CSE 291 - Virtualization & Cloud Computing | Final Project Report

Allen Zhang, Jacob Root

I. Problem

A. Definition

Containers don't have any fully functional & reliable way to be ported across servers. Existing ways may cause downtime, data loss, or data inconsistency.

B. Motivation

To move one running container from one server to another in a traditional migration, you need to first stop the container, then move everything this container has to another server, then run container from there. This process causes downtime, which is detrimental.

Although a Kubernetes-like approach can be used, where a new pod is created and then traffic gets redirected from the old pod (and the old pod may get subsequently destroyed), it only works the best when the application is a simple web server (for example, if an application is doing writes to databases, launching that as a new container might cause duplicate database insertions or even cause data corruptions due to conflicting writes).

Another complication is that most containers typically recreate their root filesystem (do not preserve changes), across different containers, so there is a possibility for data loss when done incorrectly.

We aim to minimize disruptions to network flows and packet loss, particularly for applications sensitive to such disruptions

In the real world, companies could use this to move services across datacenters, for example if that datacenter is displaying instability (as Cloudflare does [22]).

II. Design Overview

A. The High-Level Node Interaction

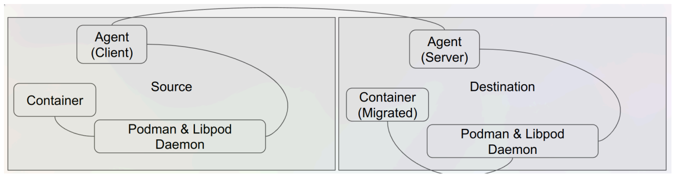

Figure 1: Overall Design

We designed our solution to make the source and destination nodes able to connect, transfer files, and give limited orders for the peer to execute.

Since we are migrating from one node to another, there are inherently two types of roles: source and destination. The source node is responsible for checkpointing and compressing the container, transferring files to destination node, and setting up WireGuard tunnels. The destination node is responsible for restoring the container, receiving files from the source node and placing them at the appropriate location, and setting up WireGuard tunnels.

We initially wanted a setup where the source node simply controlled the destination node using ssh and scp, however we realized that would be inflexible, as the destination node needs its own state. As a result, we switched to our current design.

B. Network Topology

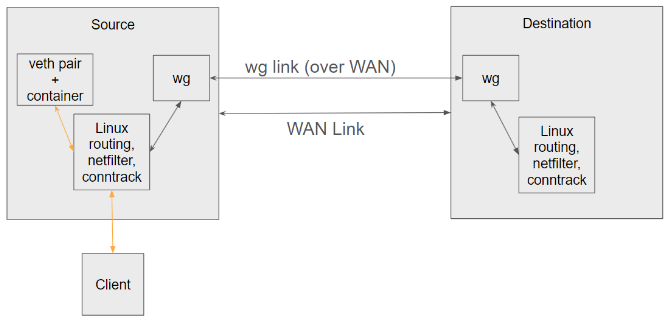

Figure 2: Network Topology (Pre-Migration; orange is packet flow)

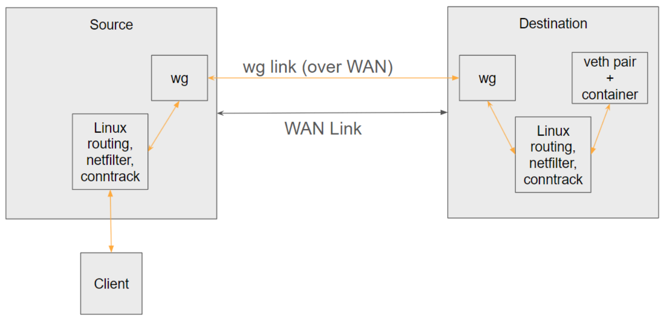

Figure 3: Network Topology (Post-Migration; orange is packet flow)

III. Solution

We implemented our two types of agents (client and server) with Python 3 [18]. The server agent is a long-running one that is supposed to take in any migration requests initiated by the server so we implemented using the Flask [19] HTTP server.

We decided to support Podman [23], using the netavark CNI implementation [24], with runc as our container runtime [25]. Podman uses CRIU [30] to do the checkpointing.

