"Follow-Me" Migration: Seamless TCP Connection Migration

Jiayi Liu, Thomas Oliver, and Benjamin Reeves

University of Michigan

{liujiayi,olivertc,bgreeves}@umich.edu

Abstract

Distributed cloud applications demand robust solutions to the problems of liveness, maintenance, provisioning, load balancing, and fault-tolerance. To help deal with these problems, one of the most versatile tools we have at our disposal is migration: the ability to migrate a running service from one host to another. One of the primary goals in migration is to make the entire process transparent to the end-user; thus, much research has been put into how to make the process of migration faster and less disruptive.

An important step in maintaining this transparency is ensuring that we preserve any network connections that may have existed between the service and its end-users at the time of migration. With the popularization of containers and the recent advent of open-source checkpoint/restore software, this is achievable in the right circumstances. However, doing so requires a network solution to redirect packets destined for the application's old host to the new host. But what if these circumstances are not available to us, like in an edge cloud?

In this report, we document our efforts to achieve transparent process migration between physical hosts without dropping any of the process' active TCP connections and without modifying application code. We take inspiration from prior work and implement a proof-of-concept patch to the Linux kernel that allows a migrated process to request its remote clients to reconnect to the new IP address. We dub this method "Follow-Me" Migration and we demonstrate its functionality by migrating a simple echo server from one physical host to another without disrupting its client-server TCP connection.

1 Introduction

The past decade or so has seen a dramatic increase in the deployment of massively distributed cloud applications and services. This trend shows no signs of stopping, with cloud providers such as AWS and Azure becoming the de-facto method of deploying new software services and distributed applications. This paradigm demands robust solutions to many problems, including machine provisioning, liveness, maintenance, and fault-tolerance. An important tool that aids in solving these problems is migration: the ability to migrate a service or application from one host to another.

For example, consider the use case of compute load balancing in a data center. Over time, a single host may become overloaded with CPU-intensive applications, but if there is another host machine that is under-utilized, some applications could be migrated from the congested machine to the idle machine. This makes better use of the available resources.

Another potential use case for migration is for maintenance. It can be difficult to upgrade a server's hardware or software without shutting down the computer, which would cause a service outage for the applications running on it. Migration can simply move the applications to another host before the maintenance period to decrease the application's downtime.

A final example of the usefulness of migration is to reduce latency. It is common knowledge that the latency of an application is an important aspect of good user experience. Given how much people are on the go, a connection that had low latency when the connection started might not be the best choice for the current location of the user. With migration, the service could be moved closer to where the end user is, reducing latency and providing a better user experience.

We believe application migration can be an effective solution to many common problems in the cloud. However, although mature software exists for performing migration, a key factor is often missing: that of preserving active TCP connections during migration. This leaving a large gap in the possible applications that can be seamlessly migrated without service interruptions.

In this report, we document our efforts to achieve transparent process migration between hosts without dropping the process' active TCP connections. We (re-)propose a modification to the TCP standard that allows one endpoint to migrate to a new IP address while maintaining the connection. We describe our proof-of-concept Linux kernel implementation of this modification and discuss our results testing our implementation with an echo server and other applications.

2 Background

Application migration is the process of taking a currently running application (whether it be a single process, a collection of processes, or even a collection of machines) and moving it to a new physical machine or set of machines, such that it is no longer running in its original location, but is now running at the new location. Application migration is a challenging problem to solve, and can be approached in a number of different ways. Here we describe what is involved in the process of application migration, the different approaches to migration, and some common tools used to achieve migration.

2.1 Approaches to Migration

Migrating an arbitrary application is a complex process, because a running application has many potential dependencies:

- The actual code / executable(s) of the application
- Its memory and stacks
- Any child or parent processes in its process tree
- Active network or UNIX sockets
- Open files

Migrating an application can be approached in different ways, and each method resolves these dependencies differently. The major choice is between the different "units of migration." These are typically either: (1) a single process (really, its entire process tree), (2) a virtual machine, or (3) a container.

