OOPS would rather drop connections than not be able to read configuration files and write logs. The interesting thing is that the file descriptors are never used (except for open and close). Our heuristic would work correctly by closing the file descriptor sooner and reusing it, and later ignoring the call to close. The problem is that we would have a file descriptor that is overloaded. In a future version, we intend to deal with this shortcoming by creating a virtual layer of file descriptors. In order to run the experiments with the current setup, we changed the source code of OOPS to reserve 5 (as opposed to 25) file descriptors.

We ran OOPS in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to download all the files.

A backlist with 10 parallel clients is presented in figure 6-8. Active connections spread out only to position 23 in the LRU list. Accesses to positions 25 though 568 refer only to FDs bound to *oops.log*.

6.4 TinyProxy

We first ran TinyProxy in Monitor Mode to observe FD usage patterns. After that, we ran TinyProxy in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran TinyProxy in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.4.1 TinyProxy in Monitor Mode

We ran TinyProxy in Monitor Mode with different numbers of clients downloading our site using an Apache server and a TinyProxy proxy. We ran the tests using clients running concurrently and clients running serially. We noticed that for serial clients, TinyProxy always creates 12 processes: two setup processes that do not handle user requests and 10 worker processes that serve user requests. In the case of parallel clients, as shown in figure 6-14, the number of processes tends to grow linearly as the number of clients increases.

Backlists, for both serial and parallel clients, go back only to position 2 in each process. This is natural because each process handles one connection at a time, and there are 3 file descriptor needed for each connection: the listening socket, the socket created by accept when a client connects to the proxy and a socket used by the proxy to connect to the HTTP server.

Table 6.4 shows a list of allocation sites for TinyProxy. Sites 1, 2 and 3 are used for logging and reading configuration files. Site 4 is used to create the listening socket. Sites 5, 7, 17 and 19 are used for accepting user connections and connecting to HTTP servers. All the remaining sites open */etc/hosts* and are called within the *gethostbyaddr* function. Sites are listed separately because the library containing *gethostbyaddr* is loaded at different addresses in different processes.

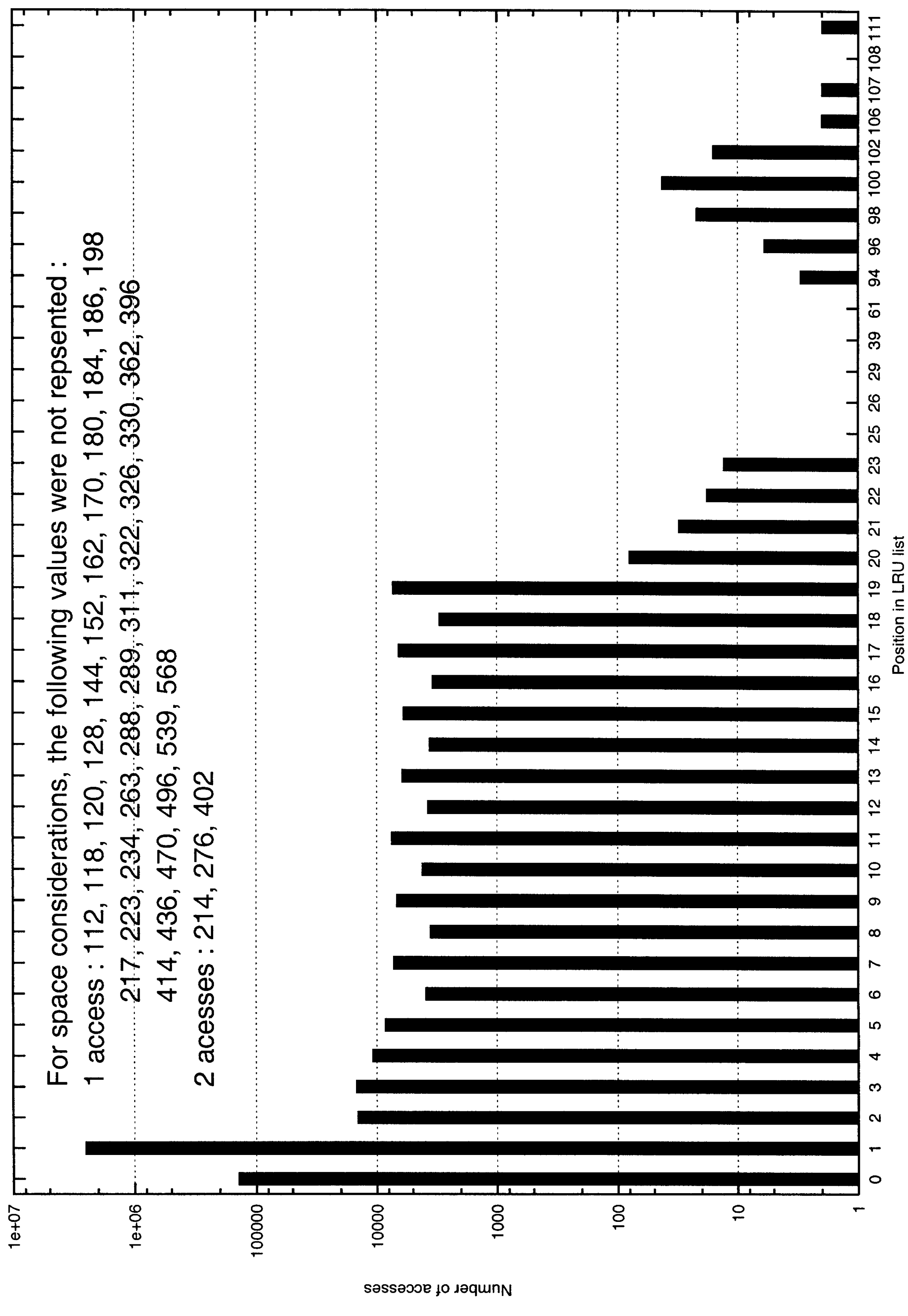

Figure 6-13: Backlist of OOPS in Fix Mode with 10 parallel clients

ID	Function	Number	Description
1	fopen	1	opened etc/tinyproxy/tinyproxy.conf
2	open	1	opened var/log/tinyproxy.log
3	open	1	opened var/run/tinyproxy.pid
4	socket	1	listening socket
5	accept	7508	accepted client connections
6	fopen	755	opened /etc/hosts (by gethostbyaddr)
7	socket	7508	connections to HTTP server
8	fopen	730	opened /etc/hosts (by gethostbyaddr)
9	fopen	749	opened /etc/hosts (by gethostbyaddr)
10	fopen	758	opened /etc/hosts (by gethostbyaddr)
11	fopen	774	opened /etc/hosts (by gethostbyaddr)
12	fopen	752	opened /etc/hosts (by gethostbyaddr)
13	fopen	771	opened /etc/hosts (by gethostbyaddr)
14	fopen	736	opened /etc/hosts (by gethostbyaddr)
15	fopen	743	opened /etc/hosts (by gethostbyaddr)
16	fopen	740	opened /etc/hosts (by gethostbyaddr)
17	accept	2772	accepted client connections
18	fopen	670	opened /etc/hosts (by gethostbyaddr)
19	socket	2772	connections to HTTP server
20	fopen	1117	opened /etc/hosts (by gethostbyaddr)
21	fopen	428	opened /etc/hosts (by gethostbyaddr)
22	fopen	371	opened /etc/hosts (by gethostbyaddr)
23	fopen	121	opened /etc/hosts (by gethostbyaddr)
24	fopen	65	opened /etc/hosts (by gethostbyaddr)

Table 6.4: FD Allocation Sites for TinyProxy

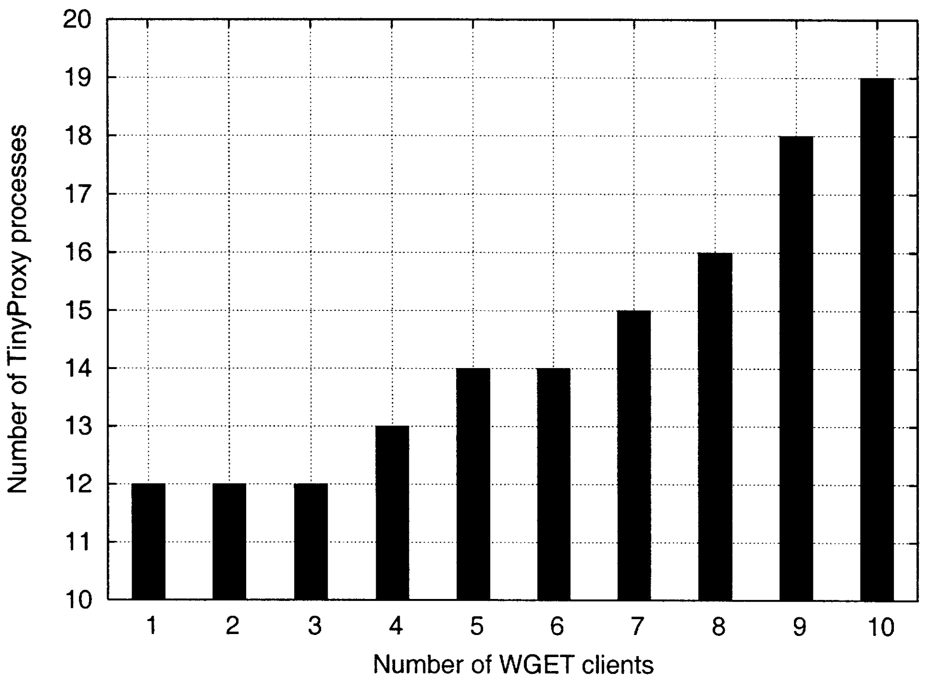

Figure 6-14: Number of TinyProxy processes

6.4.2 TinyProxy in Leak Mode

We ran TinyProxy in Leak Mode with 10 parallel clients, and then with 10 serial clients. The number of files downloaded by the parallel clients is displayed in figure 6-15. Clients managed to download around 400 files each, for a total of 4235. In the case of serial clients, the number of downloaded files is presented in figure 6-16. The first three clients managed to download all 1027 files. The fourth client downloaded 275 files and the rest of the clients did not download any files. The total number of downloaded files was 3356. More files were downloaded with parallel clients because TinyProxy forked more worker processes.