In order to programmatically interact with Podman, we utilized Podman's REST API [1] and local socket to directly execute Podman functions with Python.

To perform the live network connection migration, we used the iptables and conntrack shell commands to interact with the Linux [21] kernel's netfilter subsystem [20]. We also used iproute2 [26] to interact with the Linux network routing.

IV. Implementation

We approach our solution with the flow below:

1. On source node's agent, establish a connection and establish a handshake with the destination node's agent to make sure they are accessible with each other.
2. The two agents create a WireGuard tunnel and declare the source and the destination as peers.
3. Dump the container's pre-checkpoint with agent running on the source node (`podman container checkpoint [containerId] --pre-checkpoint --tcp-established -e pre-checkpoint.tar.gz`)
4. Transfer the container's pre-checkpoint from source node to destination node via HTTP.
5. Destination agent signals source agent about the successful transfer.
6. Dump the container running on source node using CRIU (`podman container checkpoint [containerId] --create-image [image] --tcp-established --leave-running -e checkpoint.tar.gz`)
7. Transfer the container's checkpoint from source node to destination node via HTTP.
8. Destination agent signals source agent about the successful transfer. Destination agent starts restoring the container from checkpoint. (`podman container restore --import checkpoint.tar.gz --import-previous pre-checkpoint.tar.gz`)
7. Stop and destroy the container in source node's agent.
8. Destination agent completes the container checkpoint restoration and successfully launches the container.
9. At this point, the migration is complete.

A. WireGuard

The establishment of the WireGuard tunnel is surprisingly complicated, so a flow is described here:

1. If not initialized, the source and destination node generate their public and private keys, then create a file in /tmp to describe all active WireGuard connections. A firewall rule is also created (see section B).
2. If the source node identifies that either node recognizes the connection as either not completely setup, both nodes teardown the state for that specific connection (not global state: the priv and pubkey do not change, /tmp removes all state related to the connection)
3. Both nodes initiate the connection process, and add an entry to that file in /tmp describing the connection as incomplete and with routing rules not currently setup.
4. Both nodes mutually agree on a /30 to give the WireGuard interface, and the ports they will be communicating over. They also communicate their public keys.
5. Both nodes create the interface and assign this information, and then mark the connection as complete.
6. The source node tells the destination node to setup the required routing rules (see section B), and marks the connection as fully setup.

Note that if there's a failure anywhere along this process, the state gets torn down and the process fails. The file in /tmp is kept up to date, and is also locked to prevent race conditions with multiple instances of the client or server running concurrently.

B. IPTables and Conntrack

The linux netfilter subsystem is surprisingly complicated, but is fairly well documented by [8], [10], [11], and [12]. To actually route the packets from currently active connections from the source to the WireGuard tunnel, and from the wiregurad tunnel to the source, we do the following on the source node:

1. Obtain the container's internal IP, and port mappings from podman.
2. Setup a tc netem latency qdisc [13] on the WireGuard interface, so that outgoing packets can get buffered for up to a minute while we do the migration.
3. Setup an iptables filter rule that blocks all outgoing traffic from the container. This is required to prevent issues later when we rewrite the connection tracking information. We used the following rule: `iptables -t filter -I FORWARD -p tcp -s {container_ip} -j DROP`, with one corresponding to UDP as well.
4. We rewrote all the connections in the Linux conntrack system (used to apply and reverse NATs), to set the destination as the WireGuard IP instead of the container IP. That way, all existing connections send their packets over WireGuard.
5. We can optimally add a DNAT rule that causes NEW connections to also be routed to the WireGuard interface. This should be used only if you want to still connect to the migrated container as if it were still on the old host.
6. We perform the final migration phase here (dump, transfer, and restore checkpoint)
7. We can remove the filter rule now since the container is no longer running on the source host.
8. We can finally remove the qdisc so the queued packets now get sent to the migrated container.

We also need to make sure the packets being sent out of the migrated container get sent over the WireGuard tunnel, to the original client. The conntrack rules on the source node handle the WireGuard tunnel <> client path, we just need to make sure packets going out of the container go to WireGuard.

