1 VM Migration Basics

Migrating VMs can be very useful, for instance, when the need to upgrade the hypervisor hardware or software arises. VM migration is also a key feature in cloud computing and green IT.

Task 1

Read about VM migration in KVM and

(a) Describe the differences between cold (or off-line) migration and live migration.

Cold and live migrations are two approaches for transferring virtual machines (VMs) from one host to another, each designed with different purposes and impacts on service continuity.

Cold migration, often referred to as off-line migration, requires the VM to be completely powered off before the migration begins. In this process, the VM's entire state, including its memory and CPU configuration, is saved and then transferred to the target host, where it will be restarted after the transfer completes. Because the VM is shut down during this move, there are no active memory or data connections to manage between the source and destination hosts, simplifying the transfer process. This approach reduces complexity and network demands, making it a relatively fast and straightforward method. However, the VM is entirely unavailable during the move, resulting in downtime. This downtime means cold migration is best suited for situations where a period of inactivity is acceptable, such as scheduled maintenance or planned downtime, and when continuous availability of the VM is not crucial.

In contrast, live migration allows a VM to be moved between hosts without interruption to its services, which means it continues to operate and remain accessible throughout the migration. This approach is essential in environments where even minimal downtime would disrupt critical operations. In live migration, the VM's memory, CPU state, and storage data are synchronized in real-time between the source and destination hosts to ensure consistency and continuity. To achieve this, live migration typically employs either a pre-copy or post-copy technique. The pre-copy method begins by transferring memory in stages before finally switching the VM to the target host, allowing minimal downtime but requiring potentially higher network usage to keep memory synced. Post-copy, on the other hand, initiates the move earlier by running the VM on the target host while transferring remaining memory as needed, reducing network strain but introducing a risk of faults if the VM accesses memory that hasn't yet been transferred.

Live migration, while ideal for high-availability environments, requires significant network bandwidth and computational resources to handle the real-time state transfer, especially for large or highly active VMs. Thus, it is more resource-intensive and complex than cold migration, necessitating careful planning and robust infrastructure. In summary, while cold migration is suitable for scenarios where downtime is manageable and prioritizes simplicity, live migration is designed to maintain service continuity for high-availability requirements, albeit with greater demands on resources.

(b) What mechanism makes live migration almost instantaneous under light load?

The mechanism that enables near-instantaneous live migration under light load is the pre-copy migration technique, which relies on iterative memory transfer to keep the source and destination VM states closely synchronized, especially when system activity is minimal.

In pre-copy migration, the process begins by copying the entire memory state of the VM from the source to the destination host. While this memory copy is occurring, any memory pages (specific segments of memory) modified by the VM's processes are tracked. In subsequent iterations, only the modified memory pages are transferred, rather than re-copying the entire memory. This iterative copying continues until the amount of modified data is very small or negligible, enabling a final transfer of only the remaining few changes just before the VM is switched to the destination host.

Under light load, where the VM is not generating many memory changes, fewer pages need to be re-sent in each iteration. This reduces the number of iterations needed and minimizes the final memory sync, allowing the migration to complete almost instantaneously. Since minimal data transfer occurs in each iteration, the process is quick and efficient, achieving near-zero downtime for the VM's services.

Hint: https://libvirt.org/migration.html and https://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines

Task 2

What are the technical requirements to be able to coldly migrate VMs, and why?

The primary requirements to be able to do a cold migration include, hardware compatibility, shared or accessible storage, consistent network configurations, and compatible hypervisor settings.

Hardware Compatibility between the source and destination hosts. Both hosts should have similar or compatible CPU and memory configurations. Although the VM is shut down during cold migration, having compatible CPU architectures prevents errors or conflicts when the VM restarts on the destination host, ensuring the VM's saved state can be processed without issues.