6.4.3 TinyProxy in Fix Mode

We ran TinyProxy in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to download all the files. All accesses were made to FDs on positions 0, 1, 2, 3 and 4 of the LRU list.

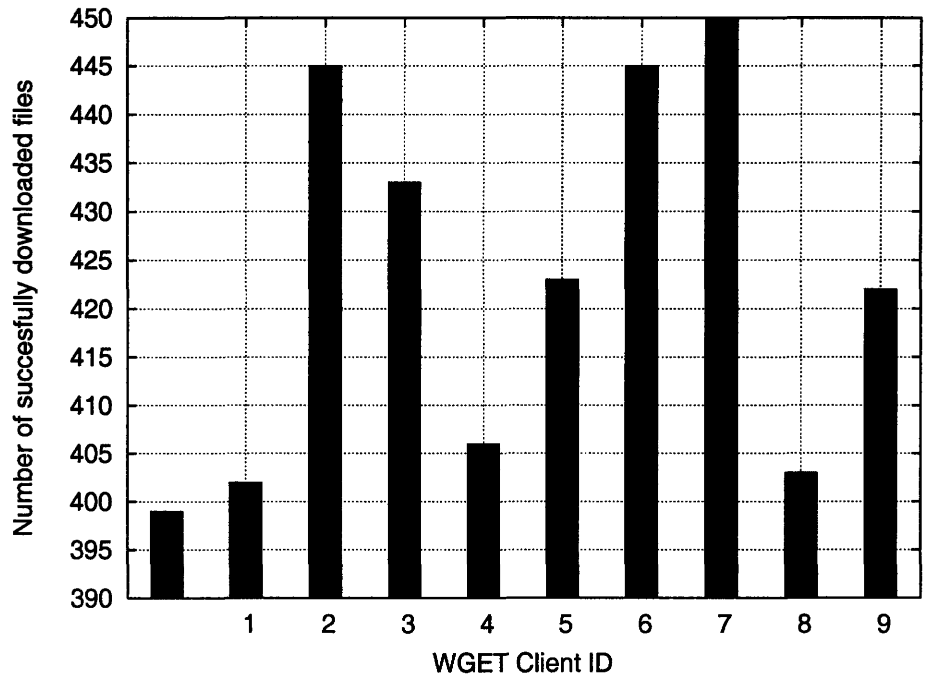

Figure 6-15: Succesful downloads with TinyProxy under Leak Mode (parallel clients)

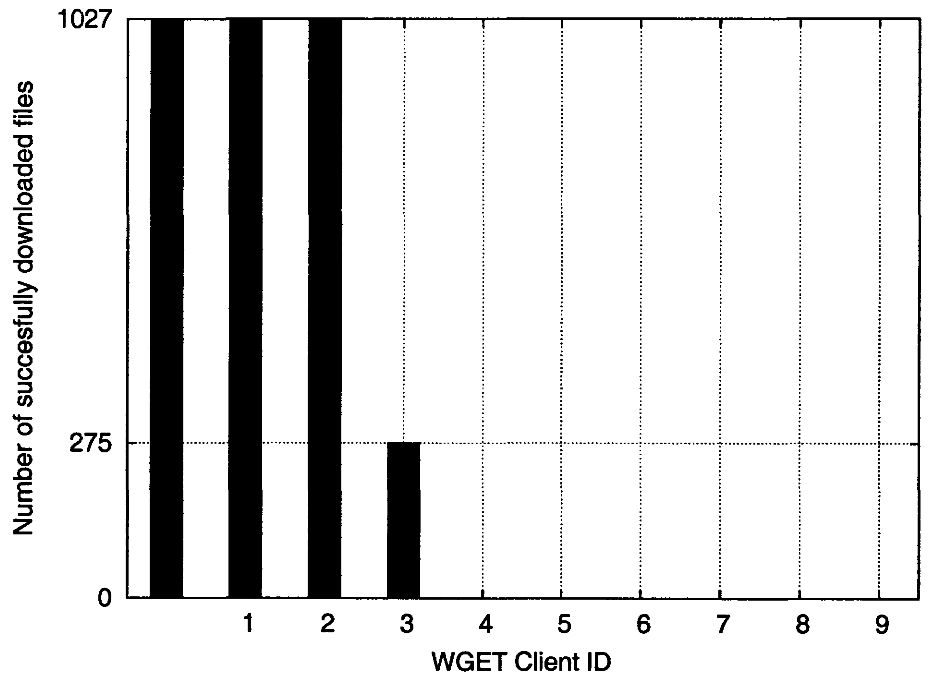

Figure 6-16: Succesful downloads with TinyProxy under Leak Mode (serial clients)

6.5 Hydra

We first ran Hydra in Monitor Mode to observe FD usage patterns. After that, we ran Hydra in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran Hydra in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.5.1 Hydra in Monitor Mode

We noticed that, regardless of the workload, Hydra always creates two processes. The first process has a single thread and it does not serve client requests. The second process always has 4 threads (regardless of the workload) that all serve client requests. Hydra uses the *poll* libc function for asynchronous IO.

When serving serial clients, the backlist for Hydra always stops at position 3, regardless of the number of clients used. In the case of concurrent clients, the position of furthest access in the LRU list grows linearly with the number of clients, as illustrated in figure 6-17. For Hydra, the full length of backlists can be accounted for by active connections.

Table 6.5 shows a list of allocation sites for Hydra. The number of FDs created at each site was recorded when Hydra was used by 10 clients concurrently. Sites 1 and 2 are used to map *standard output* to */dev/null*. Sites between 3 and 11 are used for configuration and log files. Site 12 creates a socket for listening for client connections. Sites 13, 14, 16 and 17 create sockets for handling user requests. Finally, sites 15 and 18 read files requests by clients.

6.5.2 Hydra in Leak Mode

We ran Hydra in Leak Mode with 10 parallel clients, and then with 10 serial clients. The parallel clients managed to successfully download between 76 and 154 files each (figure 6-18). The total number of files downloaded by the parallel clients is 998. In the case of serial clients, the first client downloaded 1006 files while all other clients did not download any files.

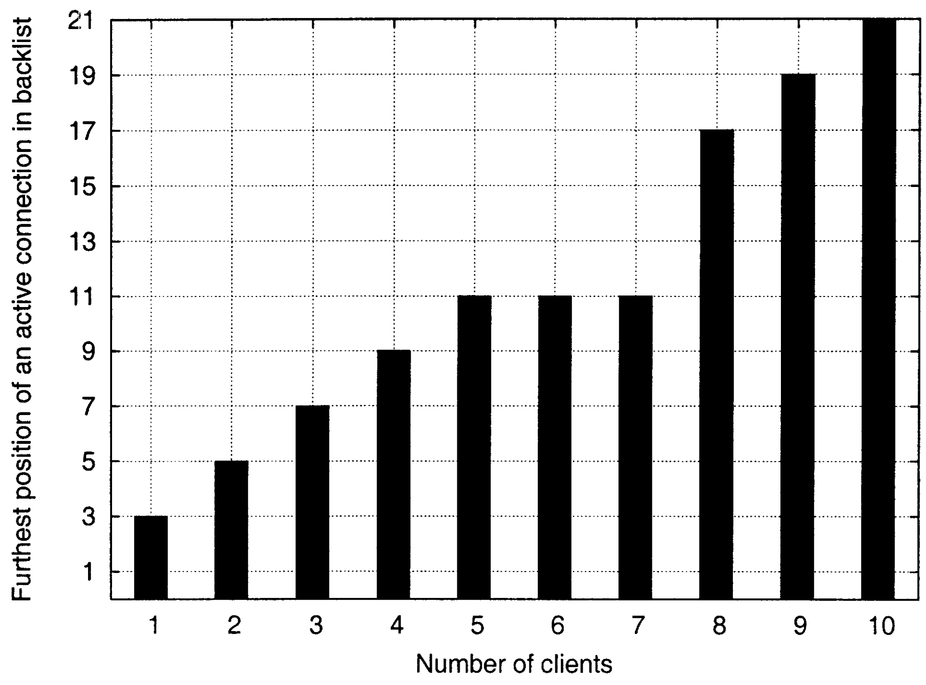

Figure 6-17: Number of FDs needed by Hydra

ID	Function	Number	Description
1	open	1	opened /dev/null
2	dup	1	dup of /dev/null
3	fopen	1	opened hydra.conf
4	fopen	1	opened /etc/passwd
5	fopen	1	opened /etc/group
6	fopen	1	opened /etc/mime.types
7	fopen	1	opened /etc/hosts
8	open	1	opened /var/log/hydra/error_log
9	dup	1	dup of /var/log/hydra/error_log
10	open	1	opened /var/log/hydra/access_log
11	dup	1	dup of /var/log/hydra/access_log
12	socket	1	listening socket
13	accept	22	user requests
14	accept	7	user requests
15	open	10204	files for user requests
16	accept	2	user requests
17	accept	0	user requests
18	open	62	files for user requests