1. During the global initialization of WireGuard, we also add an iptables rule to restore the connection mark for the connection (as assigned by the conntrack subsystem): `iptables -t mangle -A PREROUTING -j CONNMARK --restore-mark`
2. Per WireGuard connection, using the file in /tmp, we generate a nonconflicting packet mark, let's refer to it as *mark* and ip routing table ID to use to route packets to the WireGuard interface, *table*. Here are the commands we use:

```
ip rule add fwmark {mark} lookup {table}
  ip route add default dev {wg} table {table}
```

3. We create a rule that, if a packet has mark *mark*+1, then we set the packet mark to *mark*: `iptables -t mangle -A PREROUTING -m mark --mark {mark + 1} -j MARK --set-mark {mark}`
4. We have a rule that for packets coming in from the WireGuard interface, we set the connection mark to *mark* + 1. This ensures that the packet doesn't get hairpinned out of the interface, but when combined with the save rule, the above rule, and the routing/fwmark, ensures that connections that see packets coming out of

the WireGuard interface will send replies to the WireGuard interface. Here are the rules:

```
iptables -t mangle -A PREROUTING -i {wg} -j MARK --
set-mark {mark + 1}
  iptables -t mangle -A PREROUTING -i {wg} -j CONNMARK
--save-mark
```

We were inspired to do this by the stackoverflow post at [17].

Then, on the actual restore, we have to add several conntrack connections to the connection tracking table, so that packets correctly get routed from the WireGuard interface to the docker container's veth pair. Specifically, we take all the connection entries from the source host, and just rewrite them to have the WireGuard interface's IP as the pre-NAT destination address, then add to the destination node. Since podman keeps the container's IP the same, this works.

C. Multimigration

Because of the way this is designed, it is actually possible to migrate a container multiple times, while having connections from the original container persisted. We will walk through the path of a packet sent to a double-migrated container to demonstrate that this is the case.

1. The packet arrives from the client at the original node, where it is routed to the WireGuard interface because of the conntrack entry that already exists.
2. The packet exits the WireGuard interface, arriving at the first migration destination, gets marked with (for example) 2, that mark gets saved. The second migration modified the conntrack entries on the first destination, so likewise this packet enters the second WireGuard interface.
3. The packet exits the second WireGuard interface, arriving at the second migration destination, gets marked with 2 again (we can reuse marks accross different hosts), and gets routed by a conntrack entry to the podman container, so it enters the veth pair.
4. The container's response exits the veth pair, and the conmark restore rule sets the packet mark to 2 again. The next rule sees the mark is 2 (thats a match), so it sets the mark to 1.
5. In the routing phase, the ip fwmark rule sees a mark of 1, and so throws the packet to routing table 1031, which sends it out the seconds WireGuard interface.
6. Exiting the second WireGuard interface and arriving at the first migration destination, the same process repeats, which results in it getting sent to the first WireGuard interface.
7. Finally, it exits the first WireGuard interface, arriving at the original node, where it then gets sent back to the client.

V. Evaluation

A. Testing - Criteria

We consider our solution "working" if it meets the minimum criteria:

1. The migration process is able to move a container with no I/O work from one node to another node without disrupting the execution state.
2. The migration process is able to move a container that is currently involving in a network connection with an external resource from one node to another without disrupting the execution state and network state.

B. Testing - Procedure

We validate our solution with three parts:

1) *Part 1 - Incrementing Counter:* We create a container with a simple image that has an integer stored in memory, and there is an infinite loop that keeps printing out that number every second and increase that number by 1.

We proceed to migrate this container from source node to destination node. If the destination node's restored container resumes printing from the last number the container on source node has printed out, then we can claim this migration has happened without disrupting the container's execution state.

2) *Part 2 - Network Persistence:* We create a container that acts as TCP echo server that replies the packets it received from client back to client. Then We create a simple TCP echo client which establishes a connection with our previous created echo server. We first hop on the TCP client and connect it to the echo server with provided address and port, then we can send few packets to show the echo server is echoing back, which means the client and server are indeed connected. Then we start migrating the container - after the container is being migrated the client will still be able to send packets and get echoing back - which shows the connection is not interrupted throughout the migration process. This backs our claim that our migration process can happen without disrupting the network state.