2.1.1 Process Migration

The most obvious unit of migration is a single process (or, potentially, a tree of processes). In this approach, we would like to migrate a single running process tree from one machine to another machine of similar architecture and specs. This approach was popular in the early days of migration nearly two decades ago; however, each attempt had its own significant drawbacks and limitations [8].

The most common issue that faces process migration is dealing with “residual dependencies” that the process had on its original host. For example, you may need to retrieve files from the old host, or you may need to forward packets received at the old host to the new one. These issues could be solved by using a networked file system and a network-level router respectively.

Today, process migration is accomplished with the use of checkpoint/restore software. Using C/R, the running process tree is frozen and has its current state dumped to disk (this is the checkpoint stage), and a new process is created on the new host, and has its current state overwritten to be identical to the checkpointed process. One growingly popular C/R software today is CRIU (Checkpoint/Restore in Userspace), which we will discuss in §2.2.

2.1.2 Virtual Machine Migration

When OS and hardware virtualization started becoming feasible and popular, developers realized that they could utilize virtual machines (VMs) to solve many of the inherent dependency problems in process migration. They simply had to run their applications inside one or more virtual machines, and then migrate the VMs themselves, which would transfer essentially every dependency (file system, hardware, kernel networking stack state, process memory) as an atomic unit. This technique has been used in many migration works (e.g., [3] and [7]).

Today, in our opinion, VM migration is a bit of an overkill approach. With the popularization of containers, we do not find it necessary to have to migrate and entire operating system and the machine itself. Nevertheless, VM migration has been proven practical and is widely available from certain providers like Microsoft Azure [1] and VMware [2].

2.1.3 Container Migration

Recently, the use of containers has exploded in popularity, and thus the container has become another option for the unit of migration. Containers offer a number of the same benefits as virtual machines when it comes to migration, as many of the resources can be bundled inside the container itself, such as the file system and network namespace. Additionally, unlike VM migration, it does not require a totally different migration approach; modern C/R software can migrate both processes and containers (after all, a container is simply a set of one or more processes in its own namespace and cgroup). Works investigating container migration include [9] and [10].

2.2 Checkpoint/Restore

Process or container migration is done with the help of checkpoint/restore (C/R) software. Once a process has been frozen and checkpointed to disk, the generated image files may be sent over the network to a new host machine, where the application may then be restored. This process is depicted in Figure 1.

CRIU [5] is a modern tool under active development and increasing adoption that enables process or container C/R from userspace. Running CRIU requires no custom kernel as of Linux 3.5, and its functionality is even being integrated into popular container frameworks such as Docker.

CRIU facilitates nearly every possible need in the process of migration. It provides the functionality for freezing and restoring processes, extracting their memory and dumping it to disk, and extracting kernel-level information such as the process’ open files and sockets. The extraction and restoration of TCP connection state achieved with the help of the TCP Repair features in the Linux kernel [4]. Currently, CRIU allows TCP connections to be restored only if they are restored on the same machine as they were checkpointed from.

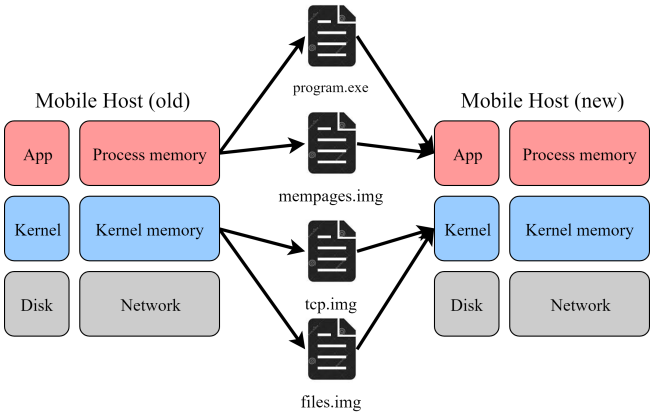

Figure 1: A depiction of the checkpoint/restore procedure for migrating a process from one host to another.