Table 6.5: FD Allocation Sites for Hydra

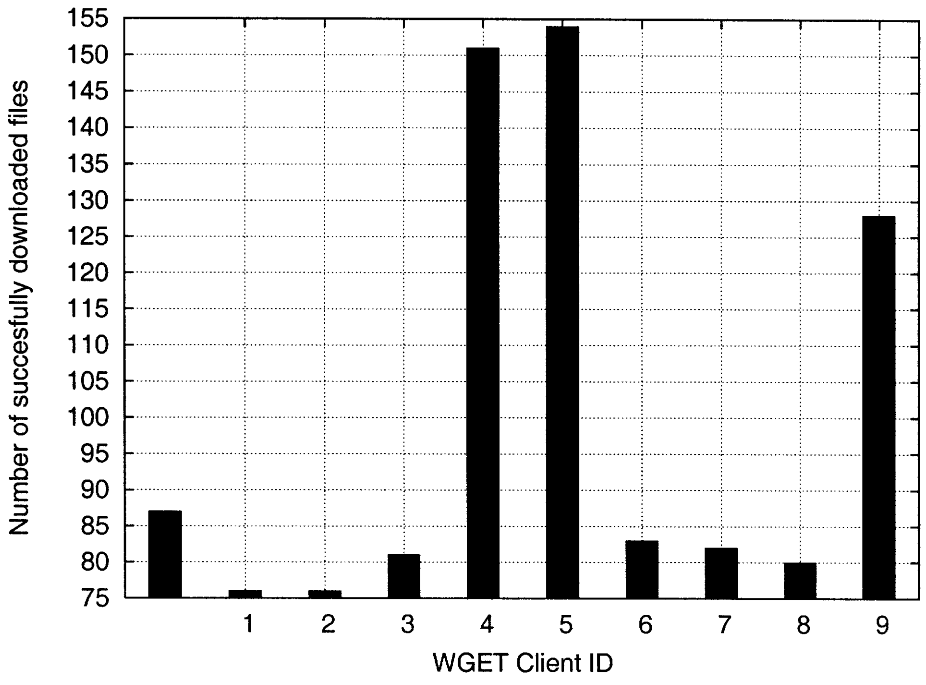

Figure 6-18: Succesful downloads with Hydra under Leak Mode

In Leak Mode, after the entire pool of file descriptors is used, the system calls to *accept* fail and connections are dropped.

6.5.3 Hydra in Fix Mode

We ran Hydra in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to download all the files.

A backlist with 10 parallel clients is presented in figure 6-19. Active connections spread out as far as position 70.

6.6 Pure-FTPd

We first ran Pure-FTPd in Monitor Mode to observe FD usage patterns. After that, we ran Pure-FTPd in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran Pure-FTPd in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

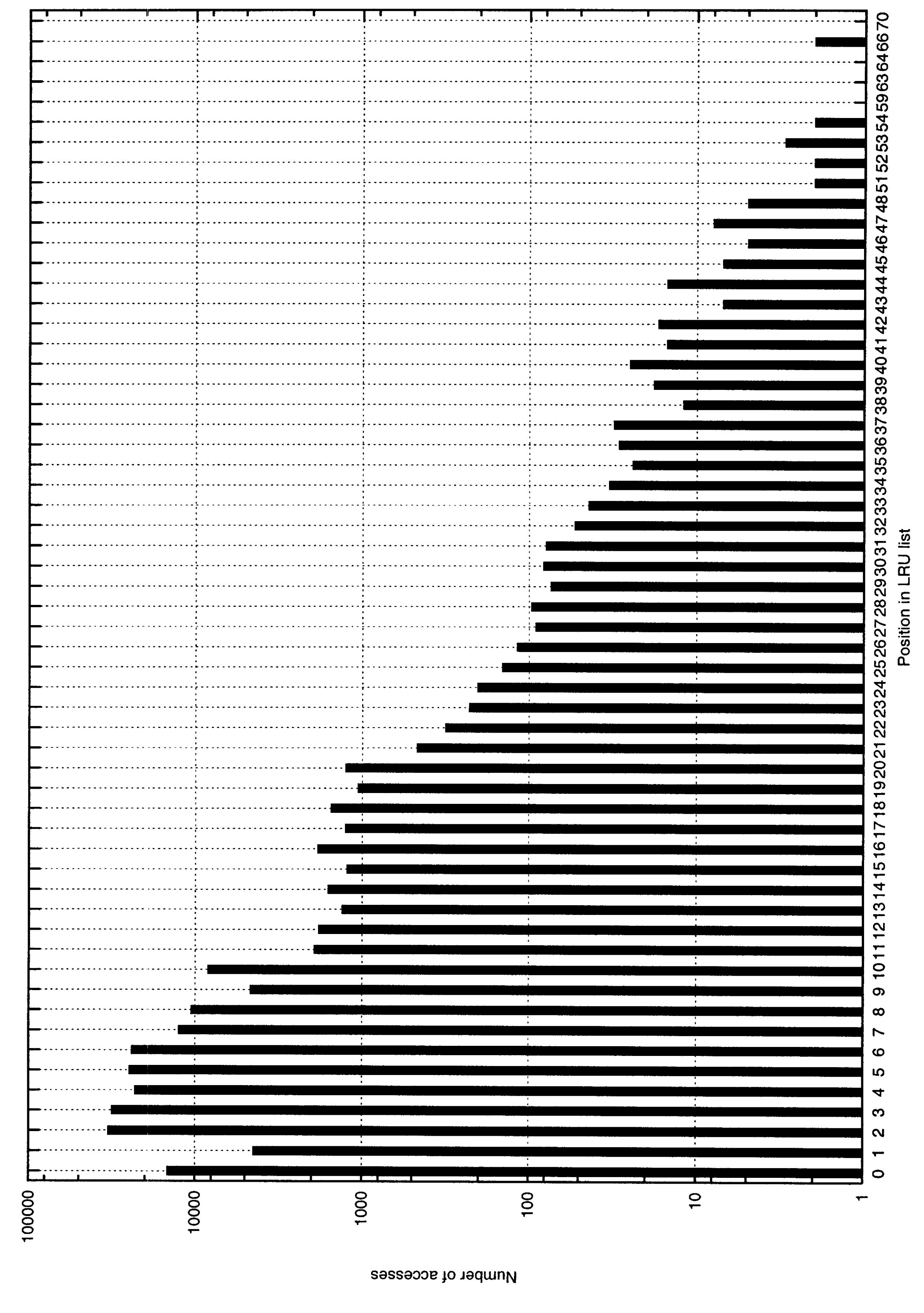

Figure 6-19: Backlist of Hydra in Fix Mode with 10 parallel clients

6.6.1 Pure-FTPd in Monitor Mode

Even before running our experiments, we noticed that Pure-FTPd is a multi-process application that has one thread per process. The first process listens for connections, and starts children processes to handle connections.

We ran repeated experiments with different numbers of WGET clients serially and concurrently downloading our test site through a Pure-FTPd server. Unlike any previous application, for a fixed number of clients, Pure-FTPd created the same number of processes regardless of whether the clients were running serially or concurrently. The number of processes needed is shown in figure 6-20. As the figure shows, Pure-FTPd has a main processes that listens for connections and starts a new process for each connection.