Shared Storage or Disk Transfer can be used, although Shared storage is ideal for cold migration, as it enables faster migration with minimal network load. Shared storage solutions, such as Network File System (NFS), Internet Small Computer Systems Interface (iSCSI), or Storage Area Network (SAN), allow both the source and destination hosts to access the VM's disk images directly. This setup eliminates the need to transfer large disk files over the network, significantly reducing migration time and simplifying the process.

In environments without shared storage, cold migration is still possible but may introduce longer downtime. Without shared storage, the VM's entire disk image must be transferred over the network, which can increase the time and bandwidth required. Using tools like scp or rsync for the disk transfer, along with careful network planning, can make this method effective. This alternative is especially useful in smaller setups or where shared storage is not feasible.

Consistent Network configuration compatibility between hosts is crucial to enable the VM to reconnect seamlessly after migration. Both the source and destination hosts should have matching network settings, including virtual switches, IP addresses, and security policies. Properly aligned virtual network configurations prevent potential connectivity issues, allowing the VM to access network services immediately upon restart without needing additional reconfiguration.

Hypervisor and VM Compatibility Consistency in hypervisor versions and settings across both hosts is necessary to ensure the VM’s configuration is accurately interpreted on the target host. Matching hypervisor configurations help maintain compatibility with the VM’s virtual hardware components, such as network adapters and disk controllers, reducing the risk of errors related to virtual device configuration and minimizing the need for manual adjustments after migration.

These requirements ensure the integrity of the VM’s data, minimize configuration issues, and streamline the cold migration process, allowing the VM to start up smoothly on the new host with minimal downtime or technical complications.

Task 3

There are two primary options when live migrating, using shared storage or copying the disks. What are the technical requirements of each and why?

When live migrating virtual machines (VMs), the two primary options are either using shared storage or copying the disks. Each option has its own technical requirements that are essential for ensuring data consistency, performance, and minimal downtime during the migration.

For shared storage setups, both the source and destination hosts need to have direct access to the same storage system, typically through a networked storage solution like NFS (Network File System), iSCSI (Internet Small Computer Systems Interface), or a SAN (Storage Area Network). This shared storage setup means the VM’s disk images do not need to be copied during migration, as they are already available to both hosts. With shared storage, the technical requirements focus on ensuring high network bandwidth and low latency, as well as robust redundancy to prevent any storage outages that could disrupt the VM’s data accessibility. This configuration allows for faster migrations with minimal downtime, as the process only involves synchronizing the memory and CPU states without needing to transfer large disk files over the network. Shared storage setups are particularly beneficial in high-availability environments, as they reduce the network load and the time needed for migration.

For the disk-copying option, where the VM’s disk image is copied to the destination host as part of the migration, the requirements are more extensive due to the need to transfer large amounts of disk data in addition to the memory and CPU states. This approach requires a high-speed network connection with ample bandwidth to accommodate both the VM’s memory state and its disk data. Disk copying may use incremental copying methods, where only changes made to the disk since the last transfer are copied, but this still requires robust bandwidth and efficient data transfer protocols. Additionally, the destination host must have sufficient storage capacity to accommodate the VM’s disk image, and ideally, the network infrastructure should support error-checking mechanisms to ensure the integrity of the copied data. Disk-copying methods are generally used when shared storage is not available or feasible but can lead to longer migration times and increased downtime, especially for VMs with large disk images.

In summary, shared storage requires compatible networked storage solutions and high bandwidth for data access, enabling faster migration with minimal downtime. Disk copying, however, requires ample network bandwidth, storage capacity on the destination host, and efficient data transfer methods to handle the movement of large disk files, often resulting in longer migration times.

Task 4

Form a group of two and discuss how you are going to migrate VMs to each other's hypervisor. Set up your systems so you can do both cold and live migrations. Describe your setup in your logs. Hint: Do not use LVM, use qcow2. Remember your eno2.

We successfully migrated the virtual machine 'Guest-01' from our previous lab environment on the tallinn server to the skopje server.

