Copy-on-Reference File Mechanism Extends Scope of Dynamic Reconfiguration

Jeremy Krach
Department of Computer Science
University of Maryland, College Park
jakrach@umd.edu

***Abstract*—Maintaining resource availability for processes as they dynamically move between hosts is difficult without building relationships for the underlying namespaces of those resources. For example, migrating opened files as part of a process migration system commonly relies upon shared file-systems that provide a common namespace between hosts. This paper describes a new technique for process-migration that does not require a common namespace for file resources, instead migrating the opened files on reference. The copy-on-reference, or lazy, approach to file resources adds to a rich corpus of past work on process migration by dynamically migrating files from different namespaces instead of referencing static, shared namespaces. In particular, this technique is motivated by on-going efforts using dynamic reconfiguration to implement active defense security mechanisms. By describing the spectrum of past work and dynamic reconfiguration scenarios suited for a new model, this study demonstrates the value of copy-on-reference mechanisms for handling file resources during process migration.**

I. Introduction

Process migration has been an active area for study, with many tools and capabilities having been introduced. Designers rely upon migration of running processes to increase the reliability of their distributed systems, improve performance, and enhance their security. In our work, *process migration* refers to relocation of a running process without substantial interruption to either its progress or its access to system resources (specifically opened files). Process migration is one of the three dimensions of change invoked in our effort to improve security through active defenses. Hofmeister and Purtilo describe *dynamic reconfiguration* as change with respect to any of an applications *topology* or structure the logical connectivity of components; its *geometry* the locality at which components operate; or its components *implementation* [1]. In the present paper, process migration falls within geometry-based dynamic reconfiguration changes.

When performing migration, the operating system may manipulate the process as an object that encapsulates code and data; the location or composition of this object may be changed accordingly. When a process is changed, we need a way to maintain the availability of system resources. This may be done by associating the resources with the old process in a new location or by remapping the access to a new (potentially dynamically reconfigured) process. One of the greater challenges posed by the resource information kept by the operating system is maintaining access to open file descriptors.

Despite the vast space of existing projects that offer process migration, only a subset support file migration beyond shared file-systems. These projects are effective, and file operations generally work as expected. Most services accomplish this by offering a common namespace, where the files location is primarily immutable: it is up to system to map each file descriptor to the final location. One such example, *networked file systems* (NFS), offer a single namespace so that any process may reference a file regardless of the processs location within a network.

Our work in active defenses (which leverages dynamic reconfiguration to improve a systems security properties) was substantially constrained by the need to restrict changes to only environments which could handle open file descriptors natively. We sought a more general tool to map such system resources in order to enjoy the full security value of our technique.

We offer a system that promises consistent read-write semantics from the point of view of any individual process. Our approach does not require a common name space in order to effect change. We enable the agent requesting change to direct not only how processes should be reconfigured, but also how their open file descriptors should be remapped. We accomplish this by accepting a more relaxed view of run time properties with respect to interplay between processes; two processes which may have shared an open file on one host might, after a reconfiguration, have different views (or not) depending on the changes requested by the agent. This freedom supports our security research involving active defenses.

Specifically, this paper offers LazyFS, an alternative within the space of process migration intended to work within user-space and use the copy-on-reference paradigm for file migration. This helps populate the diverse spectrum of work on process migration by providing another option for the end-user. Under particular circumstances, each technique has strengths and weaknesses: LazyFS is no different. However, LazyFS enriches the run-time environment of live process migration by adding a new tool with a different design methodology. Its unique feature, lazy file retrieval, allows for migrations to happen over time, rather than all at once. This feature applies specifically within dynamic reconfiguration motivated scenarios, as outlined in the design portion of this paper.

The copy-on-reference file migration scheme drew inspiration from Accent, which uses copy-on-reference to transfer process memory; this achieved a 58% reduction in data transferred [2]. By lazily retrieving opened files on reference, LazyFS makes savings over methods that copy the entire file-system at the time of migration. Although something like NFS