2.3 The Missing Client Piece

We could say that the problem of application migration is a puzzle that has been gradually solved over the past two decades. Today, most of the puzzle pieces are already in place, thanks to tools like CRIU. However, despite the success that developers have found in performing migration at all three of the levels mentioned above, there remains a fundamental issue: **If we have an application that we want to migrate to a new physical machine (with a different IP address), and it currently has open network connections to its external clients, how do we handle this?** The default and undesirable behavior is for these connections to break, because the application will terminate on the original host, causing the kernel to RST any further incoming TCP packets to the original port.

There are a number of different ways in which the issue of connection migration can be solved:

- **Re-route packets.** After performing a migration, one could set up a proxy on the original host machine to forward packets destined for the migrated connection to the new host. Alternatively, if working in a data center environment, one could reconfigure the front-end load balancers to redirect packets to the new internal host machine. This is most likely the solution that is implemented by data centers today in order to support migration. However, for our general-purpose use cases, we would like to be free of this dependency on the network.

- **Have the client initiate the migration.** This is the approach suggested by works such as Mobile TCP [13]. In this case, the client is the one who instructs the server to perform a migration. This could theoretically be useful if the client is a mobile device and has knowledge about which server endpoint would provide superior performance. However, in general, this is entirely the opposite of what we want to achieve. Recall the use case of distributing compute load in a data center. In this case, it is the server who knows that it wants to migrate, and it must find a way to notify the client.

- **Checkpoint the client too.** This is the approach taken by MIGSOCK [6]. Essentially, they checkpoint the client application along with the server application. While the client application is frozen, they can modify the client application's socket to point to the new endpoint. This unfortunately results in a disruption of service on the client. Furthermore, it will not work if the client application runs with a GUI or any sort of user interaction.

- **Modify the TCP stack to support migration.**

  One particular implementation of this is proposed by Snoeren and Balakrishnan in [11]. Essentially, the client and server's TCP stacks are both modified to support connection migration and a migrated application is able to send a special packet to its clients to instruct them to migrate their TCP sockets to point to the new IP address. The authors of this paper implemented their TCP migration scheme as a patch for Linux 2.2 [12]. In the end, our approach is highly inspired by this work, and we will discuss it in more detail in §3.

This missing piece to the puzzle is precisely what we set out to solve in this project. We will now discuss our high level approach to implementing TCP connection migration support in the TCP stack.

3 Design Approach

When it came time to decide how to approach achieving seamless migration without dropping TCP connections, we had to decide between two options:

1. Do everything in userspace, which would require checkpointing the client application in order to modify its sockets (similar to MIGSOCK [6]).

2. Modify the TCP stack in the kernel, similar to the end-to-end approach in [11].

In the end, we decided that the kernel approach would be better. Unlike the userspace approach, it would work for applications that have a GUI and would not cause any disruption to the client program (save for any incidental TCP delays). Furthermore, we decided that it would not really end up being any easier to do the userspace approach; it would have required us to write client- and server-side daemons to communicate migration information between the server and client, plus lots of scripts.

We will now describe the details of our approach. Not all of these ideas have been implemented in our proof-of-concept

kernel patch (see our implementation in §4.2), but ideally, all of the following functionality would be present.

3.1 "Follow-Me" Migration

The use cases for migration that we mentioned in the introduction all beg for a solution which requires no interaction from the fixed host[1]. Thus, the solution we propose is dubbed "Follow-Me" Migration, an extension to the TCP stack which allows the endpoints of a TCP connection to migrate between IP addresses[2].

The high-level method for "Follow-Me" Migration is depicted by example in Figure 2 and described below.

3.1.1 Migration Token

During the initial connection establishment, the client and server inform each other about whether or not they support TCP connection migration. If they both do, they negotiate a secure token that uniquely represents this connection. This all happens during the three-way handshake.

In order for this to be a secure protocol, it is required that the token not be leaked to any outside listener and only ever be known by the two end hosts (and any authorized hosts to which one endpoint migrates). How this is accomplished is not the focus of this project.