Prerequisites

To ensure the virtual machine boots properly and avoids any potential time synchronization issues, we begin by setting the correct timezone on both tallinn and skopje servers.

Set timezone on both servers.

```
timedatectl set-timezone Europe/Amsterdam
```

Configure the 'eno2' interface. The bridge uses the wan interface 'eno2' that is connected to the OS3 Router which allows us to create a route from the source to destination.

To enable network communication for the migration, the eno2 network interface was configured to support bridging.

The bridge kvmbr0 on tallinn uses the WAN interface eno2, which connects to the OS3 Router. This connection allows us to establish a route from the source (tallinn) to the destination (skopje), enabling seamless VM migration.

On the tallinn server, eno2 was added to the bridge kvmbr0, and a default route was configured to reach the kvmbr0 bridge on skopje. The routing configuration was updated in /etc/netplan/00-installer-config.yaml to ensure proper connectivity for the migration process.

```
network:
  ethernets:
    eno1:
      critical: true
      dhcp-identifier: mac
      dhcp4: true
      nameservers:
        addresses:
          - 145.100.96.11
          - 145.100.96.22
        search:
          - studlab.os3.nl
    eno2:
      dhcp4: false
  bridges:
    kvmbr0:
      interfaces:
        - eno2
      dhcp4: false
      addresses:
        - 145.100.106.193/28
  version: 2
```

We created a dedicated user account named migration on both the tallinn and skopje servers. This user was then added to the libvirt group on each server, granting it the necessary permissions to manage virtual machines through libvirt commands.

```
sudo adduser migration
sudo chmod -aG libvirt migration
```

To enable passwordless SSH access from tallinn to skopje, we generated an SSH key pair on tallinn and configured it for use by the migration user. This setup allows commands to be executed on skopje remotely without prompting for a password, which is essential for automation in the migration process.

```
ssh-keygen -t ed25519
ssh-copy-id -i /home/migration/.ssh/id_ed25519 migration@192.168.0.1
```

For the migration process and SSH access over the OS3 network, the following ports need to be open: Port 22: Required for SSH access, which facilitates secure communication and file transfer. Ports 49152–49216: Required for libvirt migration over the OS3 network. These ports handle the transfer of VM state and memory, ensuring data consistency during migration.

```
sudo ufw allow from 145.100.96.0/20 proto tcp to any port 22
sudo ufw allow from 145.100.96.0/20 to any port 49152:49216 proto tcp
```

To prepare for a fresh migration and ensure the destination server skopje is in a clean state, we stop the virtual machine and reset the volume group on skopje.

```
virsh destroy Guest-01 && \
virsh undefine --domain Guest-01 --remove-all-storage && \
virsh vol-create-as VolumeGroupKVM guest01volume 20G
```

To facilitate repeated migrations for collecting metrics, we created a series of helper scripts. These scripts automate key steps of the migration process, allowing us to perform multiple migrations in a controlled, consistent manner without manual intervention. By using helper scripts, we can seamlessly repeat the migration process as many times as needed, capturing reliable metrics across multiple runs.

Scripts for cold migration

We developed a bash script, `lab2_cold.sh`, to automate the cold migration process. This script simplifies the workflow for migrating the virtual machine by handling each step required to move it from the source server `tallinn` to the destination server `skopje` in a consistent and reliable manner.