C. Network Performance tests

We decided to use iperf3 [7] because it supports a wide range of network performance, is highly configurable, and can produce machine-readable output (json). We ran two iperf tests. On both tests, we migrated twice: once at approximately 20 seconds, and again at about 40 seconds. We only included test runs where the first migration completed within 20 seconds. The first iperf test was a bidirectional iperf test using tcp, with the following command line options:

```
  iperf3 -J --bidir -t 80 -c {CONTAINER_HOST_IP} -p
5201 --get-server-output
```

We primarily are looking at the bandwidth from this test, but iperf also exposes the round trip time (rtt) for these tests, which is fairly useful. The second test is a bidirectional iperf test using udp that stresses the packets per second. Though research, we determined the best way was to set the packet length to 16 bytes in iperf (the minimum possible), then run a test with unlimited bitrate, so iperf will attempt to send as many packets as possible:

```
  iperf3 -J --bidir -t 80 -c {CONTAINER_HOST_IP} -p
5201 -u -b 0 -l 16 --get-server-output
```

For this test, we'll be looking at the packets received by the client, the packet loss rate associated with the previous metric, and the jitter.

For both these tests, we used the docker.io/networkstatic/iperf3 image as the server.

VI. Results

A. TCP iperf

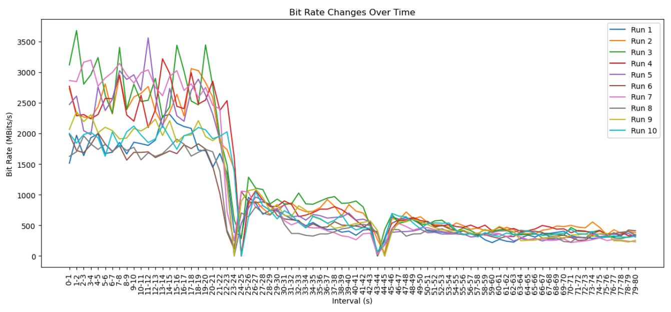

Figure 4: Bitrate (throughput) in MBit/s over time

As can be seen in the figure, we do experience drops in throughput as a result of the migration, and then a continuing drop caused by the extra hop. This makes sense with how AWS calculates bandwidth; the node which originally had the container now must forward all the packets to a new container, so its effective bandwidth should halve. The rest of the drop could be relating to WireGuard (encryption bottleneck?) or maybe tcp congestion control related. We see a similar but less pronounced effect after the second migration.

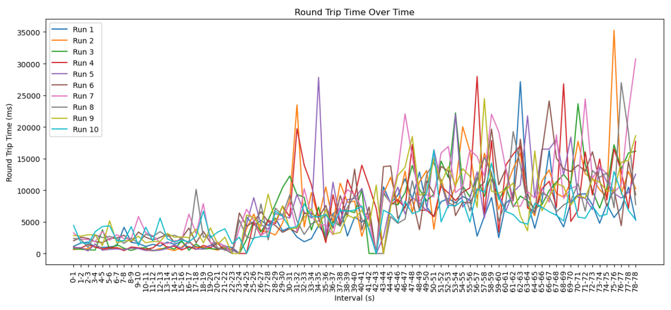

Figure 5: Round trip time (ms) over time

Notably, the latency spikes substantially after the first set of migrations, and then becomes more varied after the second migration. This further supports the hypothesis that tcp congestion control might explain some of the slowdown.

B. UDP iperf

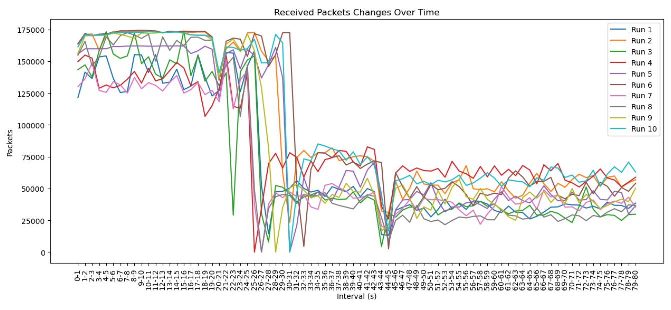

Figure 6: Packets per second over time

Note that the above chart does take into account packet loss. A similar trend to TCP is noticed, though the drop in packets per second does appear to be less extreme. Because UDP doesn't have congestion control, the sending rate remains the same, which means all the "missing" packets are being dropped.