3.1.2 Migration Request

At some point, either one of the endpoints will be checkpointed and migrated to a new IP address. When the mobile application is restored on a new host machine, it uses each of its connected sockets to send a special Migration Request packet to each previously connected fixed host. This packet uses the token that was negotiated during the handshake, which ensures to the fixed host that this migration request is authentic.

Thus, if a fixed host receives a Migration Request packet with a recognized migration token, it locates the socket that belongs to that connection and modifies its destination address to point to the mobile host's new IP address.

3.1.3 TCP Options

To implement these two stages, we propose two new TCP options that are to be included in the TCP headers of three-way handshake packets and migration request packets: the `MIGRATE_PERM` and `MIGRATE_REQ` options[3].

First is the `MIGRATE_PERM` option, which is included inside SYN and SYNACK packets to notify a remote endpoint that this machine supports TCP connection migration. The contents of this option is the migration token (or some information which allows both endpoints to infer the value of the connection token; see §3.1.4).

Second is the `MIGRATE_REQ` option. Including this option in a packet marks it as a Migration Request. The contents of this option is the migration token.

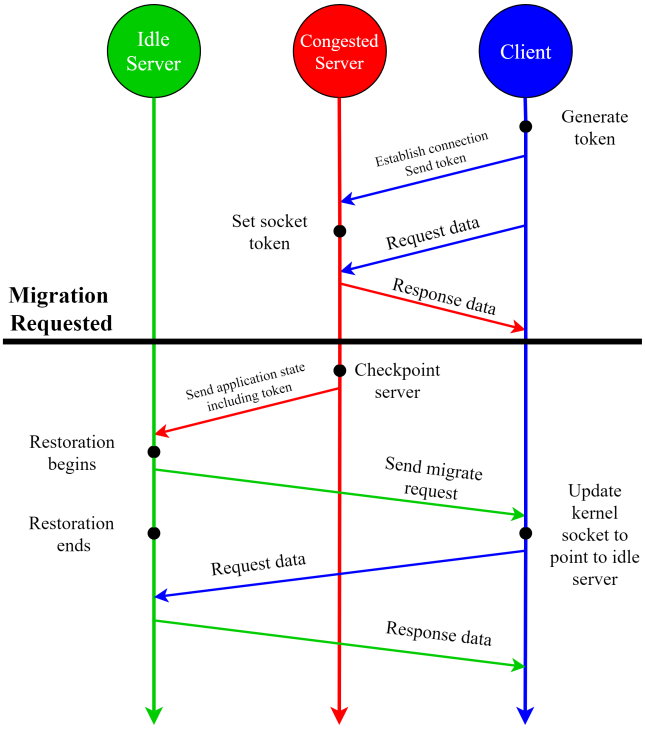

Figure 2: An example of using "Follow-Me" Migration to migrate a server application from a congested host machine to an idle host machine. The server process is first checkpointed on the congested machine, and iptables rules are erected to prevent the kernel from RST-ing incoming packets from the client. When the process is restored on the idle server, it sends a `MIGRATE_REQ` packet to the client using the token that was originally negotiated between the client and congested machine. The client accepts this migration request and redirects its TCP connection to the new IP address, and communication resumes.

[1] The "fixed host" or "fixed endpoint" is the endpoint which does not move during a migration. For example, consider migrating a web server from one IP to the other. In this case, the client is the fixed host, and the server is the mobile host. The mobile host can be further broken down and referred to by its "old host" and "new host," which signify which machines the application is running on before and after a migration respectively.

[2] Note that using this approach actually allows either endpoint of the connection to migrate. Pretty neat.

[3] In our proof-of-concept implementation, these options use type 40 and 41 respectively.

3.1.4 Security

As stated, in order for "Follow-Me" Migration to be secure, the migration token must only be known by the two endpoints of the TCP connection. We are less clear on the best method to tackle this. The method proposed by Snoeren and Balakrishnan in [11] seems like it is on the right track, but it is confusing and messy in a number of ways. Notably, it requires manually patching in cryptography code into the kernel.