The `lab2_cold.sh` script begins by shutting down the virtual machine on tallinn, ensuring data integrity by powering it off before any data transfer. Once the VM is safely shut down, the script proceeds to securely copy the VM’s disk image and configuration files from `tallinn` to `skopje`, making the entire VM, including its storage and settings, available on the destination server. After completing the file transfer, the script registers the VM on skopje and starts it, making the VM operational on the new server and completing the migration process.

```
#!/bin/bash

# Set the destination host and working directory
DEST_HOST="skopje"
WORK_DIR="/home/migration/vm"
```

```
# Start timer
START=$(date +%s)

# 1. Shutdown the Guest-01 on tallin
virsh shutdown Guest-01

# 2. Convert the Guest-01's LVM volume to qcow2 format if not already
        converted
if [ ! -f $WORK_DIR/Guest-01.qcow2 ]; then
   echo "Converting LVM volume to qcow2 format..."
   qemu-img convert -O qcow2 /dev/VolumeGroupKVM/guest01volume
        $WORK_DIR/Guest-01.qcow2
else
   echo "Guest-01's disk is already converted to qcow2 format."
fi

# 3. Dump Guest-01 xml
echo "Dumping VM xml"
virsh dumpxml Guest-01 > $WORK_DIR/Guest-01.xml

# 4. Copy the Guest-01's disk to skopje/var/vms
echo "Copying $WORK_DIR to $DEST_HOST:$WORK_DIR"
scp $WORK_DIR/Guest-01.xml migration@$DEST_HOST:$WORK_DIR/
scp $WORK_DIR/Guest-01.qcow2 migration@$DEST_HOST:$WORK_DIR/

# 5. Convert the Guest-01's disk to LVM format on skopje
echo "Converting Guest-01's disk to LVM format on skopje"
ssh migration@$DEST_HOST "qemu-img convert -O raw $WORK_DIR/Guest-
        01.qcow2 $WORK_DIR/Guest-01.raw"

# 6. Upload the Guest-01's disk to guest01volume
echo "Uploading Guest-01's disk to guest01volume"
ssh migration@$DEST_HOST "virsh --connect qemu:///system vol-upload --
        pool VolumeGroupKVM guest01volume $WORK_DIR/Guest-01.raw"

# 7. Define the Guest-01 on skopje
echo "Defining Guest-01 on skopje..."
# ssh migration@$DEST_HOST "virsh --connect qemu:///system define
        $WORK_DIR/Guest-01.xml"
virsh migrate --offline --persistent Guest-01
        qemu+ssh://migration@$DEST_HOST/system

# 8. Start the Guest-01 on skopje
echo "Starting Guest-01 on skopje..."
ssh migration@$DEST_HOST "virsh --connect qemu:///system start Guest-
        01"

# 9. End timer
END=$(date +%s)

# 10. Calculate and print the duration
echo "Time taken: $(($END - $START)) seconds"
```

We created a script named `reset.sh` to remotely stop the virtual machine on skopje and reset its volume group from `tallinn`. This script allows us to prepare the destination server `skopje` for repeated migrations by ensuring it is in a clean state after each migration cycle.

```
#! /bin/bash

# Set the destination host and working directory
DEST_HOST="skopje"
WORK_DIR="/home/migration/vm"

# 1. Stop the Guest-01 on skopje
echo "Stopping Guest-01 on skopje..."
ssh migration@$DEST_HOST "virsh --connect qemu:///system destroy
        Guest-01"

sleep 5

# 2. Undefine the Guest-01 on skopje
echo "Undefining Guest-01 on skopje..."
ssh migration@$DEST_HOST "virsh --connect qemu:///system undefine
        Guest-01 --remove-all-storage"

# 3. Create a new volume
echo "Creating a new volume"
ssh migration@$DEST_HOST "virsh --connect qemu:///system vol-create-as
        VolumeGroupKVM guest01volume 20G"

# 4. Remove everything from /home/migration/vm
echo "Removing everything from /home/migration/vm"
rm -f $WORK_DIR/*

# 5. Start the Guest-01 on tallin
echo "Starting Guest-01 on tallin..."
virsh start Guest-01
```

By executing this process from `tallinn`, `reset.sh` enables us to efficiently reset `skopje` to a baseline state, ready for the next migration. This approach is essential for collecting consistent metrics across multiple migrations, as it ensures that each migration begins with the same starting conditions.