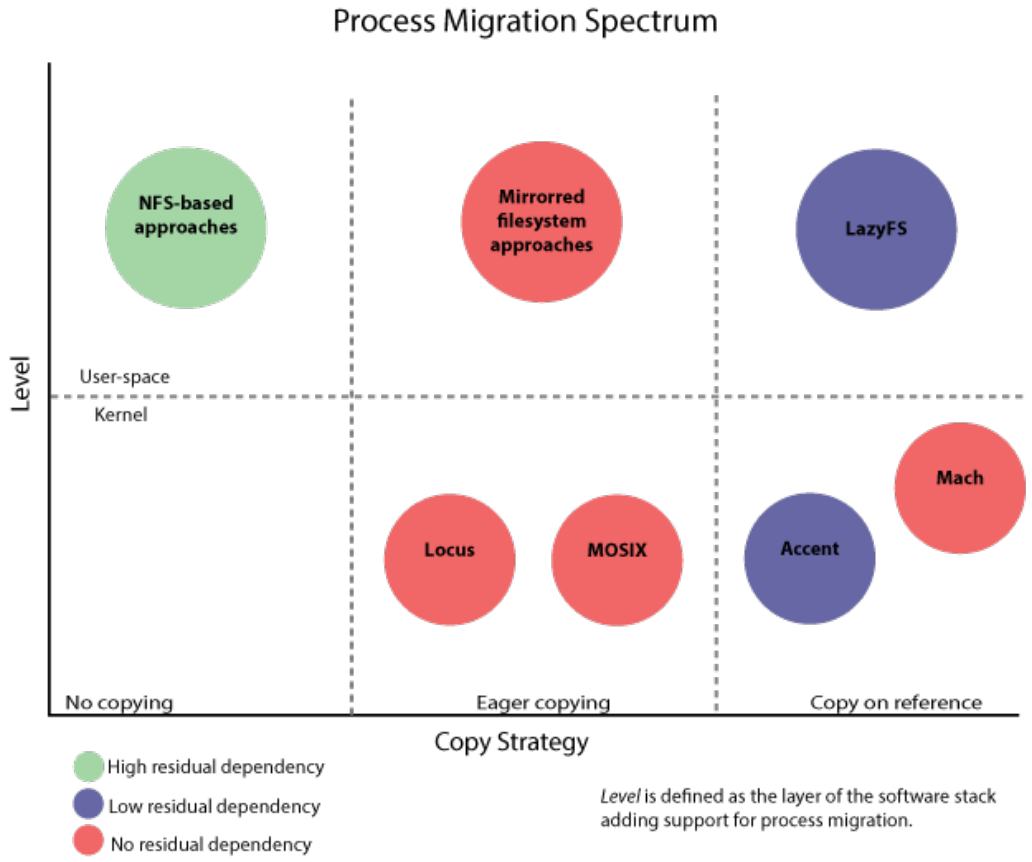

Fig. 1. Process migration spectrum summary

could also prevent mass-file transfer, LazyFS has the advantage of making no assumptions about the underlying namespaces of the participating hosts. Additionally, LazyFS brings the migrated files to the new host so that they are stored locally. Once local, these files do not rely on network availability.

This paper describes the existing spectrum of process migration capabilities and demonstrates the gap into which LazyFS fits. Section 2 outlines the spectrum of current technology in detail and compares and contrasts process migration design methodologies. Section 3 focuses on scenarios which influenced the design of LazyFS and implementation details. Section 4 describes the results from initial proof-of-concept experimentation. Section 5 is a discussion of these results and the implications for the scenarios. The final two sections outline future work (Section 6) and reflect on the outcomes of the project (Section 7).

II. Background

This research is motivated by the goal of creating a dynamic reconfiguration system to support active defense of cyber systems. Such reconfiguration involves moving processes between hosts within a network and, as a result, file references. Without assuming common file-system namespaces between hosts, migration techniques must define mechanisms to map these file resources from one machine to another. The purpose of this section is to describe the variety of alternatives existing in the process migration space and thus motivate our work.

Process migration has a long history of varied approaches. Mark Nuttall and Dejan Milojicic et al. provide excellent summaries of past work in this area [3, 4]. Both papers divide the space into the same general categories: operating system migrations, microkernel migrations, user-space migrations, and application-specific or object migration. Generally speaking, these categories run a spectrum from highly custom and optimized tools to more generic and portable ones. Each category, and some examples, are described in the following subsections.

The spectrum is summarized graphically in figure 1.

A. *Operating System Migrations*

This category describes custom systems designed to include process migration support. The benefits of such techniques are primarily performance and completeness: because these systems are created with process migration as a feature, they each handle process migration quite well. Typically, the problem with these systems is that, because they are custom operating systems, they are not portable. Migration tends to only be supported between systems running the same custom system. The overhead of deploying a custom operating system in order to support process migration outweighs the benefits provided by these systems for more casual end-users. Despite

the inconvenience of deploying such systems in a general use-case, they are frequently deployed into distributed compute environments where the requirements are different.