Our proof-of-concept kernel patch (§4.2) does not implement any security measures. However, if we *were* to implement some form of security, the general idea would be quite similar to [11]. During the 3-way handshake, the two endpoints would negotiate a shared key via a Diffie-Hellman exchange. Once this key k is securely agreed upon by the two hosts, they would both be able to infer the value of the connection token. That key could then be used to sign a migrate request to guarantee the identity of the sender of the request.

4 Summary of Work

Now that we have discussed our design goals, we will now detail the work that we actually completed for this project. After we spent some time acquainting ourselves with CRIU and investigating related academic literature, we began developing a proof-of-concept kernel patch to implement the new TCP migration features. To pair with this custom kernel, we supplied some minor changes to the CRIU source code that allowed it to take advantage of the new kernel features and provide seamless checkpoint/restore functionality between physical machines.

All of the source code that we have written can be found at our GitHub project page[4].

4.1 Initial Testing

We first wanted to understand how CRIU works and test if we could migrate TCP connections using it alone (i.e., no kernel changes). Our approach to accomplish TCP migration with only CRIU begins by checkpointing both the client and the server. Then, when both are checkpointed, we would move the server to its new host. We then modify the dumped files so that when they are restored, the sockets would be connect correctly. This involves changing the destination address of the client and the source address of the server. Finally, we restore both applications.

There are some additional difficulties to making this work, however. The first is that if only one of the client or the server is restored while the other is not, it is possible that the restored application will send a request to the new host, in which case the packet will be reset. To remedy this, we add two `iptables` rules to drop packets from the client to the new server and from the new server to the client. Once both applications were restored, we would drop both of these rules.

The problem with this approach is that this only works when the client application can be checkpointed. CRIU has restrictions on what applications it can checkpoint and an important one is that it cannot checkpoint applications with a GUI without some important changes to how the client is run. Further, because the application is checkpointed, the end user would see unpleasant UI lag. Finally, there is a decent amount of work managing the timing of all this with add and removing `iptables` rules and creating a socket connection between the new host and the client to communicate this timing. After some experimenting, we found that this approach worked for a basic echo server, giving us confidence that the project might actually work.

4.2 Proof-of-Concept Kernel Implementation

Having assured ourselves that CRIU works as expected, we had reached a critical decision point. As discussed in §3, we faced the option of pursuing either an out-of-kernel-option (which would require interrupting the client application and creating persistent daemons) or an in-kernel option, and we decided on pursuing the in-kernel option.

Thus, we have developed a limited yet functional proof-of-concept implementation of TCP migration support in a patch for the 4.15 Linux Kernel.

This kernel patch is a "*make-it-working*" subset of the features discussed in §3. We implemented the bare minimum functionality so that we would be able to demonstrate migration in time for the poster session. This has a number of implications on the current state of the work, namely a distinct lack of robustness and generality.

However, an important thing to note when evaluating the robustness, correctness, and elegance of this patch is that this was the first time any of us had ever touched kernel-level code. Though we all have an academic background is OS concepts, the challenge of reckoning with the complexity and spaghetti provided the majority of the challenge in this project.

4.2.1 Kernel Patch

We have developed a proof-of-concept kernel patch which adds support for TCP connection migration to the TCP/IPv4 stack in the Linux 4.15 kernel.

In order for TCP connection migration to succeed, both the fixed host and the mobile host must run this custom kernel. The fixed host's kernel listens for incoming `MIGRATE_REQ` packets and modifies the corresponding internal socket structure to point to the new endpoint if the request contains a recognized migration token. The mobile host's kernel is equipped with a means of sending a `MIGRATE_REQ` packet from a socket that is being restored from a previous state. Both endpoints' kernels allow userspace to peer into the socket's

[4] https://github.com/tcp-migrate-umich

token value using TCP repair mode [4]. This allows the connection to be checkpointed and migrated.

4.2.2 Migration Token

To get our patch up and running, we implemented a very naive solution for negotiating migration tokens. In fact, there really is no negotiation at all; the client socket simply selects a random token for itself and expects the server to accept it as a unique identifier for the connection.