Task 5

Together think of a definition of the downtime of a VM, and how to best measure that downtime. Write down your definition and measurement method(s) for both

We defined the definition of downtime for the migrations as follows:

(a) Cold migration

For cold migration, where the only active service on the VM was ssh, we used the availability of the SSH socket to determine downtime. The start time was recorded when the VM began shutting down, causing the ssh service to stop and the socket to become inaccessible. The end time was marked when the ssh service resumed and the socket became available again, signaling that the VM was fully operational.

Thus, downtime in this context is defined as the total elapsed time between when the SSH socket first goes down and when it becomes accessible again. This approach provides a clear and measurable indication of service interruption specifically for cold migration.

(b) Live migration

In live migration, we measured downtime by monitoring the availability of the ssh service, which was the only active service on the VM. Downtime began at the final phase of migration, specifically when the VM's state transfer started, causing the ssh service to stop and the socket to become unavailable. Downtime ended when the ssh service resumed on the destination server, and the socket was available again.

This definition of downtime captures the exact period when the VM is temporarily inaccessible during migration. To accurately measure this, we ran a Python script on one of the servers to continuously check the VM's connectivity on port 22, logging the exact moments when the socket went down and later came back up. This approach provides precise downtime metrics for both live and cold migration scenarios, allowing for consistent comparisons of service availability across different migration types.

Task 6

Perform cold migrations with your partner, and measure the downtime. Do not take just a single measurement! Compute mean and median of your chosen metric. Hint: Make sure the details match when migration e.g.

1. The storage pool, use identical names/pool based on a directory.
2. access to the disks, qemu user will require access to the storage pool location/directory on disk.
3. parameters/capabilities are available on the remote node e.g. machine type.
4. The guest does not use hardware attributes that cannot be migrated e.g. cloud-init cdrom. Hint: virsh change-media –eject.
5. https://libvirt.org/formatdomain.html

Alternatively, you can provide a modified domain.xml definition file when doing the migration to provide modified options/locations. This is more advanced!

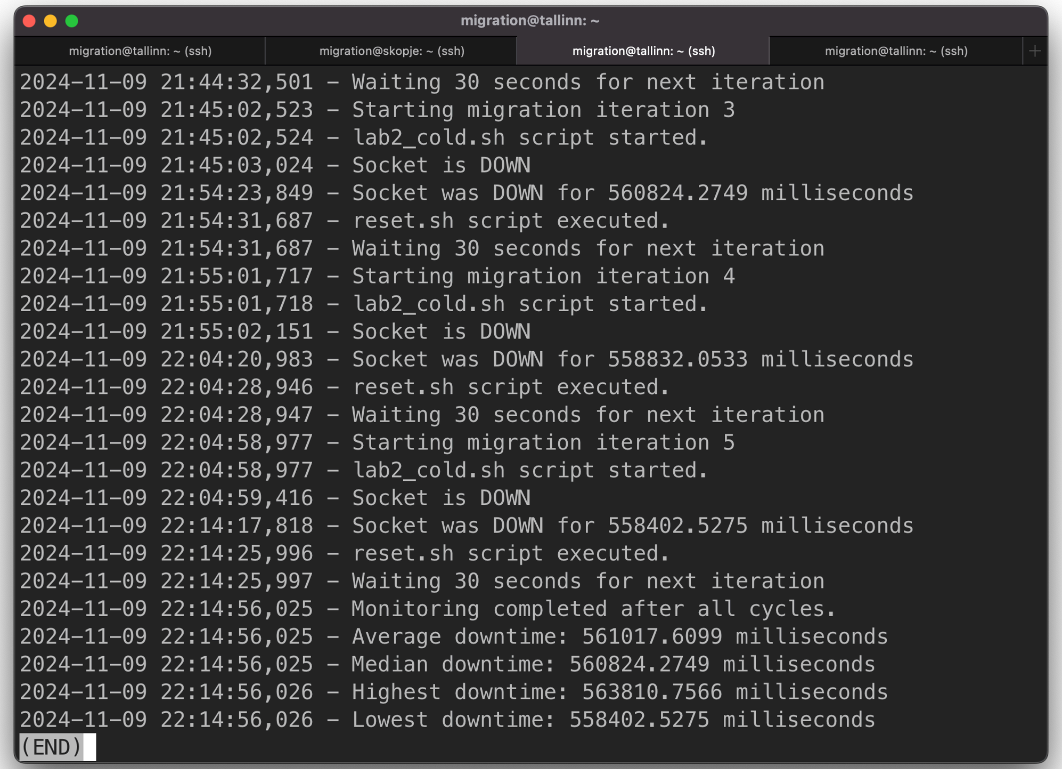
We performed the cold migrations five times from `tallinn` to `skopje` using the Python script measure_downtime.py. By running the command `python3 measure_downtime.py lab2_cold.sh 5`, the script was configured to execute the specified migration script `lab2_cold.sh` and reset scripts automatically, repeating the entire process five times.