Sprite, MOSIX, Accent, and Locus are examples of custom operating systems built with process migration tasks in mind [3]. Each is a distributed operating system, built (generally) to allow load balancing and fault-tolerance in distributed compute environments. Sprite treats a network of computers as one computer. Its approach to file-migration uses a shared networked file system for all participating machines [3, 5]. MOSIX segments open file information into upper kernel global pointers (location) and lower kernel host-local information (data) [3, 6]. Accents contribution to this work is the copy-on-reference memory model, which influenced LazyFS file migration technique [2]. Locus is UNIX-compatible, however, the functionality built in requires a large amount of kernel changes [3, 7]. Although these systems handle process migration elegantly, the custom kernels introduce difficulties in support and compatibility.

B. Microkernel Migrations

Microkernel migrations are a modified version of the operating system migrations described in the previous section. Microkernel design reduces functionality in the kernel itself to a small subset, allowing a specific operating system to handle a large amount of the functionality outside of the kernel. Integrating process migration at the microkernel level allows more portability, as the operating system placed on top can simply make calls into the microkernels migration techniques [3]. However, its custom kernel design lends itself to the same drawbacks described for the operating system techniques from the previous section. Some examples of microkernel designs supporting process migration are the V kernel, RHODOS, Arcarde, Chorus, and Mach [3].

C. User-space Migrations

Process migration in user-space provides the most flexibility of all the migration techniques described. By building applications that are portable (and run in user-space), process migration has the potential to run on top of popular operating systems with little configuration or modification. The most notable example is running process migration in Linux. Although Milojicic et al. describe migration as not being a requirement for many end-users, that is rapidly changing in todays world [3]. With security risks growing and the popularity of cloud-based applications, process migration can realistically apply to a wide variety of scenarios not imagined in the past (i.e. beyond the distributed compute and reliability use-cases), especially, in our case, active defense activities. The primary benefit of user-space migrations is their ability to easily be deployed on top of existing systems and applied to any application. The downside of such migrations are the sacrifices in low-level capabilities.

Several popular user-space migration tools have been developed over the years. An early system, built on top of UNIX, was designed by Freedman [4, 8]. Despite running on UNIX with no modifications, the technique does not support any OS state information, including file descriptors [4, 8]. Another tool is Condor, which uses checkpoint and restore techniques to migrate processes between machines. Despite working well for long-running and compute-intensive processes, the overhead makes Condor not worthwhile for more short-term processes or migrations [3]. To handle open file descriptors, Condor directed system calls back to the original host through a modified version of the C library [3, 9]. Two more UNIX-based systems were developed by Alonso & Kyrimis and Mandelberg & Sunderam. The former added system calls for taking a checkpoint and restoring processes, but required that the process to be migrated did not communicate or depend on other processes [3, 10]. The latter only supported NFS files [3, 11]. A recent tool, P.Haul (process hauler), uses the CRIU library to checkpoint a process and also relies on NFS to restore the process on the new host with the appropriate file-system namespace [12]. The downside of this is the requirement of the additional server (NFS) infrastructure and that the network be maintained for the entire lifespan of the migrated process.

Within the user-space migration category, there remains a gap for a tool that supports regular open files, while at the same time reducing the up-front cost of shipping large amounts of process data. LazyFS, with its copy-on-reference file strategy, fits into this capability gap and enhances the current space of process migration tools.

D. Object Migrations

Object migrations, albeit a less important model in the context of this project, form an important category in the broader space of process migration. Focusing more on application-specific migration, these tools generally meet a particular use case. These frequently take the form of programming languages or annotations built to take into account distributed environments where the programmer may want to move a particular object between machines. One such example is the Emerald programming language [4]. Within the spectrum, these fit on the high end above user-space migrations: they are incredibly portable but have the disadvantage of needing to have the application created using a particular method (like a specific language).

III. LazyFS Design

LazyFS, by design, fills a gap in the process migration spectrum described in the previous section. Although user-space migration techniques exist in UNIX, many of them opt to rely on NFS for a shared file-system namespace. This introduces a *long-term residual dependency* on machines external to the migrated host. Some techniques do not support opened files at all. LazyFS is designed to perform user-level process migration and support opened files through a copy-on-reference paradigm. This design enables a *short-term residual dependency* on machines external to the migrated host. The design and implementation of LazyFS was a several stage process. Each step of this process is outlined in a section below. First, scenarios were envisioned to highlight desired features. Second, modifications to an existing checkpoint and restore library (CRIU) were outlined to hook into LazyFS on process restore. Third, a file-system, LazyFS, was designed and implemented using FUSE (file-system in user-space) to enable copy-on-reference file access. Finally, the designs came

together to form a cohesive package for running process migration with open files.