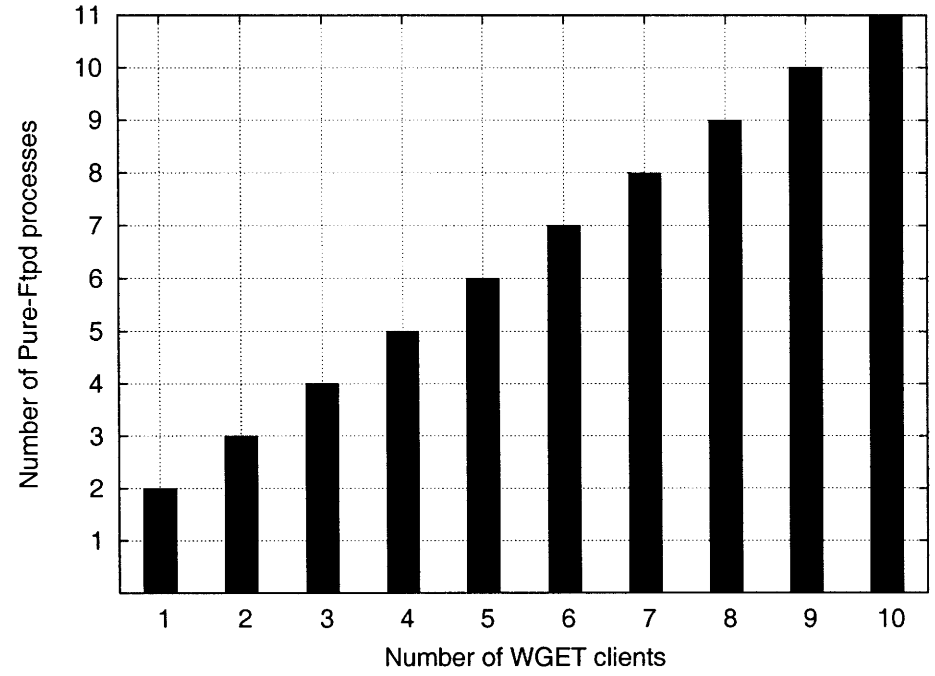

Figure 6-20: Number of Pure-FTPd processes

Table 6.6 presents a list of FD allocation sites together with the function used to allocate the FDs, the number of FDs created at each site and a brief description of what the FDs are used for. The data was obtained using 10 parallel WGET clients.

Under all workloads, the backlist for the first process is very short: accesses occur

ID	Function	Number	Description
1	socket	1	unused socket
2	socket	1	listening socket
3	socket	1	listening socket
4	open	1	opened etc/pure-ftpd.pid
5	accept	10	client connection
6	dup	10	dup of a client connection
7	dup	10	dup of (a dup of) a client connection
8	fopen	1	opened /etc/hosts (used by gethostbyaddr)
9	fopen	1	opened /etc/hosts (used by gethostbyaddr)
10	fopen	1	opened /etc/hosts (used by gethostbyaddr)
11	open	10	opened /dev/urandom
12	open	10	opened /dev/urandom
13	fopen	1	opened /etc/hosts (used by gethostbyaddr)
14	fopen	0	failed fopen (looking for banner file)
15	fopen	1	opened /etc/hosts (used by gethostbyaddr)
16	fopen	1	opened /etc/hosts (used by gethostbyaddr)
17	fopen	1	opened /etc/hosts (used by gethostbyaddr)
18	fopen	1	opened /etc/hosts (used by gethostbyaddr)
19	socket	10280	sockets for FTP data connections
20	accept	10	sockets for FTP data connections
21	fopen	1	opened /etc/hosts (used by gethostbyaddr)
22	open	10270	used to read files requested by clients
23	accept	10270	sockets for FTP data connections
24	fopen	1	opened /etc/hosts (used by gethostbyaddr)

Table 6.6: FD Allocation Sites for Pure-FTPd

to FDs only in positions 0 and 1. The backlists for all worker clients were identical among processes used in the same workload, but also among processes used in different workloads. A sample backlist is presented in figure 6-21. As the figure shows, all worker processes have very short backlists.

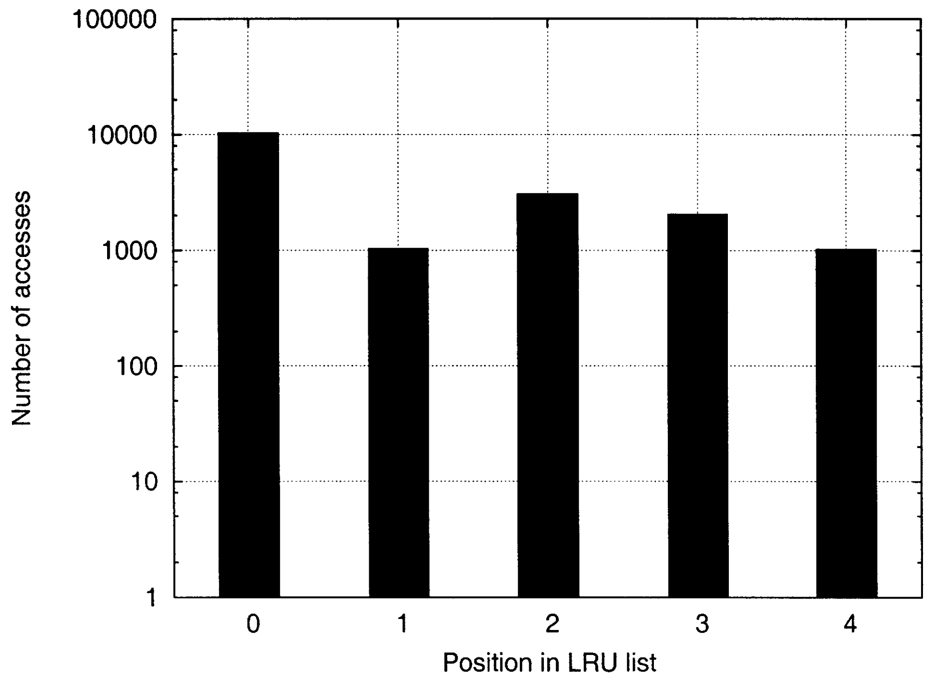

Figure 6-21: Backlist of a Pure-FTPd process in Monitor Mode

6.6.2 Pure-FTPd in Leak Mode

We ran Pure-FTPd in Leak Mode with 10 parallel clients and with 10 serial clients. To our surprize, all the clients downloaded almost all the files. The number of files downloaded by each parallel client is shown is figure 6-23 and the number of files downloaded by serial clients is presented in figure 6-22.

We looked at the number of processes created and we have noticed that, instead of 25 processes created by Pure-FTPd in Monitor Mode, Pure-FTPd in Leak Mode created 50 processes. We concluded that processes that ran out of file descriptors were terminated. As each client lost its connection to the server and then reconnected to the main process, that process created another worker process.

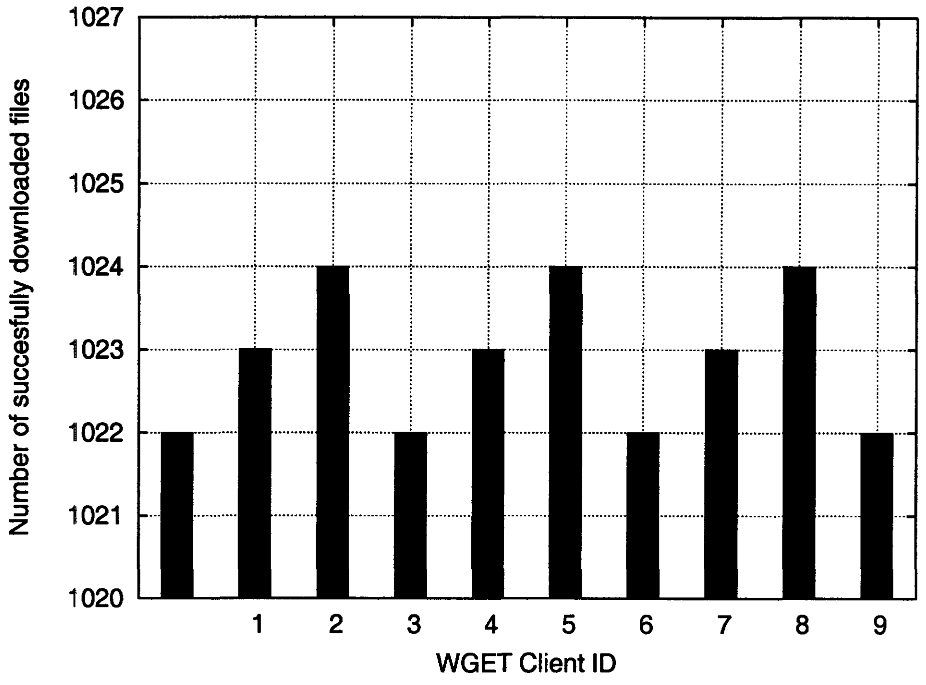

Figure 6-22: Succesful downloads with Pure-FTPd under Leak Mode (serial clients)

