🥕 Carrot: A Distributed Interposition Library for Networking and File Systems

Christopher Pondoc, Daniel Ma, Ihyun Nam, Kamyar Salahi

Final Paper for CS244B, Spring 2024

Abstract

We introduce Carrot, an interposition library that captures system calls and distributes them to different machines. We focus on two domains for interposition: file system-related as well as networking system calls. In the context of file systems, Carrot reroutes `open`, `close`, `read`, `write`, and several other system calls to multiple remote machines. As a result, a process can access and interact with files and directories as though they are on the local machine when they are distributed across several. In the networking context, we focus on the task of making HTTP `GET` requests to access website source code. Carrot thus distributes `sendto` system calls to remote machines such that processes can then retrieve any HTTP website even if such content is inaccessible, such as from behind a firewall. This is effectively a Virtual Private Network (VPN), showcasing how we can use Carrot to bypass censorship. Empirically, we show that we are able to successfully distribute file system and networking operations at the cost of added latency. As such, we suggest areas for improvement in our implementation and recommend Carrot's usage for applications where latency may not be mission-critical. Our source code is open-source and available on GitHub.

1 Introduction

Interposition is the technique of intercepting function calls, system calls, or other messages between software components to trace, change, or log the interactions. The process is typically achieved by inserting middleware between communicating parts of the software stack. Broadly, interposition can be used for debugging and monitoring; security, by enforcing certain policies and preventing malicious activities; performance analysis; and even testing and validation. Thus, we see several real-world applications employing this technique, from operating systems (e.g., `LD_PRELOAD` [7]) to debugging tools (e.g., GNU Debugger, otherwise known as GDB [8]) to performance profilers (e.g., DTrace [2]).

In this paper, we consider distributed interposition: intercepting system calls and sending them to other machines. With this motivation in mind, we created Carrot, an interposition library that captures and distributes system calls. We focus on two domains: file systems and networking. For each, we design architectures that effectively perform distribution while introducing no changes to the original process. In the real world, distributing file system calls can be used to provide more storage than what is available on a local machine, do extensive back-ups, and even compile a file "locally" while having the compiler run on a remote machine. Similarly, distributing networking system calls can help bypass censorship, allowing any process to retrieve any website content even if it is inaccessible due to a firewall. Our evaluations indicate that interposition does add latency overhead; thus, we provide areas for future work and suggest that the current Carrot prototype be used in contexts where latency may not be most top of mind.

2 Related Work

Carrot is primarily motivated by Parrot [9]. Parrot is an interposition agent that can intercept local I/O operations and forward them to remote systems. In this way, Parrot maps local application semantics onto the application semantics of a local machine. Similar to Carrot, Parrot proposes using a debugger trap such as `ptrace` to intercept the system calls of a process and replace them with calls to their remote equivalents. Unlike replacing standard libraries, debugger traps enable interception of I/O operations for software that cannot be rebuilt (e.g., source code unavailable). The popular portable debugger GDB [8] similarly utilizes `ptrace` to observe and control process execution flows and modify memory and registers but does not perform any remote operations. We extend Carrot to support remote network system calls in addition to remote file system operations. Unlike Carrot, rather than supporting Chirp [10], FTP, and other distributed I/O services, we implement our own server to enable creation, opening, reading, and writing of files and directories.

To implement our remote file system, we have implemented Remote Procedure Calls (RPC) [1] in our client and server wherein the client sends a payload containing system call ID, function arguments stored in registers, as well as any buffers that need to be transmitted to the remote machines for IO and awaits a packet containing the return value of the remotely called function as well as any output buffers. Our stubs are implemented using Google Protocol Buffers (protobufs) [4].

To make our system tolerant to single failures of remote machines, our file system calls leverage consistent hashing [5] to replicate files across multiple machines as to ensure incremental scalability of our approach. To achieve more uniform distribution, DynamoDB [3] suggested a variation on consistent hashing wherein the ring is split into many fixed-sized segments with virtual nodes. We leave this extension to future work.

We note that our networking syscall interception mimics the behavior of a VPN in enabling a machine to access network data as if it were on another network.

3 Design Overview

The Carrot library is written as a C++ executable. Using standard input, you can specify what process you want to trace, and Carrot spawns a child process of the application that it is then able to trace and interpose upon. In terms of the general system architecture, there are two types of machines: senders and receivers. The original process and the Carrot interposition library both live on sender machines, which make requests to receiver machines to perform the operations remotely. Communication between senders and receivers are serialized through protobufs.

3.1 Architecture of Distributed File Interposition

We aimed to create a load-balanced and fault-tolerant system to distribute file-based system calls. At a high-level, we have one sender, from which requests are transmitted, and many receiver machines where files are stored. Every file is stored on its corresponding receiver machine, and its closest neighbor machine in a consistent hashing scheme, resulting in uniform load-balancing of all files. [5]. We replicate the directory structure across all receiver machines and correspondingly update any state on the sender machine. This allows the sender machine to be completely independent of the receiver machines. If the sender machine goes down, another machine can be set up to access teh remote file system through the receiver machines. The file system will be fully accessible despite any singular failure of receiver machines due to directory and file replication under consistent hashing.