Currently, the token is a simple 32-bit unsigned integer limited to an arbitrary range (0 to 9,999). It is stored inside the `struct tcp_sock` data structure as `tp->migrate_token`, along with a `tp->migrate_enabled` flag. This allows each socket to either support or not support TCP migration individually.

The kernel is configured to enable TCP migration support by default for all newly created sockets, although at this point, the migration token remains unassigned (`TCP_MIGRATE_NOTOKEN`). A socket's migration token is set in one of two places:

1. When the user calls `connect()`, the kernel assigns that socket a random token that is unused on this machine. This token is used for the `MIGRATE_PERM` option in the SYN packet header.

2. When a listening socket receives the final ACK of a three-way handshake, it assigns to the newly created connection socket the same token which arrived in the original SYN packet.

This approach to the migration token has obvious drawbacks and is certainly not robust. Since tokens are assigned randomly to client sockets, the assigned token may conflict with the server machine if that token is already in use by a client socket on that machine. The way the kernel currently handles this is to reject the final ACK of the three-way handshake if the token conflicts.

4.2.3 Custom Socket Options

CRIU's ability to checkpoint and restore active TCP connections is achieved with the help of the `TCP_REPAIR` kernel patches (which have been in mainline Linux since Linux 3.5) [4]. These patches allow the user to put a socket into "repair mode" via a special socket option. Once a socket is in repair mode, certain internal attributes of the TCP socket may be extracted for the purposes of checkpointing (for example, the send and receive sequence numbers, the contents of the send and receive buffers, and the congestion window size).

Similarly, in order for CRIU to be able to successfully migrate a process using our new migratable sockets, we must give userspace some way to access a socket's migration token. Therefore, we expose two new socket options to the user: `TCP_MIGRATE_ENABLED` and `TCP_MIGRATE_TOKEN`, the latter of which is only available while the socket is in repair mode. CRIU can then use these sockopts to extract the migration token and restore it on the new host (see §4.3).

4.2.4 Connection Establishment

When a SYN packet is prepared in the client's kernel, the kernel checks the value of the socket's `migrate_enabled` member. If it is enabled, it adds a `MIGRATE_PERM` option to the TCP header of the SYN packet, which will notify the server that this machine supports TCP migration.

In the server's kernel, if it receives a SYN packet with `MIGRATE_PERM`, it checks the value of `migrate_enabled` on the listening socket. If migration is enabled, then it responds in kind with a `MIGRATE_PERM` option in its SYNACK packet, echoing the same token back to the client. In the Linux TCP stack, the kernel does not create a full-blown socket for pending three-way handshakes. Instead, it creates an intermediate "request socket". This is where our custom kernel stores the incoming token while we wait for the handshake to complete.

Upon the server's reception of the final ACK of the handshake, a full-blown `tcp_connection_sock` is created, and our custom kernel takes the opportunity to copy the token from the request sock to the connection sock. Now both endpoints agree on the token which identifies this connection, and either endpoint may be successfully checkpointed and restored on a different machine.

4.2.5 Sending Migration Requests

Rather than exposing an API to userspace that allows the user to send a migration request (`MIGRATE_REQ`) at any time, we do it automatically inside the kernel. If the kernel ever receives a valid `setsockopt()` call for the `TCP_MIGRATE_TOKEN` option while in repair mode, then we assume that this must be a socket from an old, migrated connection. Thus, inside `tcp_do_setsockopt`, we automatically invoke the code to send a migrate request if the user sets the value of the migration token. Again, this was a "*make-it-working*" approach, so we may revisit this decision in the future.

The code to send the migration request calls `tcp_send_ack`, but makes sure to add a `MIGRATE_REQ` option, complete with the user-assigned migrate token, to the TCP header before the packet gets sent out. We made use of the `send_ack` function because we did not feel the need to figure out the complexity of sending a real SYN from a socket that's already in the established state. It worked so we stuck with it.

4.2.6 Handling Migration Requests

This was arguably the most technically challenging part of implementing this kernel patch. The core dilemma that has to be solved is that the Linux kernel's TCP stack rejects any incoming packet whose TCP/IP 4-tuple (srcaddr, srcport, dstaddr,

dstport) does not match any of its existing sockets. This is of course the case for an incoming `MIGRATE_REQ` packet, as it comes from an unrecognized host.