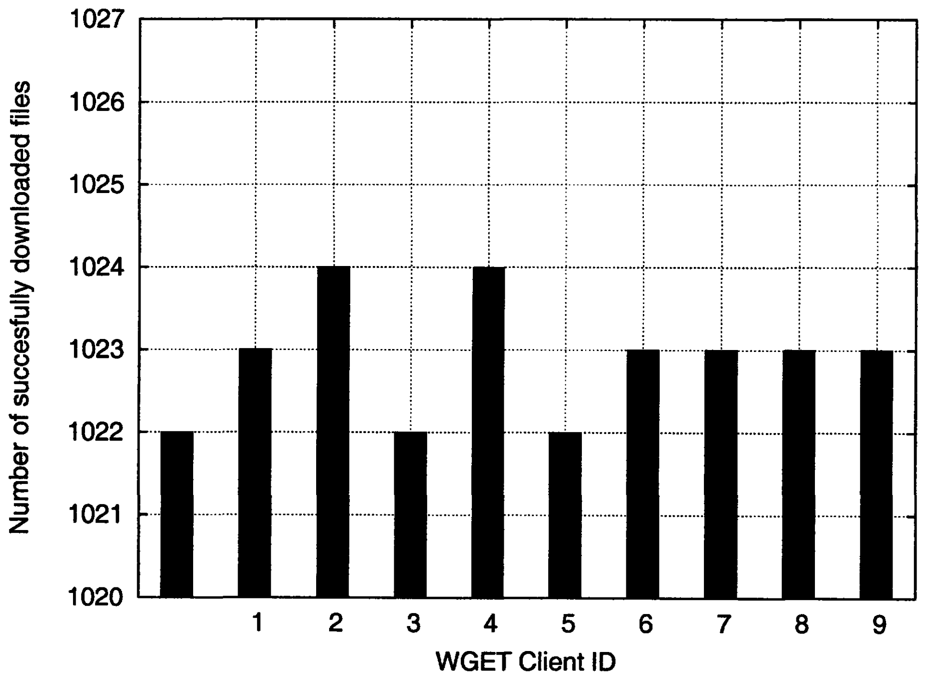

Figure 6-23: Succesful downloads with Pure-FTPd under Leak Mode (parallel clients)

6.6.3 Pure-FTPd in Fix Mode

We tested Pure-FTPd in Fix Mode in two situations: 10 parallel clients and 10 serial clients. In both cases, the clients have successfully managed to download all files, and the number of worker processes created equaled the number of processes created under Monitor Mode.

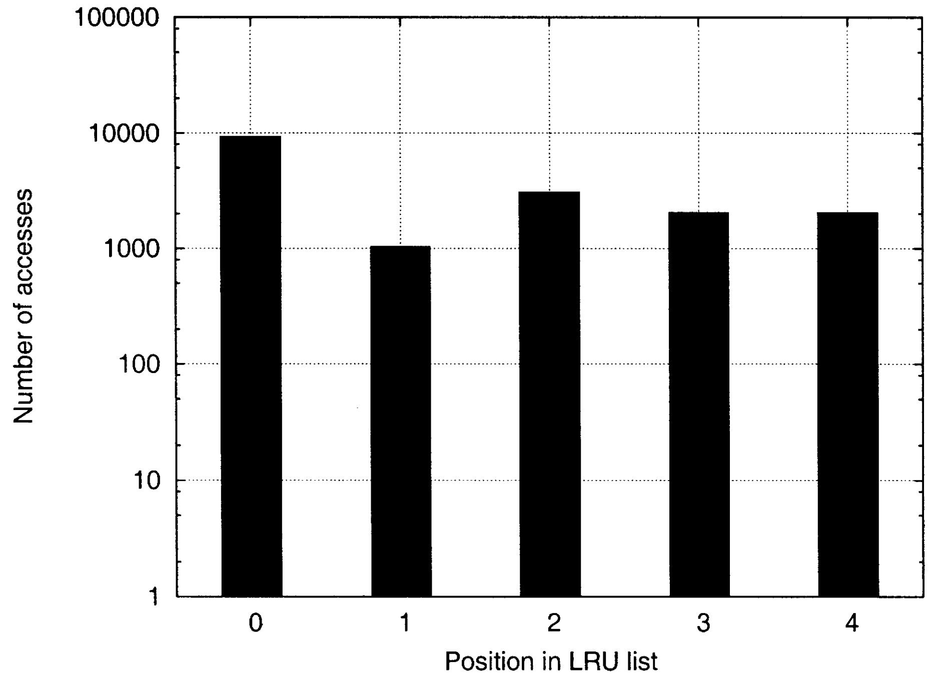

Figure 6-24: Backlist of a Pure-FTPd process in Fix Mode

We also noticed that backlists for all worker processes were identical. A sample backlist is presented in figure 6-24. The backlist is very short. In fact, it is not longer than the backlist for processes in Monitor Mode, however, the number of accesses to further out position is greater under Fix Mode than under Monitor Mode.

Pure-FTPd is the only application that has identical behavior for both serial and parallel clients. Most applications create a pool of worker threads or processes that get reused, thus keeping the number of processes or threads small for serial clients. Unlike other applications, Pure-FTPd terminates processes after they served their clients. Terminating a child process after its connection is handled is easier than keeping a pool of workers, but it is more costly because of the overhead associated with creating new processes.

6.7 Thttpd

We first ran Thttpd in Monitor Mode to observe FD usage patterns. After that, we ran Thttpd in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran Thttpd in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.7.1 Thttpd in Monitor Mode

We ran Thttpd in Monitor Mode with a different number of clients downloading our site. We ran the tests using clients running concurrently and clients running serially.

Thttpd always creates exactly two processes. The first process just starts the second process and exists. The main purpose of the first process is to prevent applications starting and waiting for Thttpd from blocking. The second process has a single thread, listens for connections and handles all client requests. Thttpd uses the libc function *poll* for asynchronous IO.

Thttpd had a very short backlist when it is used by serial clients. All FDs accessed were on positions 0 and 1 of the LRU list. The size of the backlist is longer for parallel clients. A backlist for 10 parallel clients is shown in figure 6-25. We looked at the sizes of backlists for different number of clients, and we concluded that, as shown in figure 6-26, the length of the backlist was equal to the number of clients. For other single-process applications, the length of the backlist was linear in the number of clients, but the proportionality constant was higher.

Table 6.7 shows a list of allocation sites for Thttpd. The number of FDs created at each site was measured with 10 parallel clients. Thttpd is a very simple application and it has a very small number of allocation sites. Thttpd created two listening sockets: one for IPv4 and one for IPv6 (the two sockets are created at sites 1 and 2). Site 3 created 10280 sockets used to serve files to clients (we used 10 clients,

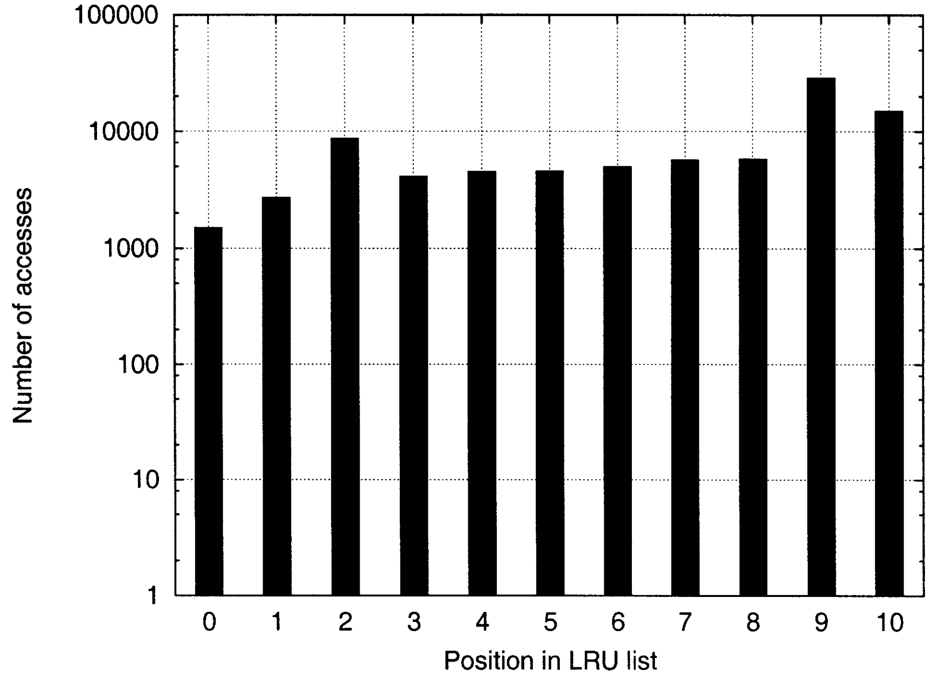

Figure 6-25: Backlist of Thttpd in Monitor Mode with 10 parallel clients

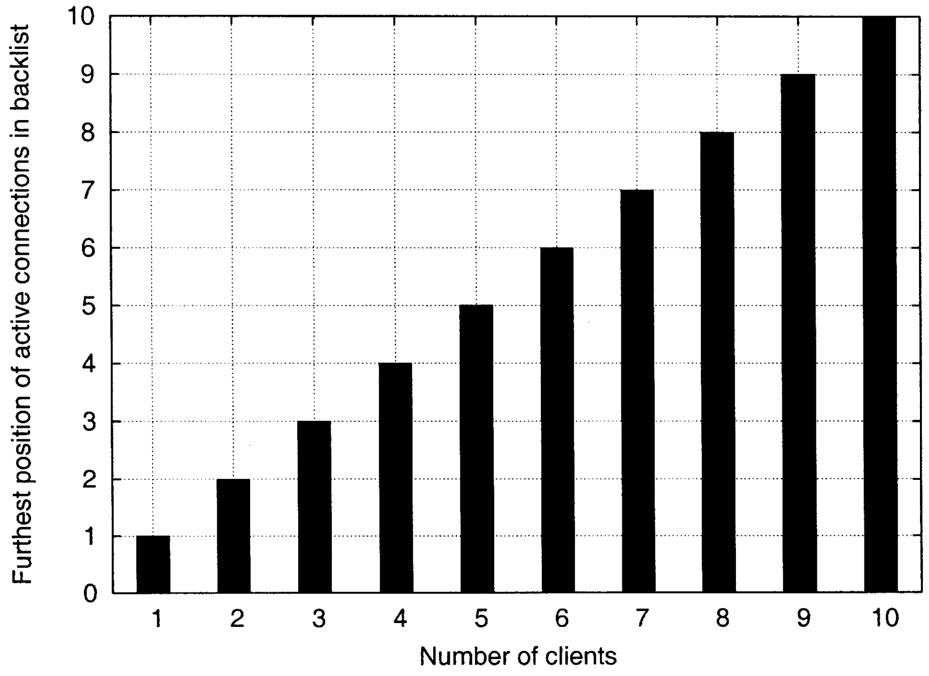

Figure 6-26: Number of FDs needed by Thttpd

ID	Function	Number	Description
1	socket	1	listening socket
2	socket	1	socket for IPv6
3	accept	10280	client connections
4	open	1027	files requested by clients
5	fopen	0	missing error file