3.2 Architecture of Distributed Networking Interposition

We envision that Carrot's networking side functionalities could be used to access blocked websites under censorship. Consider a network under censorship in which `google.com` is blocked on some machines, including our sender, and not on some machines. Carrot interposes on `GET` requests, but forwarding the request to just one other machine has a high probability of failure, as that machine may also have `google.com` blocked. Therefore, upon interposing on the `sendto` system call, Carrot sequentially forwards the system call to multiple distinct machines. Machines that have `google.com` blocked will simply not send any response back while machines that have access to the website will send the normal fetched website.

We also architected a separate architecture, albeit untested, in which after forwarding the interposed system call to multiple machines, each of the machines forwards the system calls to other *intermediaries* that are closer to the destination. This is a more accurate simulation of a censored network in which the sender is multiple nodes away from a machine inside the firewall that can fetch the requested website. We also integrate a timeout functionality, which has the sender wait for a predetermined amount of time before eventually aborting to avoid indefinitely waiting for a response.

4 Implementation

At a high-level, we built Carrot by using `ptrace`, which allows one process to inspect, change, and effectively control another process. For both networking and file system-related system calls, we first extract all of the arguments within each corresponding system call by accessing the corresponding registers. Then, we would serialize all of this information into a protobuf, with different protobuf schemas for networking and file systems. Finally, we would send over this message to several remote machines to perform the operation requested by the original process. The results of these operations would then be sent back to the interposition library, which would update the return value and discontinue the original system call from running.

Initially, we leveraged `ptrace` memory access functionality to iteratively perform reads and writes on a byte-by-byte basis through `POKEDATA` requests. However, the overhead of performing these reads and writes led to an order of magnitude increase in I/O latencies. As a consequence, we instead directly read from and write to the memory of the child proccess using `/proc/[pid]/mem`. This virtual filesystem allows efficient accesses the memory address space of an arbitrary process.

4.1 Interposing on File System-Related Calls

We first developed a version of Carrot that interposes on several file system-related system calls. These include `open`, `close`, `read`, `write`, `mkdir`, `chdir`, and `getcwd`. During interposition, we used two separate protobuf schemas: one for sending over requests from the interposition library to the distributed remote machines (`CarrotFileRequest`) and another for sending responses back (`CarrotFileResponse`).

4.1.1 Distributing File System-Related System Calls

To set up consistent hashing, we compute the SHA-256 hash of the IP address of each receiver machine. To simulate a hash ring, we use an ordered map, where the value of each key is the previously computed SHA-256 hash. Each node, then, represents one receiver machine. To assign files to machines, we compute the SHA-256 hash of the absolute path of each file. Each file is then assigned to the immediate receiver machine with a hash greater than or equal to it, as well as the receiver machine immediately after it in the ordered map.

Upon processing an `open` system call, we receive file descriptors back from the receiver machines where the file was opened. This file descriptor is saved to a map, where it is mapped to the absolute path of the corresponding file. This map is used for `close`, `read`, and `write` system calls, to translate the given file descriptor parameter to a receiver machine's corresponding file descriptor to send the system call to.

Finally, changes to the current working directory only occur when we do a `chdir`. Thus, when we distribute a `chdir` to the receiver machine, we also make sure to return the current working directory and set that equal to the corresponding path recorded in the interposition library.

4.2 Interposing on Networking System Calls

The networking side of Carrot interposes on `sendto` and `recvfrom` system calls. Namely, we focus on the task of making HTTP requests to websites. To this end, we built our own custom `curl` executable, which makes a GET request to a website and returns its page source.

Similar to the file system context, we first use protobufs (`CarrotMessage`) to send over a request to remote machines to access the source code of websites. Using these three pieces of information, the remote machine then makes the `GET` request on behalf of the original process. Upon receiving a response from the website, the remote machine waits to collect all of the bytes before sending over all of the contents from the website's response back to the original process. Note that this does add additional overhead compared to simply streaming all of the bytes – in other words, continuously doing a `recv` until we don't receive more bytes. However, the latter approach ended up leading to synchronization issues during experimentation.

4.2.1 Distributing Networking-Related System Calls

In the setting where we are unsure which machines can access which websites, we send requests to each receiver machine one-by-one and waits for each response. Upon receiving, Carrot checks that the response contains a predefined HTTP success code and closes the socket upon receiving it, preventing itself from receiving any more responses.

5 Performance Evaluation

When testing Carrot, we mainly wanted to see whether or not we would see higher latencies due to either interposition or distributing the system calls across multiple machines. Specifically: we set up experiments for both file systems and networking applications and experimented by running them locally without Carrot, using Carrot with one remote machine, and using Carrot with more than one remote machine. Unless otherwise specified, we performed all of our experiments on Google Cloud Platform (GCP) `e2-small` instances running on Intel Broadwell.