The measure_downtime.py script automates each step of the migration, including initiating the migration, measuring downtime, and resetting the environment on `skopje` after each iteration. This setup ensures that each migration follows a consistent workflow, allowing us to capture precise and comparable downtime metrics across all five migrations. Automating these repeated migrations allowed for efficient data collection with minimal manual intervention, contributing to more reliable performance metrics for cold migrations.

```python
import socket
import time
import subprocess
import logging
import statistics
import argparse

parser = argparse.ArgumentParser(description="Monitor socket and run
        migration scripts.")
parser.add_argument("script", help="Migration script to use.")
parser.add_argument("iterations", type=int, help="Number of iterations
        to run the script.")
args = parser.parse_args()

# Target host, port, and settings
HOST = '145.100.106.194'  # IP address to monitor
PORT = 22  # Port to monitor
TIMEOUT_MS = 100  # Connection timeout in milliseconds
CHECK_INTERVAL_MS = 50  # Interval for checking the socket status
LOG_FILE = "monitoring.log"  # Log file path

# Configure logging
logging.basicConfig(
    filename=LOG_FILE,
    filemode='a',  # Append to the log file
    format='%(asctime)s - %(message)s',
    level=logging.INFO
)

# Create a console handler for logging
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_handler.setFormatter(logging.Formatter('%(asctime)s - %
        (message)s'))

# Add the console handler to the root logger
logging.getLogger().addHandler(console_handler)


# Check if a socket is reachable
def socket_reachable(host, port, timeout_ms):
    """
    Check if a socket at (host, port) is reachable within timeout_ms.
    Returns True if up, False if down.
    """
    timeout_sec = timeout_ms / 1000.0
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
        sock:
            sock.settimeout(timeout_sec)
            sock.connect((host, port))
        return True  # Connection succeeded
    except (socket.timeout, socket.error):
        return False  # Connection failed


# Initialize the counter and list to store downtime durations
count = 0
downtime_durations = []

logging.info("########################################")
while count < args.iterations:
    # Run migration script in the background
    logging.info(f"Starting migration iteration {count + 1}")
    subprocess.Popen(f"./{args.script}")
    logging.info(f"{args.script} script started.")

    # Monitor the socket status and log downtime
    down_start_time = None
    while True:
        while socket_reachable(HOST, PORT, TIMEOUT_MS):
            # Delay before the next check
            time.sleep(CHECK_INTERVAL_MS / 1000)

        down_start_time = time.perf_counter_ns()
        logging.info("Socket is DOWN")

        while not socket_reachable(HOST, PORT, TIMEOUT_MS):
            # Delay before the next check
            time.sleep(CHECK_INTERVAL_MS / 1000)

        # Stop counting downtime as soon as socket is back up
        down_duration = (time.perf_counter_ns() - down_start_time) /
        1_000_000
        downtime_durations.append(down_duration)
        logging.info(f"Socket was DOWN for {down_duration:.4f}
        milliseconds")

        # Run reset.sh until and wait until its completed
        subprocess.run("./reset.sh", check=True)
        logging.info(f"reset.sh script executed.")

        count += 1
        break

    # Wait for 30 seconds before the next iteration
    logging.info(f"Waiting 30 seconds for next iteration")
    time.sleep(30)

# Calculate metrics
if downtime_durations:
    average_downtime = sum(downtime_durations) /
        len(downtime_durations)
    median_downtime = statistics.median(downtime_durations)
    highest_downtime = max(downtime_durations)
    lowest_downtime = min(downtime_durations)

    # Log metrics
    logging.info("Monitoring completed after all cycles.")
    logging.info(f"Average downtime: {average_downtime:.4f}
        milliseconds")
    logging.info(f"Median downtime: {median_downtime:.4f}
        milliseconds")
    logging.info(f"Highest downtime: {highest_downtime:.4f}
        milliseconds")
    logging.info(f"Lowest downtime: {lowest_downtime:.4f}
        milliseconds")
else:
    logging.info("No downtime recorded during monitoring.")
```