Therefore, our custom kernel is unable to leverage the existing hashtables that exist in the TCP stack. Typically, whenever any TCP packet is received, the kernel uses the TCP/IP 4-tuple to do a hashtable lookup to find either a listening socket or an established socket to which that packet should be routed. If the kernel on the fixed host receives a packet from the newly migrated mobile host, it will not locate a socket in the lookup step and will skip to sending a RST packet to the migrated host, completely thwarting the attempt to migrate the connection.

This ideally would not be that much of an issue; we could simply add code to the "reject" path that tries to process the migration request on the correct socket. The only problem is, which socket is that? The code has just failed to find a socket in the hashtable lookup, so we are at a loss for which socket this migration request could be destined for.

In Snoeren and Balakrishnan's original TCP migration kernel patch [12], this problem was solved by keeping a separate hashtable for migrate-enabled sockets. Migrate-enabled sockets were hashed and unhashed from this table at the same time as they were hashed and unhashed from the established sockets hashtable. This allowed the code to query for the socket to whom a given migration token belonged if the default socket lookup failed.

In our kernel patch, we took a similar but much more hacky approach. We created a global, fixed size array of size 10,000. The index of this array corresponds to a token, and the value at that index is a pointer to a socket struct. If the pointer is `NULL`, then that token is unused. To assign a new token to a client socket, we generate a random integer between 0 and 9,999, and we check the value of the pointer at that index. If it is `NULL`, then we select that token and update the pointer value, otherwise we linearly probe the array until we fall upon an open slot. If we probe once around the entire array, then we return an error as all tokens are currently used up.

Our implementation of hashing and unhashing from this "hashtable" was hastily written in the last few days before our project deadline, and is thus not robust at all (and often totally nonfunctional, as covered in §5.2). In fact, for the final presentation, we ended up rolling back to a previous version of the patch which only supported a single migrate-enabled socket per machine. We suspect that with a little more time, we could iron out the kinks and make it totally functional.

Finally, once we have hashed into the migrate "hashtable" and retrieved a pointer to the correct socket, the last step required for handling a migration request is twofold: first, update the `daddr` of the socket to point to the new endpoint, and second, re-hash the socket within the established sockets hashtable so that future packets received on this migrated connection will be routed to the correct socket instead of rejected.

4.3 CRIU Modifications

In order to work with our new kernel modifications, we had to make a small modification to the CRIU source code so that it would incorporate the socket's migration token into the checkpoint and restore process. To accomplish this task, we needed to add `migrate_enabled` and `migrate_token` fields to the image file dumped by CRIU. These fields are filled when other important fields related to TCP sockets are filled, like the congestion window size, using the `getsockopt` option provided by the kernel modification.

Then, the dumped image files would be changed using CRIU's companion tool CRIT to a readable format to modify the source address of existing sockets. We changed the image files to make it so the source address of any existing TCP socket matched the new host so the socket would be recreated correctly on the new host without changing any kernel code. Finally, when restoring the socket in CRIU, immediately after the socket is connected to the client in TCP repair mode, we restore the migrate enabled and migrate token using `setsockopt`. The `setsockopt` for the migrate token automatically sent out the migrate request, so CRIU did not need to manually send the request.

5 Evaluation

To test our work, we first used a simple echo server and then we progressed to testing more non-trivial applications.

5.1 Echo Server

Our most basic test application was a simple client-server echo program, where the client repeatedly sends a message to a remote server, and the server responds by echoing back the same message. Using our custom kernel, we were able to successfully checkpoint the server application and restore it on a different machine, and without interacting with the client application at all, the TCP connection continues exchanging data between the client and the newly migrated host application.

5.2 Failed Tests

While we were to make our echo server example work, we changed a lot of the kernel patch to support multiple migratable sockets at one time. After this change we found that the echo server still worked, but we encountered problems when we tested our system on more complex software.