A. Scenarios

The primary goal in designing scenarios was to discover how file migration through copy-on-reference could enhance the existing space of process migration: particularly the active defense and security goals of dynamic reconfiguration efforts. These scenarios informed the design: not all are perfect fits for the current implementation of LazyFS. For completeness, all of the scenarios envisioned are outlined below. The results section contains more information about which scenarios work practically with LazyFS as implemented.

1) Limited bandwidth: The first scenario is limited bandwidth migration. Limited bandwidth could be enforced as a requirement for a variety of reasons: perhaps the network is suffering a distributed denial of service attack, bandwidth is expensive, or excessive bandwidth would prevent higher priority tasks from completing. Since network bandwidth is at a premium in this scenario, another migration scheme might not be feasible: shipping the entire file-system as a bulk transfer could clog up the network and maintained network access over the processs lifetime may not be desirable. Lazy file migration using copy-on-reference allows files to only be retrieved as needed and within a limited time window following migration, reducing stress on the network at the time of a migration and for the lifetime of the migrated process.

2) Honeypots: Perhaps an administrator would like to perform dynamic process migration to move a malicious agent onto a honeypot. When the process is moved to the honeypot, it expects all the same files it was previously using to also exist on that system. By crafting a mechanism that intercepts file system calls, any semantics could be introduced to retrieve content for those opened files. In this case, migration from a production host to a honeypot, sanitization or dummy-data generation could become the policy enforced by said mechanism. This would ensure the confidentiality and integrity of the production data while simultaneously misinforming malicious agents.

3) Maintenance: A system may need to have reduced access for maintenance. Dynamic file migration allows the process to rapidly migrate to a different machine for clients to interact with (assuring minimal downtime), while only occasionally calling back to the original host to retrieve necessary files. This allows the original host to have maintenance performed without compromising availability for the particular process in question. Like the limited bandwidth scenario, this also covers denial of service attacks. If one host is being hit with a large amount of illegitimate traffic, it could be migrated to a new host that is positioned handle the legitimate traffic. The original host could continue to eat the traffic from the attack and occasionally respond to requests for a single file from the migrated host.

4) Rapid short term migration: By focusing on only moving the files that absolutely need to be moved the initial cost of performing migrations is much less compared to one that clones the file-system. This allows for more rapid and frequent migration: perhaps to confound outsiders peeking into an internal network. Imagine a pool of machines each with the capability to run a certain sensitive process. Perhaps employees or insiders have knowledge about the rotation of the sensitive process around this pool, but to an outsider it would be more difficult to predict given rapid migration.

5) IPC: In some models, all the files of interest are moved at a single point in time (instead of using NFS, they rely on duplicate local file-systems). Perhaps, however, only one process is being migrated and the original machine is still being used for other processes. Imagine also those other processes are producing information for the process that was migrated to consume. The previous model would essentially only gain a snapshot of the old information: dynamic file migration could allow for additional information to continually come in as files were fetched dynamically over the network when needed. This model is more similar to one that uses NFS.

6) Embedded devices: The Internet of Things and embedded devices also provide interesting scenarios for this capability. Having a variety of small devices and moving processes rapidly between them could add even more capabilities. By removing the cruft of heaving around the entire file-system with each migration, a copy-on-reference model lends itself to the smaller memory of IoT and embedded devices.

In addition to each of the above scenarios, LazyFS also encapsulates any scenario enabled by the general concept of process migration.

B. CRIU Modifications

Checkpoint/Restore in User-space (CRIU) is an open-source tool for taking a checkpoint and restoring a process on a Linux system [12]. In the checkpoint workflow, CRIU saves information about the current process into a variety of image files. These files are stored as either Google protocol buffers, CRIU custom binary data, or 3rd party formats. On restore, CRIU reads in these image files and recreates the process at the same point it was checkpointed. As part of this process, CRIU saves open file data into several image files: reg-files.img and *fdinfo.img. The reg-files file stores the paths to any file opened by a call to `open()`. The fdinfo files store information about all the open file descriptors for that process. Performing a union over these files gives us the file descriptor and path for any opened regular files used by the checkpointed process. This information is all that is required to retrieve files for a checkpointed process; as such no modifications were made to the checkpoint logic.

On the other hand, the restore logic needed some simple modifications. The changes are kept at this fork of the main CRIU repository: https://github.com/jakrach/criu. Since the changes are quite minimal, they have been included in their entirety in figure 2.