Table 6.7: FD Allocation Sites for Thttpd

each downloading 1027 files[1]). Site 4 was used to read the files requested by clients. Unlike other tested HTTP servers, Thttpd cached the contents of the files already downloaded, so each file was read only once. The last site tried to open an error file (which did not exist). The error was generated when *WGET* tried to download *robots.txt*.

6.7.2 Thttpd in Leak Mode

We ran Thttpd in Leak Mode with 10 parallel clients, and then with 10 serial clients. In the case of serial clients, the first client managed to download 506 files (roughly $\frac{1024}{2}$). For parallel clients, clients downloaded between 89 and 98 files each (the exact number of files is presented in figure 6-27). The total number of files downloaded by parallel clients is 916. This number is considerable larger (almost double) than the number of files downloaded by the serial clients. The reason is that the parallel clients downloaded the same files, so fewer FDs were needed to read files from disk. In the case of serial clients, 506 FDs were created at site 4, while only 98 FDs were created at site 4 for parallel clients.

In Leak Mode, after the entire pool of FDs is used, the system calls to *accept* fail and connections are dropped.

6.7.3 Thttpd in Fix Mode

We ran Thttpd in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to

[1]WGET also tries to retrieve *robots.txt* which cannot be found.

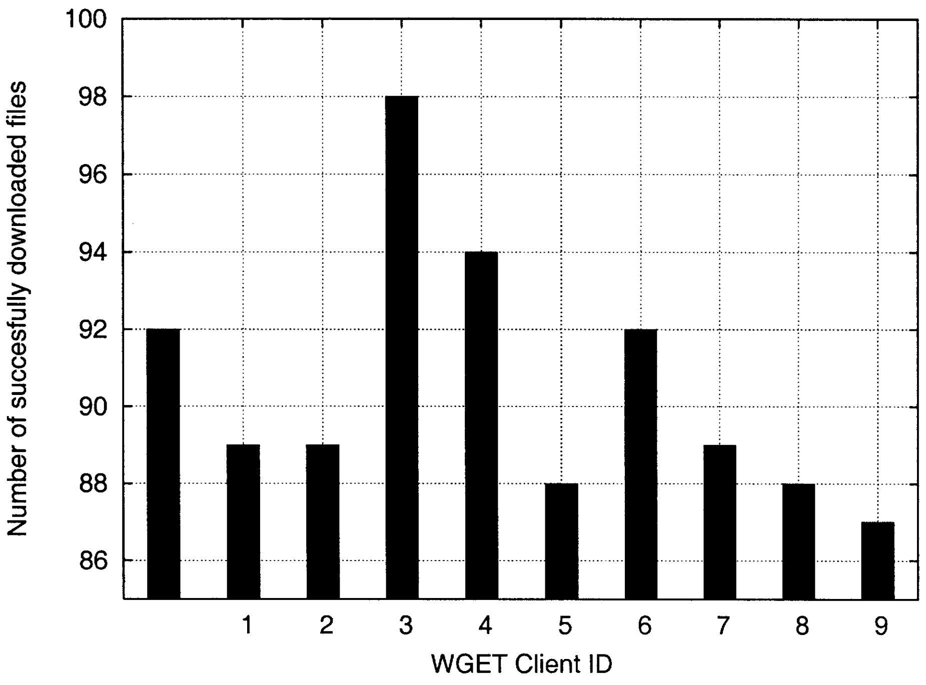

Figure 6-27: Succesful downloads with Thttpd under Leak Mode

download all the files.

A backlist with 10 parallel clients is presented in figure 6-28. The backlist is only one position longer than the backlist measured in Monitor Mode.

6.8 WebFS

We first ran WebFS in Monitor Mode to observe FD usage patterns. After that, we ran WebFS in Leak Mode to observe how a massive FD leak would affect clients. In the end, we ran WebFS in Fix Mode to see if our library could improve the behavior of the application in the presence of FD leaks.

6.8.1 WebFS in Monitor Mode

We ran WebFS in Monitor Mode with different numbers of clients downloading our site. We ran the tests using clients running concurrently and clients running serially. We found that WebFS has many similarities with Thttpd.

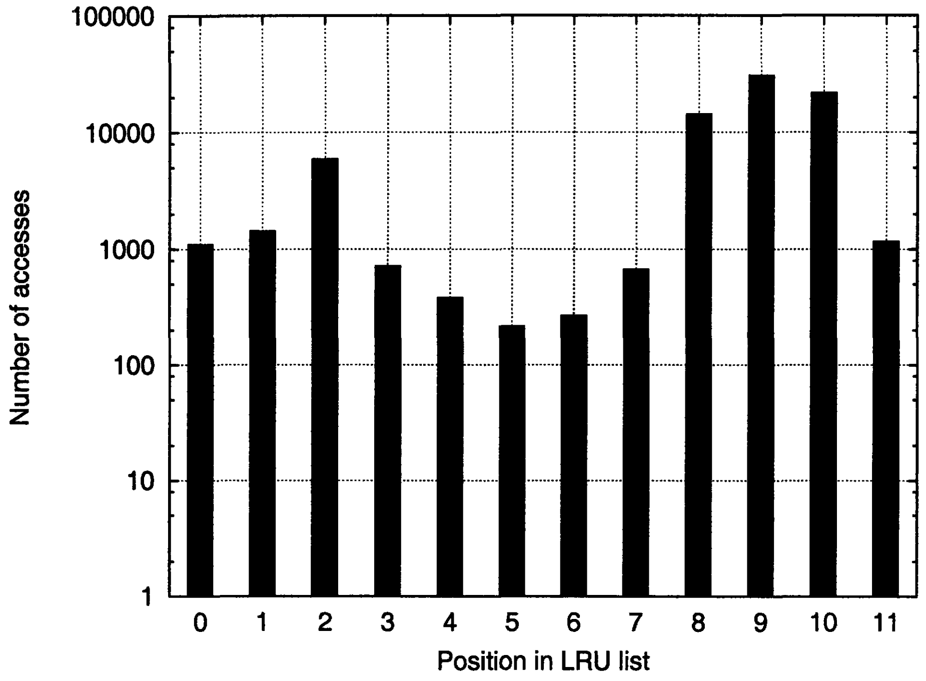

Figure 6-28: Backlist of Thttpd in Fix Mode with 10 parallel clients

WebFS always created exactly two processes. The first process just starts the second process and exists. The second process has a single thread, it listens for connections and handles all client requests. WebFS uses the libc function *select* for asynchronous IO.

Like Thttpd, WebFS has a very short backlist when it is used by serial clients. All FDs accessed are on positions 0 and 1 of the LRU list. The size of the backlist is longer for parallel clients. A backlist for 10 parallel clients is shown in figure 6-29. We looked at the sizes of backlists for different numbers of clients, and we concluded that, as shown in figure 6-30, the length of the backlist is equal to (or slightly smaller than) twice the number of clients.