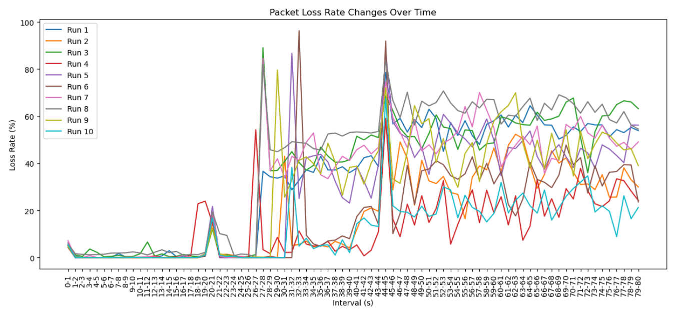

Figure 7: Packet loss rates over time

And, as the above graph shows, the packet loss does increase sharply when we do migrations. Interestingly, the first few runs have low packet loss after the first migration. We had to restart the VMs after running the first few runs, so the lower loss could be explained by the source and the first migrator VM being colocated (or close within the datacenter).

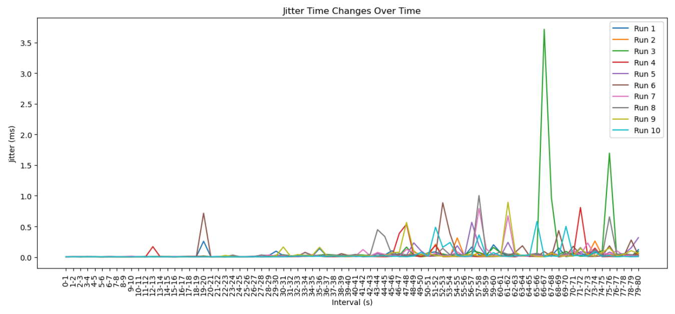

Figure 8: Jitter (ms) over time

Lastly, it is interesting to note that the more migrations we do, the higher the jitter gets. At 0 migrations there is effectively no jitter, there is a slight amount from 20 seconds to 40 with one migration, and we start to get noticeable spikes after 40.

C. Hot vs cold start

We also ran 5 cold start tests, and 10 hot start tests, using the `docker.io/cjimti/go-echo` container. For the cold starts, we got a total migration time of (on average) 49 seconds, with all results falling between 48 and 50. We estimate the first run is slow because of some kind of caching. This is because on hot/warm starts, we get a total migration time of 1.14 seconds on average.

For downtime, cold starts are still dominated by the hot starts: they have 24 seconds of downtime on average, most of which is taken up by the restoring, which furthers the hypothesis that this is cache related. Hot/warm starts, by comparison, have on average 0.85 seconds of downtime. All of the iperf tests were run on hot/warm starts.

VII. The project experience

A. Difficulties

1) *The live demo:*

The live demo was in fact working the night before.

We diagnosed the issue to multiple firewall rules interacting in an untested configuration. The night before, when we tested the demo, one of us used the world-routable IP address of the destination node and one of us used the aws VPC address. This led to the creation of two WireGuard tunnels terminating at the same (destination) node, which was not a configuration that we had tested. In our firewall rules, originally, instead of creating the --restore-mark rule once per vm globally, we created it once per WireGuard connection. This meant that packets ariving from the WireGuard interface corresponding to the WAN IP were getting their mark rewritten, saved, and correctly routed to the container. However, when replies were getting sent from the container, the first set of rules sets the mark correctly so it gets routed to the WireGuard interface. However, the second set of rules then re-loads the mark from conntrack. As a result, the packet is not routed to the WireGuard interface, and goes to the WAN, where the client receieves it, doesn't match it to a connection, and drops it.