This code is introduced into CRIUs logic for restoring regular, open files. First, it checks whether the file to restore exists on the current host. If the file is not present, then the files path is altered to point to a location managed by LazyFS rather than throw an error. This is done by prepending /lazyfs/ to the file path and replacing all previous directory delimiters (/) with a notation LazyFS will recognize (.) to reconstruct the path when performing file retrieval.

```
+    // File not local, point to CRIU
+    if (faccessat(ns_root_fd, rfi->path, F_OK, 0) == -1) {
+        pr_info("%s does not exist, hooking up with Lazyfs\n", rfi->path);
+        int i;
+        char new_path[PATH_MAX];
+        for(i = 0; i < strlen(rfi->path); i++) {
+            if (rfi->path[i] == '/') {
+                rfi->path[i] = '.';
+            }
+        }
+        sprintf(new_path, "/lazyfs/%s", rfi->path);
+        strcpy(rfi->path, new_path);
+        pr_info("Lazyfs path is %s\n", rfi->path);
+    }
```

Fig. 2. Code change in modified CRIU repository

C. LazyFS Architecture

The LazyFS utility is implemented in Golang. LazyFS relies on go-fuse, the implementation of the file-system in user-space interface in Golang. LazyFS also uses the protocol buffer library for parsing some of CRIUs image files. LazyFS is open-source and available at https://github.com/jakrach/lazyfs. The program, when run, mounts a directory at the specified mount point with files representing all the regular open files of the process whose checkpointed image files are specified at the command line. These files, initially, are placeholder files that have no contents (however, their size is accurate to the expected file size from the checkpoint). Once one of these files is read from or written to, LazyFS uses SCP to retrieve the remote file from the original host. Once the file has been copied, it is kept locally. Any further operations on that file will simply loop back to the local copy.

Since the repository is relatively small, the details of each file will be covered here.

1) protobuf/: This directory contains a condensed version of the protocol buffer specification from CRIU. This gives the structure for both the reg-file.img and *fdinfo.img files. It can be compiled into Golang, which allows the rest of the code to parse out the appropriate information from CRIU image files.

2) files.go: This file defines a structure called `LazyFile` and common file operations on that structure. A `LazyFile` contains a link to the protocol buffer structure, the original file path, and a link to the local copy of the file, if it has been retrieved. The majority of the functions on `LazyFiles` are standard and execute if there exists a local copy of the file. The two interesting functions are `Read` and `Write`. Both check if a local copy exists and, if not, retrieve the file from the original host by performing a SCP call. The details of that call are found in `fetchRemote`.

3) lazyfs.go: This file contains the bulk of the management logic for the file-system itself. It defines the `LazyFS` structure, which contains information about the remote host and a map of protocol buffers from CRIU representing open regular files. `LazyFS` defines `GetAttr` to pull information about the represented files from the protocol buffers, `OpenDir` to retrieve the listing of represented files, and `Open` to retrieve a `LazyFile` representation of the file requested. This file also houses the main function, which verifies the user input, parses the CRIU image files, and mounts the FUSE interface. The main function also handles SIGINT by gracefully shutting down.

4) reader.go: This file contains the logic for parsing the image files from CRIU. The `ImageReader` interface specifies the required functions for parsing a given file format. Since LazyFS is only concerned with data from the reg-files.img and *fdinfo.img files, only two parsers are defined: `RegFileImg` and `FdinfoImg`. The CRIU image files are stored as an array of protocol buffers. Both parsers traverse the array and extract each element one-by-one. The specifics of these file formats were not well documented and as a result had to be reverse engineered. However, the structure was easy to predict and, once understood, consistent between files.

D. Complete Migration

The entire process for performing a migration is diagrammed in figure 3. First, a process is started on the original host. While executing, the process is checkpointed by a user running the `criu dump` command. The CRIU version on the original host does not need to be the modified version since the changes only impact restores. The dump command creates CRIU image files representing the checkpointed process. From there, the user must copy the image files to the new host. The LazyFS process is then launched on the new host, using the image files from CRIU. This instantiates placeholder files in the LazyFS mounted directory. The process can now be restored on the new host by using the modified version of the `criu restore` command. As soon as the restored process accesses (through read or write) one of its opened files, LazyFS will retrieve the file from the original host with SCP and use the local copy for all further read and write queries.