Table 6.8 shows a list of allocation sites for Thttpd. The number of FDs created at each site was measured with 10 parallel clients. Like Thttpd, WebFS is a simple application with a low number of allocation sites. Site 1 is used by WebFS to obtain information about the host computer. Sites 2 and 3 are used to get information about clients. */etc/hosts* is opened twice: once by *gethostbyname2* and once by *gethostbyaddr*. It is always a good policy to call *gethostbyaddr* before *gethostbyname2*

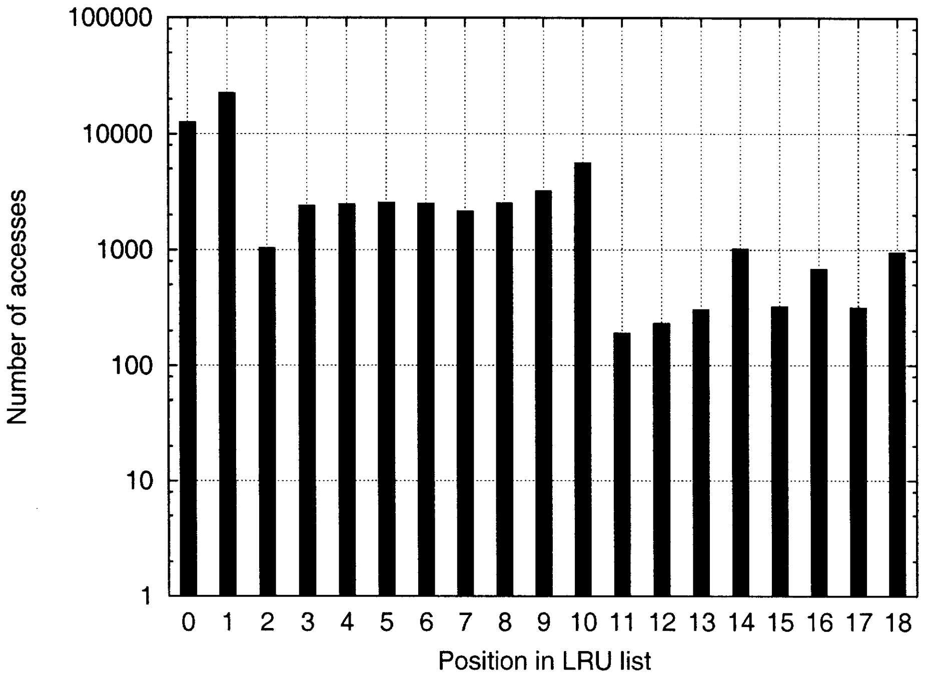

Figure 6-29: Backlist of WebFS in Monitor Mode with 10 parallel clients

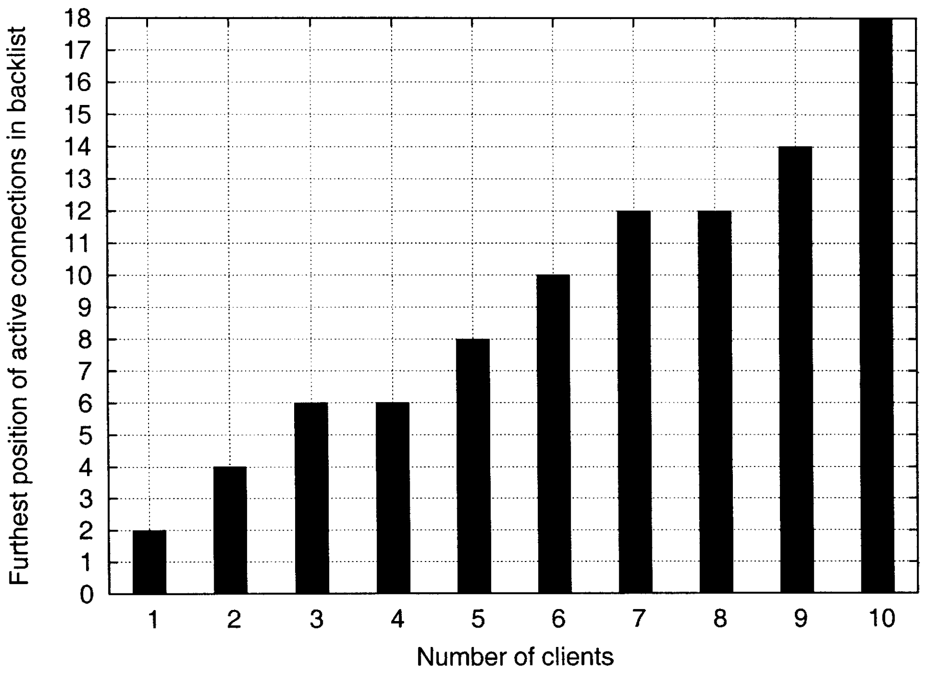

Figure 6-30: Number of FDs needed by WebFS

ID	Function	Number	Description
1	fopen	1	opened /etc/hosts (used by gethostbyname2) to get HTTP host information
2	fopen	1	opened /etc/hosts (used by gethostbyname2) to get client information
3	fopen	1	opened /etc/hosts (used by gethostbyaddr) to get client information
4	socket	1	listening socket
5	fopen	1	opened /etc/mime.type
6	fopen	1	opened /etc/passwd
7	fopen	1	opened /etc/group
8	accept	20	client connections
9	open	10270	files requests by clients

Table 6.8: FD Allocation Sites for WebFS

even-though *gethostbyname2* also works for addresses, because *gethostbyname* is much slower.

Site 4 creates the listening socket, and sites 5, 6 and 7 are used to read configuration files and information about the user running the WebFS daemon. Site 5 is used to create sockets that handle user connections. The number of such sockets is twice the number of clients and it is much smaller than the number of files requests. Unlike Thttpd, WebFS keeps HTTP connections alive. Site 9 is used to read requested files. It is easy to see that WebFS does not cache files.

6.8.2 WebFS in Leak Mode

We ran WebFS in Leak Mode with 10 parallel clients and with 10 serial clients. In the case of serial clients, the first client managed to download 1008 files and the rest of the clients did not download any files. For parallel clients, clients downloaded between 61 and 119 files each (the exact number of files is presented in figure 6-31). The total number of files downloaded by parallel clients is 990, which is slightly less than the number of files downloaded by the serial clients. The difference can be accounted for by the fact that a single socket was used by WebFS for the first serial client, but WebFS had to create one socket for each parallel client.

In Leak Mode, after the entire pool of file descriptors is used, the system calls to

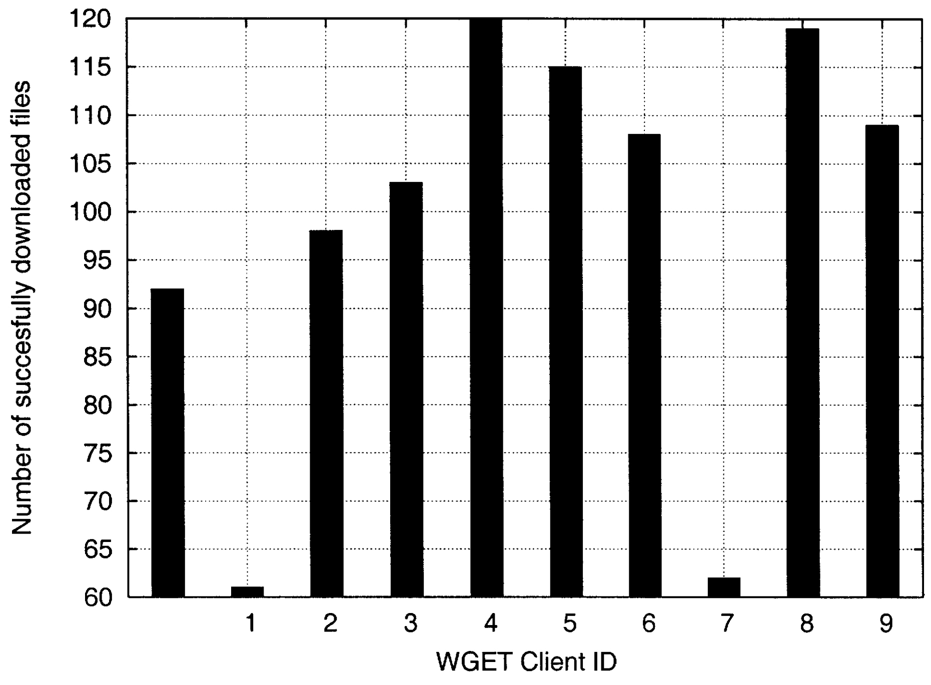

Figure 6-31: Succesful downloads with WebFS under Leak Mode

accept fail and connections are dropped.

6.8.3 WebFS in Fix Mode

We ran WebFS in Fix Mode in the same two cases used for Leak Mode (10 parallel clients and 10 serial clients). In both cases, all the clients successfully managed to download all the files.

A backlist with 10 parallel clients is presented in figure 6-28. The backlist is only two positions longer than the backlist measured in Monitor Mode.

6.9 WGET

WGET is probably the application most suited to our library. WGET downloads a single file at a time. It has a single process and a single thread and WGET handles only a small constant number of file descriptors at a time (the actual number depends on the protocol used).