Notably, because for the local ip case, its firewall rules were added second (and iptables processes firewall rules in order), so it is not suceptible to this bug, which explains why both our tests succeeded the night before, but then when we attempted to run it using the WAN IP during the demo it failed.

2) *Package versions:*

One issue we had when initially using Ubuntu was that it was near impossible to get up to date software that played well together. We used Arch Linux [15], as it was a rolling distry with AMIs present on AWS [16]. We had to use the standard kernel, as the ec2 optimized kernel was compiled without support for CRIU pre-dumps or WireGuard.

3) *Lack of access to connection tracking:*

We interfaced through the Linux connection tracking subsystem primarily using the command line tool conntrack. This is fine for typical use (listing, flushing, deleting connections), but it fails to provide atomic operations to update the source ip, dest ip, and other relevant fields. Instead, we need to delete the connection, rewrite it, and re-create it. This opens us up to all sorts of race conditions.

We also have a use case for creating multiple "conflicting" connections. This would allow us to remove the filter rule, and technically achieve 0 packet drops. However, the CLI simply fails and says operation not permitted. Calling the libnetfilter_conntrack [14] library directly might allow us to bypass these restrictions and fix issues in our code. The logic for parsing is also very dependent on the output format of conntrack, which is not ideal.

B. *What worked well*

1) *Learning about and using the netfilter subsystem:*

Using the Linux netfilter subsytem, which was designed from the start to be extensible and support multiple protocols, allowed us to rapidly add features. For example, preliminary UDP support only required a two-digit number of code changes. By building using the "blocks" given to us by the netfilter diagram [8], we were also able to rapidly add support for multiple migrations, beacuse we injected our code into the routing level as opposed to the level of individual interfaces.

2) *WireGuard tunnels:*

By using WireGuard which is widely-used and "battle-tested", we didn't have to worry about HOW the packets got from our source to our destination node, we just had to worry about what the packets did when they got there. Because AWS keeps most ports open by default, we didn't have to deal with NAT [9].

3) *Automation of testing and setup:*

Initially, most of the migration was a manual process, documented as a list of commands to run. This was inefficient and slow. Automating the end-to-end process massively improved our efficiency, and improved repeatability.

Additionally, for the iperf tests, although it was difficult to get synchronized at exactly 20 seconds into the test, we were able to get way closer than we would if we did it manually. Unfortunately, limitations inherent to the way iperf outputs did not allow us to get exact time intrevals. Automating the iperf tests also made it much easier to actually collect data, since it's a very repetitive process that takes a while when done manually.

C. *What didn't*

1) *Our initial lack of knowledge of iptables:* We had several issues researching our initial implementation of live connection migration. The attempt we had could migrate containers, and you could make new connections to the migrated container from the source node, but used SNAT, which, by rewriting the source ip, made it impossible to migrate live connections.

Fixing that issue led to our new issue: how to get the responses from the container to go through the WireGuard interface. We attempted to use the forward table to do this, as the netfilter [8] diagram makes one think it is possible to reroute packets using a FOWARD rule. Unfortunately, that is not the case, by debugging I determined that that check did not get run for some reason. Eventually, we discovered a stackoverflow post that provided a sane way to implement what we wanted [17].

2) *Heavy usage of iptables NAT:* Our initial implementation of network connection migration made heavy use of DNAT rules, and conntrack flushes (emptying the conntrack table), to fully migrate the network connection. This was really bad, since it led to lots of packets getting lost, or getting sent to the wrong place, causing erroneous connection closures or RST packets. We solved most of these issues by directly editing using the conntrack command line tool. However, we initially kept the DNAT rule in because that way, if someone wanted to establish new connections to the migrated node via the old node, it would be possible to do so. This had some drawbacks, mainly being that you couldn't start a new container using the same host port on the old node, and you needed to clean up the DNAT rule at some point. As a result, we ended up removing it.

D. *Why we changed plans*

The final plan we cane up with was notably different from what we were exploring during the proposal and progress report. The main changes were made with respect to the network, and the filesystem monitoring.