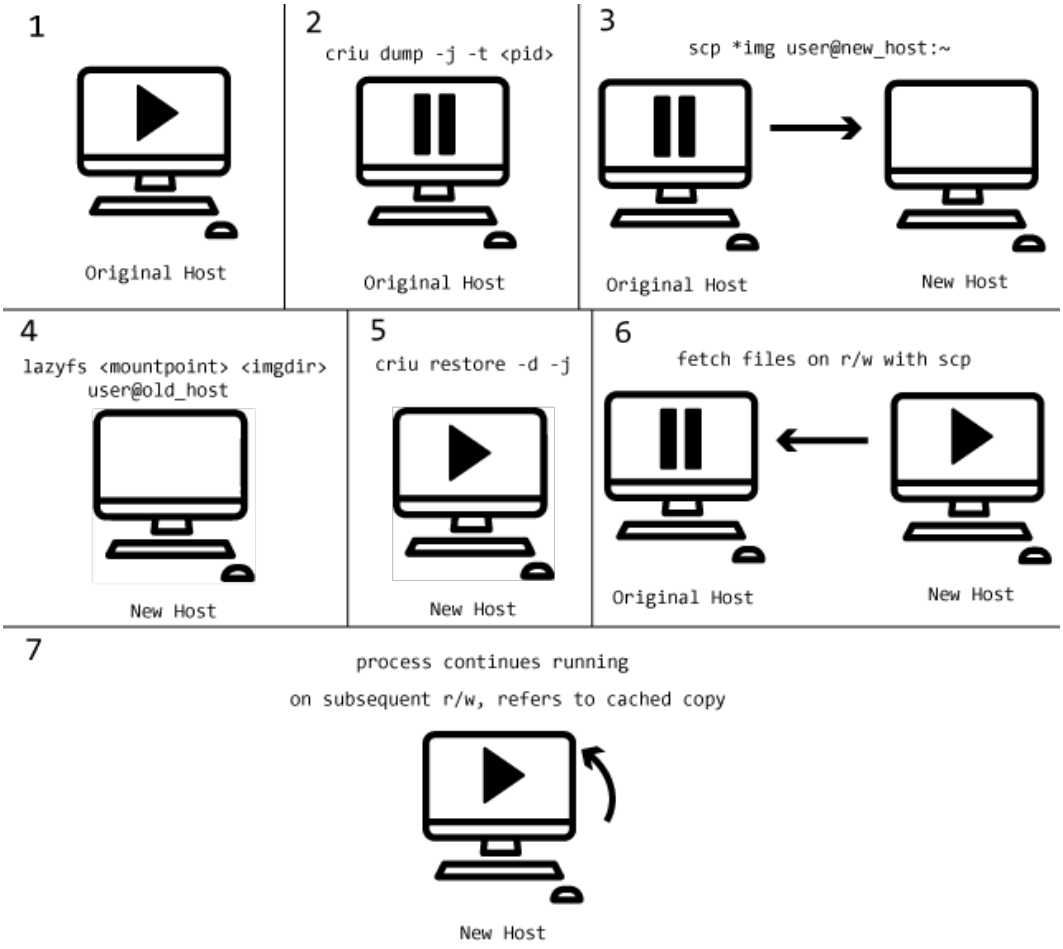

Fig. 3. Migration process using LazyFS (icons courtesy of flaticon.com)

IV. Results

Two video demonstrations can be found on the aforementioned GitHub page. Each involved a simple program running with a single file opened for either reading or writing (each experiment focused on one mode). The network traffic for both programs was measured when using LazyFS to perform migration and using NFS for the same task. The graphs of the traffic are below.

V. Discussion

The primary goal of this research is to provide a new capability in the process migration space that aligns with the research objectives of ongoing dynamic reconfiguration efforts within the SEAM (Software Engineering at Maryland) group [13]. Here, the new capability takes an alternative approach to the oft-neglected problem of opened files within the user-space side of the process migration spectrum. The current space has two approaches for opened files (1) ignore opened files and trust the process will not use any or (2) rely on a duplicate file-system or NFS. The first approach is not a realistic assumption, as many processes that would benefit from migration frequently use the file-system. The second approach does handle files, but makes key assumptions that LazyFS does not. NFS assumes the network will remain available throughout the entire lifetime of the migrated process. LazyFS only assumes network availability for the initial start-up and some period after to allow complete lazy migration. LazyFS adopts a short-term residual dependency, contrary to NFS-based techniques long-term residual dependency.

This assumption can be understood by looking at the graphs describing the network traffic in each trial. The NFS tests relied on the network, sending packets regularly throughout the processs duration. The LazyFS tests sent their data earlier on in bulk, slowly tapering off until the network was no longer used. This demonstrates the space that LazyFS fits within the process migration spectrum: LazyFS provides support for file migration without a long-term residual dependency like NFS. Although LazyFS still relies on the network to some extent, its dependency lessens as the process continues to run, eventually becoming nonexistent. The more file operations and the longer running a process, the more appealing LazyFS will appear relative to NFS. That said, certain use cases may find one approach more useful than the other. For example, by copying the file from the original host to the new host, other processes on the original host will not receive or be able to transmit any modifications to the original file. This is good, as LazyFS is not out to replace NFS-based migrations, only provide a new alternative that provides different capabilities. However, it is worth noting that the mechanism of intercepting read and write calls on process restore allows the introduction of arbitrary