Our first test was of video streaming over the browser. We tested our new system with a NodeJS server streaming a video to a Firefox client. When we migrated the server, it did not send the migrate request to the client. When we investigated the problem, we found the socket had added and then quickly removed the token from our token table. Therefore, when the

checkpoint occurred, the socket did not have an associated token and would not issue a migrate request. A similar problem happened in our testing of Minecraft. Due to time constraints, we decided to revert back to a more stable version of our code to ensure that our poster presentation went smoothly.

6 Future Work

Due to a lack of expertise in modifying the kernel and time constraints, there are many aspects of the project that can be improved upon to produce a more usable product.

6.1 Secure Token

Currently, our implementation randomly generates a token to enable migration and this token is exchanged in the original SYN packet from the client to the server. This has obvious security holes where an attacker could read that token and then send a migrate request to have the client redirect its traffic to the attacker. A possible solution to this problem was suggested by Snoeren and Balakrishnan [11] (for more details on this approach, see §3.1.4). Another potential solution would be to allow this security hole and have a higher layer of the network stack handle the security, such as TLS with HTTP traffic.

6.2 Performance Improvements

There is a lot of room left to optimize our proof-of-concept. A major optimization to decrease the downtime of the system would be to use CRIU's page server feature. This feature enables us to directly send the image files from one host to another without ever writing to disk, which can be a potential bottleneck in a large application that uses a lot of memory. Another important optimization to perform would be iterative migration where the application has its data dumped while it is still running. Then, the application is stopped and only the changes since the first dump need to be recorded. This can further decrease the downtime of the system. This technique is currently used by vMotion [2] to decrease downtime and we believe it would be valuable to add to the project.

6.3 CRIU Failure Conditions

It is possible that the user may be migrating and the restoration fails for some reason. Our implementation as it stands would treat that application as dead and would require restarting the application. A possible improvement to our system would be to detect the failure and bring the application back up on the old host to ensure that only the migration failed and not the whole application.

6.4 Automatic Load Balancing

While our solution works for individual processes, there is work to be done to integrate it into a container orchestration framework. First off, we would need to ensure that our modification to CRIU work with Docker's existing container checkpointing functionality. Further work would be to ensure that entire pods of containers are migrated together because each pod is guaranteed to be colocated on the same host machine. Finally, we would need to implement a system to detect when a host is congested and perform an evaluation as to which containers should be moved based on CPU load, memory used, time to migrate, and other important factors.

7 Conclusion

We implemented "Follow-Me" Migration to enable migration of TCP connections. It has been to shown to work for simple applications and could be useful for load balancing, maintaining and upgrading servers, and reducing latency for cloud-based applications.

References

[1] Improving Azure virtual machine resiliency with predictive ml and live migration. https://azure.microsoft.com/en-us/blog/improving-azure-virtual-machine-resiliency-with-predictive-ml-and-live-migration/.

[2] Live migration of virtual machines. https://www.vmware.com/products/vsphere/vmotion.html.

[3] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[4] Jonathan Corbet. Tcp connection repair, May 2012. https://lwn.net/Articles/495304/.

[5] Pavel Emelyanov. CRIU: Checkpoint/Restore in Userspace, 2011. https://criu.org/.

[6] Bryan Kuntz. MIGSOCK: Migratable TCP socket in Linux. Master's thesis, Carnegie Mellon University. Information Networking Institute, 2002.

[7] Peng Lu, Antonio Barbalace, Roberto Palmieri, and Binoy Ravindran. Adaptive live migration to improve load balancing in virtual machine environment. In *European Conference on Parallel Processing*, pages 116–125. Springer, 2013.

[8] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[9] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[10] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142. IEEE, 2017.

[11] Alex C Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM, 2000.

[12] Alex C Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility source code, 2000. http://nms.lcs.mit.edu/software/migrate/.

[13] Florin Sultan, Kiran Srinivasan, Deepa Iyer, and Liviu Iftode. Migratory TCP: Connection migration for service continuity in the internet. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 469–470. IEEE, 2002.