1) *Network:* We initially were going to try to use eBPF [28] and DPDK [29] for the packet routing, but we abandonded this decision after further thought. The main reason behind this choice was at the time, Jacob was (and still is) working with DPDK and eBPF in his job, and "when you have a hammer, everything looks like a nail" [3]. We quickly realized this would be a bad idea: it would involve implementing a custom overlay (rearranging packet contents), and would require that we add at least some data to the packet, which would cause many issues (how do we distinguish our custom "overlay" protocol from normal traffic while still getting it routed over AWS/the internet, how do we deal with path MTU). This led

us to adopt WireGuard, as it provides lots of that functionality effectively built-in.

From then, we could have still used eBPF in packet routing, however any solution using that (due to our relative inexperience) would likely be bespoke and difficult to extend, which would make adding multiple migrations, migrating to multiple hosts at once, or adding support for more protocols much harder. As a result, we decided to explore the Linux kernel's firewall, connection tracking, and routing stack. This also allowed us to use bridge networks instead of macvlan or ipvlan, because it is likely AWS's network wouldn't enjoy us spoofing IP or MAC addresses [4] [5].

We think the "correct" way to implement network connection migration would be to have an overlay network for containers, assign each container a unique ip, and use the overlay network to route to the "new" host after a migration. However, implementing that for our project would be too large in scope, and wouldn't work for the average user's use case, which is typically containers behind a bridge network with ports forwarded. Interestingly, our solution is starting to look increasingly like an overlay network, and support for more complicated topologies (migrate back to source, migrate one container from node A to B, then a different one from node B to A), would likely increase the resemblance.

2) *Container Volume Monitoring:* We were initially going to try to do a volume content patching on the destination node so that if the source has written during the migration it would still be fine. However, since the actual checkpoint happens during the final stage of transfer and we would have already stopped the container at this point already, there is not really a point to monitor the file change anymore since there cannot be more writes available - the container has been stopped.

E. Known Problems

1) *Privileged Requirement:* CRIU checkpoint and restore requires privileged environment in order to work. Unfortunately, we don't see that changing; lots of the core capabilities that we require (arbitrary configuration of network interfaces, firewall rules, memory dumps, and various pieces of kernel state) fundamentally cannot be offered to an unprivileged user without giving them so many dangerous powers that they might as well have root.

2) *Limited Kernel Support:* Pre-checkpoint feature in Podman calls CRIU, which utilizes the dirty tracking feature in Linux kernels to produce memory delta. However, some kernels do not have dirty tracking support[2] which partially limits our solution to only work in certain kernels.

3) *Downtime:* With respect to network downtime, during the first set of connection tracking table rewrites, we have to drop all packets sent from the container, to prevent the connection tracking table from entering a bad state. It appears it still sometimes enters a bad state (see UDP below), but it does so less often, and did not do so on any tcp tests (potentially due to some interaction with ACKs or flow control algorithms).

4) *Source machine dependency:* Although the resulting container doesn't depend on the original host insofar as new connections to the container can only be made to the new host, migrated network connections currently have to go through the original node, because the client should not be aware that the container was migrated.

5) *UDP:* Near the end of the project, we managed to get UDP connection migration somewhat working. However, it has some amusing failure cases; one bug we fixed involved unidirectional (container to client) udp flows not being migrated on the second migration for the container. More relevantly, at high packets per second, migrating causes the (iperf) container to become unable to receieve packets on its test connection. Additionally, at high throughputs, migrating causes a socket error which terminates the test early. We think these issues may be caused by weird interactions between the high packet/throughput rates, our latency qdisc (others have reported issues that may be related) [6], and race conditions in how we handle conntrack.

F. Overall project result

We believe that the project was overall a success, as measured by the milestones we set out to complete at the start of the project. We completed milestones 0-3. Milestone 1 was made easy by the fact that CRIU supported migrating with open file descriptors, including, notably, open FDs with pending writes. We didn't mention this in the testing or evaluation sections because the test we used was weird and required substantial manual work. We implemented milestone 3 before milestone 2, because after abandoning the idea of containers getting their own IPs, the idea of sending an arp request [27] no longer made sense, so there was no meaningful difference between migrating within a LAN and over the WAN. Milestone 4 is ultimately impractical within the scope of a class project, but would still be interesting as a standalone project. We definitely leaned more into the network aspect then the minimizing downtime aspect, as that was subjectively more interesting then (for example) running multiple checkpoints until the sizes converge to ensure minimal downtime.