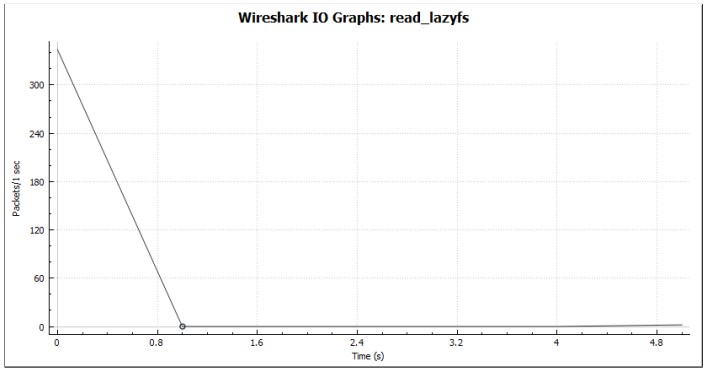

Fig. 4. Network traffic for read example process migration using LazyFS.

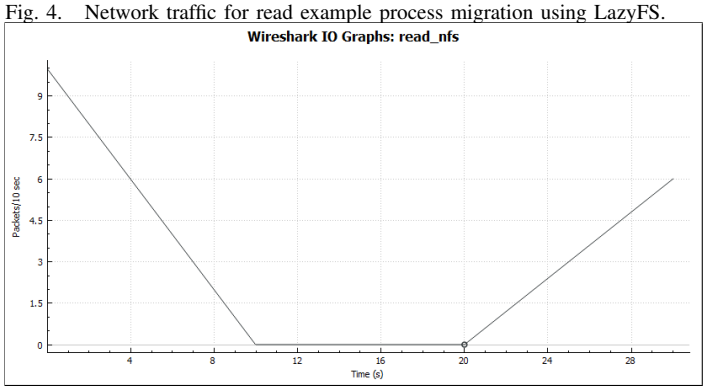

Fig. 5. Network traffic for read example process migration using NFS.

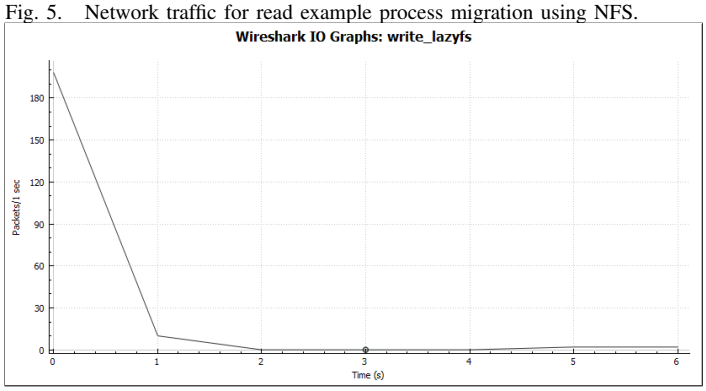

Fig. 6. Network traffic for write example process migration using LazyFS.

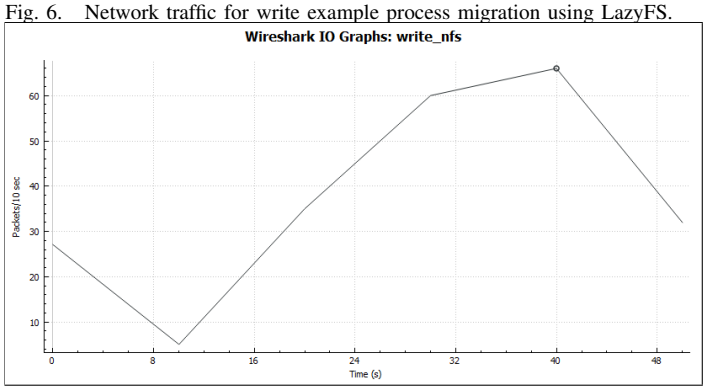

Fig. 7. Network traffic for write example process migration using NFS.

policies to enable different semantics for file migration.

LazyFS directly supports several scenarios envision in the design section. The most obvious fits are the first and third scenarios: limited bandwidth and maintenance. The previous discussion of LazyFS vs. NFS particularly highlights the ability of LazyFS to draw out its migration over the long term and rely less on the network as time goes on. Short-term migration and embedded device migrations may be better suited to an NFS-based approach, as repeated migrations would repeat the cost of migrating files when using LazyFS as it is currently implemented. Modifying LazyFS to have repeated migrations refer back to the original host could resolve that particular drawback, but could introduce additional issues. LazyFS also does not support the honeypot scenario: however, a modification of the `LazyFile` could replace the SCP logic with a scrubbing or scrambling function that cleans or manipulates the data before returning it to the migrated process (thus preventing the service on the honeypot from accessing the original, sensitive data). Finally, the IPC scenario is more suited to an NFS-based migration scheme.