To run the script:

```
python3 monitor_downtime.py lab2_cold.sh 5
```


migration@tallinn: ~
migration@tallinn: ~ (ssh)
migration@skopje: ~ (ssh)
migration@tallinn: ~ (ssh)
migration@tallinn: ~ (ssh)

```
2024-11-09 21:44:32,501 - Waiting 30 seconds for next iteration
2024-11-09 21:45:02,523 - Starting migration iteration 3
2024-11-09 21:45:02,524 - lab2_cold.sh script started.
2024-11-09 21:45:03,024 - Socket is DOWN
2024-11-09 21:54:23,849 - Socket was DOWN for 560824.2749 milliseconds
2024-11-09 21:54:31,687 - reset.sh script executed.
2024-11-09 21:54:31,687 - Waiting 30 seconds for next iteration
2024-11-09 21:55:01,717 - Starting migration iteration 4
2024-11-09 21:55:01,718 - lab2_cold.sh script started.
2024-11-09 21:55:02,151 - Socket is DOWN
2024-11-09 22:04:20,983 - Socket was DOWN for 558832.0533 milliseconds
2024-11-09 22:04:28,946 - reset.sh script executed.
2024-11-09 22:04:28,947 - Waiting 30 seconds for next iteration
2024-11-09 22:04:58,977 - Starting migration iteration 5
2024-11-09 22:04:58,977 - lab2_cold.sh script started.
2024-11-09 22:04:59,416 - Socket is DOWN
2024-11-09 22:14:17,818 - Socket was DOWN for 558402.5275 milliseconds
2024-11-09 22:14:25,996 - reset.sh script executed.
2024-11-09 22:14:25,997 - Waiting 30 seconds for next iteration
2024-11-09 22:14:56,025 - Monitoring completed after all cycles.
2024-11-09 22:14:56,025 - Average downtime: 561017.6099 milliseconds
2024-11-09 22:14:56,025 - Median downtime: 560824.2749 milliseconds
2024-11-09 22:14:56,026 - Highest downtime: 563810.7566 milliseconds
2024-11-09 22:14:56,026 - Lowest downtime: 558402.5275 milliseconds
(END)
```