Our source code is available at https://github.com/nadecancode/live-container-migration. The wg_migration_test.py file is what we used to automate iperf tests, the agent-client folder contains the client part of the agent, the agent-server folder contains the agent server, container/example contains some of the containers we used in testing, and common/net.py is what contains a lot of the core logic for network connection migration. We use the poetry python package manager to ensure sane dependency management. We used the following (non-exhaustive) list of packages in addition to what came preinstalled with our AMI: `wireguard-tools tcpdump base-devel neovim kitty linux podman inetutils conntrack-tools iproute2 runc`.

G. Further work

We could get UDP working, explore the CPU impact of migrating under high pps/throughput loads, explore the impacts of migrating over high bandwidth (10G+) links, and fully integrate netfilter into our code. We could also explore migrating workloads that are more complicated from a kernel perspective, like postgres databases with mmapped pages.

VIII. Related Works

https://CRIU.org/index.php?title=Disk-less_migration - Disk-less Migration with CRIU

https://CRIU.org/Iterative_migration - Iterative migration with CRIU

https://www.usenix.org/system/files/atc21-planeta.pdf - Information on container migration process that involve network requests

IX. Appendix

[1] Supports a RESTful API for the Libpod library (5.0.0) https://docs.podman.io/en/latest/_static/api.html

[2] [CRIU] A problem of dirty tracking https://lists.openvz.org/pipermail/CRIU/2019-November/044802.html

[3] Law of the Instrument https://en.wikipedia.org/wiki/Law_of_the_instrument

[4] Docker MacVLAN in AWS fails! https://github.com/moby/moby/issues/30772

[5] Multiple IP addresses on AWS https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/MultipleIP.html

[6] Add latency with tc without restricting bandwidth https://stackoverflow.com/questions/31056559/add-latency-with-tc-without-restricting-bandwidth

[7] IPerf https://iperf.fr/

[8] Netfilter packet flow diagram https://en.wikipedia.org/wiki/File:Netfilter-packet-flow.svg

[9] How NAT Traversal Works https://tailscale.com/blog/how-nat-traversal-works

[10] Conntrack manpage https://manpages.debian.org/jessie/conntrack/conntrack.8.en.html

[11] Iptables manpage https://man7.org/linux/man-pages/man8/iptables.8.html

[12] Iptables-extension manpage https://www.man7.org/linux/man-pages/man8/iptables-extensions.8.html

[13] TC netem manpage https://www.man7.org/linux/man-pages/man8/tc-netem.8.html

[14] libnetfilter_conntrack https://www.netfilter.org/projects/libnetfilter_conntrack/index.html

[15] Arch Linux https://archlinux.org/

[16] Arch Linux AMIs https://wiki.archlinux.org/title/Arch_Linux_AMIs_for_Amazon_Web_Services

[17] FWmark reflection https://unix.stackexchange.com/questions/416492/is-it-possible-to-force-fwmark-reflection-in-arbitrary-tcp-reply-packets

[18] Python https://www.python.org/

[19] Flask https://flask.palletsprojects.com/en/3.0.x/

[20] Netfilter https://www.netfilter.org/

[21] Linux https://www.linux.org/

[22] Post mortem on the Cloudflare Control Plane and Analytics Outage https://blog.cloudflare.com/post-mortem-on-cloudflare-control-plane-and-analytics-outage/

[23] Podman https://podman.io/

[24] Netavark https://github.com/containers/netavark

[25] runc https://github.com/opencontainers/runc

[26] iproute2 https://wiki.linuxfoundation.org/networking/iproute2

[27] https://cseweb.ucsd.edu/~yiying/cse291j-winter20/reading/VM-Migration-Replication.pdf

[28] eBPF https://ebpf.io/

[29] DPDK https://www.dpdk.org/

[30] CRIU https://CRIU.org/Main_Page