All said, LazyFS enables scenarios in new ways and extends the current feature space of user-space based process migration techniques. By using copy-on-reference, LazyFS relies on the network less than the traditional NFS approach. This allows user-space migrations to use regular files that are not hooked up to NFS, a limitation of earlier process migration tools that worked in user-space (such as P.Haul). Additionally, LazyFS enables migrations between hosts that do not share a common resource namespace. LazyFSs approach of intercepting file-system calls through FUSE also enables the designer of active defense systems to build arbitrary file migration policies not explored here. As such, LazyFS has found a new niche within the process migration spectrum.

VI. Future Work

The SEAM group has several projects in the works involving dynamic reconfiguration [13]. These projects generally focus on the active defense implications of introducing change at various levels of the security stack. One project, for example, deals with network topology reconfiguration through a software bus API. Another is looking into the models needed to predict when network-level reconfiguration is necessary to counter security threats. LazyFS deals specifically within the space of geometric reconfiguration by opening new capabilities for process migration.

Within the LazyFS project, there is still work to be done. More tooling in the form of scripts or fully-integrated software packages (combining CRIU modifications and file-system code) would make performing migrations easier on the designer looking to use LazyFS as a full-on migration tool. Implementing and envisioning additional scenarios would help extend the prevalence of LazyFS and provide incentives for adoption. Adding robust testing would ensure quality for future development.

VII. Conclusion

LazyFS introduces a new method for process migration that fits a niche unsatisfied by previous work. The spectrum of process migration capabilities runs from custom-built operating systems to user-space programs layered on-top of existing systems. By introducing copy-on-reference as a scheme for migrating opened files, LazyFS handles file migration differently

than many existing user-space process migration tools, which rely on a shared file-system namespace, typically through NFS. Such a design enables a wide variety of scenarios that include security, reliability, and convenience implications. LazyFS demonstrates the diverse applications that copy-on-reference design can bring to process migration, as already seen with process memory in the Accent system [2]. By utilizing FUSE, LazyFS also introduces the ability to enforce arbitrary file-migration policies within user-space process migration.

All code used in this project is open-source and available on GitHub at https://github.com/jakrach/.

ACKNOWLEDGMENT

I would like to thank Dr. James Purtilo, director of the Software Engineering at Maryland (SEAM) group, who served as my advisor throughout this research.

REFERENCES

[1] J. Purtilo and C. Hofmeister. "Dynamic Reconfiguration of Distributed Programs," *11th International Conference on Distributed Computing Systems*, pp. 560–571, May 1991.

[2] E. Zayas. "Attacking the process migration bottleneck," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 13–24, Nov. 1987.

[3] D. S. Milojii, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241-299, Sep. 2000.

[4] M. Nuttall. "A brief survey of systems providing process or object migration facilities," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 4, pp. 64–80, Oct. 1994.

[5] F. Douglis and J. Ousterhout. "Transparent process migration: Design alternatives and the sprite implementation," *Software Practice and Experience*, vol. 21, no. 8, pp. 757–785, Aug. 1991.

[6] A. Barak and A. Litman. "MOS: A multicomputer distributed operating system." *Software Practice and Experience*, vol. 15, no. 8, pp. 725–737, Aug. 1985.

[7] B. Walker, G. Popek, et al. "The LOCUS distributed operating system," *Proc. 9th ACM Symp. on Operating System Principles*, pp. 49–70, 1983.

[8] D. Freedman. "Experience building a process migration subsystem for UNIX," *USENIX Winter Conference*, pp. 349–354, Jan. 1991.

[9] M. Litzkow and M. Solomon. "Supporting checkpointing and process migration outside the UNIX kernel," *USENIX Winter Conference*, pp. 283–290, Jan. 1992.

[10] Alonso, R. and Kyrimis, K. "A Process Migration Implementation for a UNIX System," *Proceedings of the USENIX Winter Conference*, pp. 365372, Feb. 1988.

[11] Mandelberg, K. and Sunderam, V. "Process Migration in UNIX Networks," *Proceedings of USENIX Winter Conference*, pp. 357363, Feb. 1988.

[12] CRIU. [Online]. Available: https://criu.org/Main_Page. Accessed: Feb. 12, 2017.

[13] J. Purtilo. *Software Engineering at Maryland*. [Online]. Available: https://seam.cs.umd.edu/. Accessed: Feb. 12, 2017.