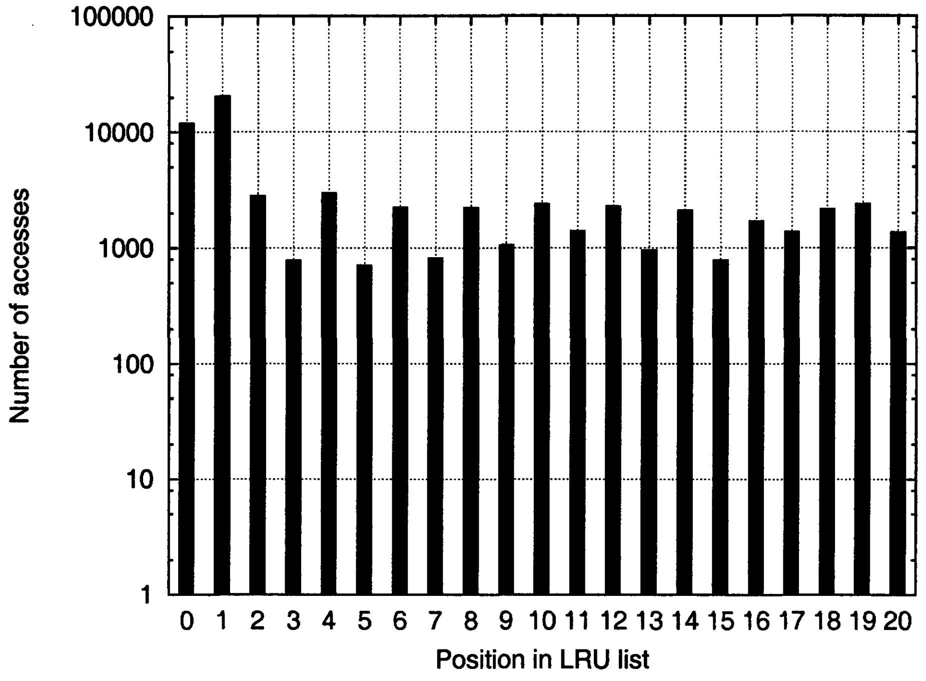

Figure 6-32: Backlist of WebFS in Fix Mode with 10 parallel clients

ID	Function	Number	Description
1	fopen	1	opened /etc/hosts (used by getaddrinfo)
2	fopen	1	opened /etc/hosts (used by getaddrinfo)
3	socket	11	socket used to connect to HTTP server
4	fopen	1027	used to downloaded write file to disk
5	fopen	1	used to read first downloaded file (for links)

Table 6.9: FD Allocation Sites for WGET

We tested WGET by having WGET download our test site from an Apache server. WGET kept the HTTP connections alive. Table 6.9 contains a list of the traces created by WGET. Both file descriptors created at sites 1 and 2 were created by the same single call to *getaddrinfo*.

In Normal Mode, all FD accesses were made to positions 0 and 1 in the LRU list. In Fix Mode, there were accesses to positions 0, 1 and 2. All accesses to position 2 were referring to the communication socket.

WGET in Leak Mode failed to download the entire site. WGET managed to download the first 1004 files after which it could no longer write files to disk. This caused WGET to disconnect from the server. Future connection attempts failed because WGET could not open */etc/hosts*. In Fix Mode, WGET downloaded the entire site.

Chapter 7

Conclusion

We have analyzed nine applications that make extensive use of FDs: four HTTP servers (Apache, Hydra, Thttpd and WebFS), three HTTP proxies (OOPS, Squid and TinyProxy), one FTP server (Pure-FTPd) and one download manager (WGET). For most of these applications, an FD leak is critical: the application eventually exhausts its pool of FDs and can no longer function properly. Our heuristic of recycling unused FDs helps the applications continue running despite leaking FDs.

Throughout our tests, our library has never tried to close an FD that was still in use (with the exception of the reserved FDs in OOPS). While we are aware that our tests cannot cover all possible scenarios, and that closing a FD that is still in use is very dangerous, our tests were performed under extreme conditions: all file descriptors were leaked. In reality, no application can leak that many FDs without developers noticing and correcting the problem. In a real situation, the number of leaked FDs is much smaller, and our library is not forced to close FDs too often, thus further reducing the chance of recycling an FD that is still in use.

Section 7.1 discusses FD allocation, while sections 7.2, 7.3 and 7.4 group the applications according to the their method of handling concurrent IO, and discuss more on FD usage patterns.

7.1 Allocation Sites

Our tests have shown that our heuristic for separating persistent FDs from short-lived FDs works. Most of the allocation sites in our test application either create one (or two in the case of pipes) FDs, or they created a number of FDs proportional to the number of client requests. Furthermore, we have seen that most FD accesses happen to FDs close to the front of the LRU list. The few exceptions (accesses to FDs far back in the LRU list) were always persistent FDs.

We have also seen that applications do not have a lot of allocation sites, so even if we reserve[1] a small number of FDs for each site, the pool of recyclable FDs is still large enough.

7.2 Applications with Multiple Processes

The tested applications that use multiple processes to handle concurrent IO are Apache, TinyProxy and Pure-FTPd. Apache and TinyProxy fork more processes while under heavy load, while keeping the number of processes small while under light load. Pure-FTPd just forks another process for each incoming connection. The child process is terminated when the connection is closed.

These applications have some innate immunity to FD leaks because the application can fork more processes that have fresh pools of FDs. When a process leaks too many FDs and becomes unserviceable, it crashes and the kernel closes all FDs. Multi-process applications have a natural way of getting their FDs recycled.

Multi-process applications can still exhibit problems due to FD leaks because the applications might not fork new worker processes when processes die or because the application creates too many children that leak enough FDs to exhaust the kernel-wide limit.

In multi-process applications, each process usually handles a single connection, so the backlists are extremely short. Recycling FDs in a multi-process application is almost risk-free.

[1]The reserved FDs are un-recyclable.

7.3 Multi-threaded Applications

The only tested application that uses multi-threading to handle concurrent IO is OOPS. Unlike multi-process applications, multi-threaded applications have serious problems in the presence of FD leaks because

- (a) they do not receive new pools of FDs when spawning a new thread,
- (b) all threads share fate (if one thread crashes, so do all the others).

The advantage of multi-threaded applications is that an FD leak is contained in a single process and it does not threaten to destabilize the entire system by depleting the kernel of all its resources.

In single-threaded applications, all connections are handled by the same process, so backlists are considerably longer. The furthest position of an active connection FD is linear in the number of connections. Recycling FDs in a single process application has a higher chance of closing an FD that is still in use.

7.4 Single-threaded Applications

The tested applications that handle all IO in a single thread are Squid, Hydra, Thttpd and WebFS. WebFS uses *select* while the other three applications use *poll* to handle concurrent IO. Single-threaded applications have similar characteristics to multi-threaded applications.

Single-threaded applications have a drawback. Imagine that no FD is available, and, that all FDs were used within the last second. Right now, our library fails and does not create a new FD. A better solution would be to wait for a while, recycle an FD and create the new FD. This is easy to do for multi-threaded applications, however, waiting in a single-threaded application blocks the entire application, so other FDs cannot be used in the meanwhile. Because of POSIX semantics, there are no asynchronous calls for creating FDs.

Chapter 8

Future Work

Our library has some limitations:

- (a) not all system calls are trapped
- (b) there is no communication between instances of the library running within different processes of the same applications
- (c) the library can cause FDs to be overloaded
- (d) to trap as many system calls as possible, applications need to be recompiled and linked dynamically
- (e) the library can be used only with C programs.

We propose a new implementation based on the *ptrace* system call that uses a layer of virtual FDs to prevent overloading. The new implementation should translate all system calls (such that the kernel sees only real FDs and the applications see only virtual FDs). We propose that the new implementation create one thread for each thread or process monitored. The advantages of this new approach are:

- (a) this method is guaranteed to trap all system calls (thus allowing the use of the virtual layer of FDs)
- (b) this method works with all applications (even non-C) and it does not require recompilation of source code

- (c) all decisions are made within the same external process, so gathering statistics and limiting the total number of FDs (across multiple processes) can be done easily.

We think that some challenges in implementing the new prototype are:

- (a) the system calls *fork*, *clone* and *exec* cause races with *ptrace*. If a layer of virtual FDs is used, no system call may be missed (all system calls must have their parameters translated); however, there is no way to have *ptrace* automatically attach to a child process after fork, so the child process might issue system calls before *ptrace* can attach.

- (b) the *select* libc function uses a bitmap for passing in sets of FDs. The size of the bitmap is fixed, so a larger space of virtual FDs cannot be created.

- (c) the program will be running in an external process, so it will be difficult to inject code within the targeted process.

Bibliography

[1] Apache http server website. http://httpd.apache.org/.

[2] Hydra website. http://hydra.hellug.gr/.

[3] Oops! proxy server website. http://zipper.paco.net/ igor/oops.eng/.

[4] Pure-ftpd website. http://www.pureftpd.org/project/pure-ftpd.

[5] Shared libraries howto. http://www.linux.org/docs/ldp/howto/Program-Library-HOWTO/shared-libraries.html.

[6] Squid website. http://www.squid-cache.org/.

[7] Thttpd website. http://www.acme.com/software/thttpd/.

[8] Tinyproxy website. http://tinyproxy.sourceforge.net/.

[9] Webfs website. http://linux.bytesex.org/misc/webfs.html.

[10] Wget website. http://www.gnu.org/software/wget/.

[11] Hans J. Boehm. Space efficient conservative garbage collection. *SIGPLAN Not.*, 39(4):490–501, 2004.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[13] Huu Hai Nguyen and Martin Rinard. Using cyclic memory allocation to eliminate memory leaks. Technical report, MIT, 10 2005.

[14] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[15] Stephen R. Walli. The posix family of standards. *StandardView*, 3(1):11–17, 1995.