5.1 Benchmarking Distributing File System Operations

To test out our file system interposition, we first used some smaller experiments that included only a small number of file operations, such as opening and closing or reading and writing a single file, as well as making a singular directory, changing directories, and then creating a file. In addition, we decided to do one main stress test that involved making numerous directories and changing directories to create files in each of them. Our timing results are below:

Test	**Local**	**1 Machine (sending to 1)**	**2 Machines (sending to 1)**	**3 Machines (sending to 2)**	**3 Machines (sending to 1)**
Test1 (create 1 file)	0ms	6ms	115ms	369ms	257ms
Test2 (create a directory and 1 file)	0ms	9ms	339ms	661ms	334ms
Test3 (create 4 files one at a time)	1ms	27ms	597ms	1537ms	879ms
Test4 (create a directory, 5 files one at a time)	3ms	24ms	679ms	2097ms	1652ms
Test5 (create a directory, 5 files at the same time)	10ms	28ms	753ms	2088ms	1006ms
Test6 (create 10 directories)	0ms	14ms	753ms	1474ms	1478ms

Table 1: Results from File System Benchmarks

Table 1 shows our results. The latency added from file operations sending to just 1 machine is quite minimal. However this latency increases significantly after just adding 2 machines, with a large linear increase to 3 machines. We think this might have to do with our somewhat inefficient implementation of the hash ring, requiring multiple lookups to decide what auxiliary machine to send to. One thing of note is that the directory system seems to be working as intended, with Test6, our intensive directory test, resulting in a similar latency with both 3 machines sending to 2 as well as 3 machines sending to 1.

5.2 Benchmarking Networking Interposition

We perform two main experiments for networking interposition. First, we try routing networking system calls to only one machine. Our second experiment involves distributing the network system calls to multiple machines. We simulate certain machines to fail in accessing a certain website, and we query all machines until we get a successful response. In sum, this allows us to access a wide variety of websites across different machines with varying firewall policies. We specifically try to access `google.com` and `example.com`.

Table 2 shows our results. We acknowledge that our interposition library, while successful in returning website source code when either a direct connection or several machines are unable to access the website, does add latency. Send latency appears to scale linearly, which makes sense, given that they are quite fast. On the other hand, we see relatively large jumps in receive latency. For starters, `GET` requests can

Latency	Local	1 Machine (sending to 1)	3 Machines (2 succeed)
Send	1ms	2ms	6ms
Receive	52ms	70ms	104ms
Total	56ms	86ms	132ms

Table 2: Results from Networking Benchmarks

be quite large. Given that our implementation of `curl` iteratively reads all chunks of the GET response on the receiver before sending back to the sender message, this understandably adds noticeable latency. Likewise, we send to each machine and wait for a response using `recv`. This operation is blocking which results in latencies compounding as we scale our system up.

6 Conclusion + Future Work

In this paper, we introduce Carrot, an interposition library that distributes system calls across multiple machines. Specifically, we interpose on specific file and networking system calls to showcase applications such as a distributed file system as well as a VPN-like service. Our benchmarks indicate that latency did not drastically decrease with the introduction of interposition and connection to remote machines, while introducing no change for the end programmer.

Naturally, there is much work that can be done in the future. On the general performance side, being more careful about when the tracer gets interrupted can help overall speed. Thus, a first step could be to use `seccomp`, alongside the Berkeley Packet Filter [6], to filter out non-relevant system calls during interposition. For networking, HTTPS is the most natural next step. This would involve more thorough interposition and careful tracing of the system calls involved in SSL, but would enable a lot more functionality. We also want to study latency of data transmission in a multi-hop fashion to overcome potentially more challenging firewalls.

On the file system side, one of the main design improvements involves how we deal with directories. Upon the creation or change to a new directory, we send a request to each remote machine to take care of it. As we can imagine, though, this leads to a lot of overhead, as well as could increase latency with an increasing number of machines. In an alternative approach, the interposition library could manage all of the directory structure locally (or write it to a separate centralized database). The interposition library could manage mappings of directories to files or subdirectories. Remote machines could then store data with filenames as the hash of their absolute path, resulting in arbitrarily shallow remote directory structures.

We also consider an improvement in fault tolerance. If a remote machine fails, we could simply use the interposition library's mappings to shift files that were written to the failed machine to other machines. In the case of the interposition library failing, we could serialize and store the contents of this metadata structure at certain intervals and load in a checkpoint upon reboot.

References

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, feb 1984.

[2] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[3] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[4] Google. Protocol buffers, 2008. Accessed: 2024-06-07.

[5] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the

world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[6] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[7] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

[8] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[9] Douglas Thain and Miron Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[10] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *J. Grid Comput.*, 7:51–72, 03 2009.