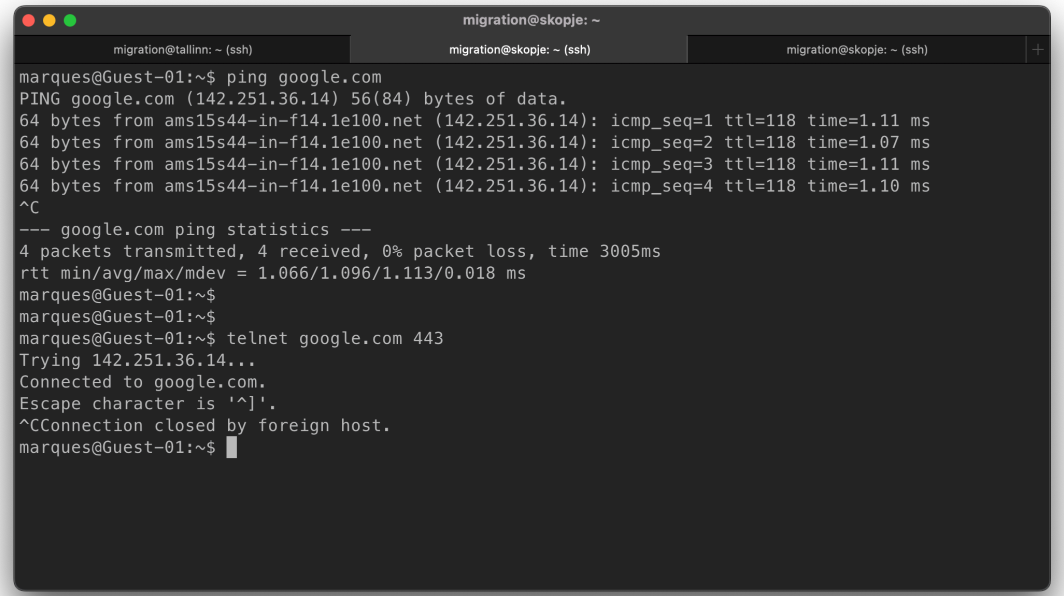
The downtimes for the cold migration were the following:

1. 563.8108 seconds
2. 563.2184 seconds
3. 560.8243 seconds
4. 558.8321 seconds
5. 558.4025 seconds

The time is most affected by the `qemu-img convert` and the `virsh vol-upload` commands.

- Mean: 561.0176 seconds
- Median: 560.8243 seconds

After migration, the new VM is functional and able to connect to network resources


migration@skopje: ~
migration@tallinn: ~ (ssh)
migration@skopje: ~ (ssh)
migration@skopje: ~ (ssh)

```
marques@Guest-01:~$ ping google.com
PING google.com (142.251.36.14) 56(84) bytes of data.
64 bytes from ams15s44-in-f14.1e100.net (142.251.36.14): icmp_seq=1 ttl=118 time=1.11 ms
64 bytes from ams15s44-in-f14.1e100.net (142.251.36.14): icmp_seq=2 ttl=118 time=1.07 ms
64 bytes from ams15s44-in-f14.1e100.net (142.251.36.14): icmp_seq=3 ttl=118 time=1.11 ms
64 bytes from ams15s44-in-f14.1e100.net (142.251.36.14): icmp_seq=4 ttl=118 time=1.10 ms
^C
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.066/1.096/1.113/0.018 ms
marques@Guest-01:~$
marques@Guest-01:~$
marques@Guest-01:~$ telnet google.com 443
Trying 142.251.36.14...
Connected to google.com.
Escape character is '^]'.
^CConnection closed by foreign host.
marques@Guest-01:~$
```

Task 7

Perform live migrations including copying the storage as part of the live migration with your partner, and measure the downtime. Compute mean and median of your chosen metric

Prerequisites

Needed to open ports:

```
sudo ufw allow from 145.100.96.0/20 to any port 16509 proto tcp
```

The script lab2_live.sh, created on tallinn, automates the process of live migration for the virtual machine from tallinn to skopje. This script initiates a live migration, where the VM continues to run during the migration process, minimizing downtime by transferring the VM’s memory, storage, and state while it remains operational.

```
virsh migrate --live Guest-01 qemu+ssh://migration@skopje/system
```

Method for measuring

We used a Python script named ‘lab2_live.py’ to measure the downtime during live migration with a threshold of at least 100 milliseconds and accuracy of 50 milliseconds. This script continuously checks the VM’s availability by attempting to connect to its SSH port, allowing us to detect when the VM becomes temporarily inaccessible.

```python
import socket
import time

# Target host, port, and settings
HOST = '145.100.106.194'  # IP address to monitor
PORT = 22  # Port to monitor
TIMEOUT_MS = 100  # Connection timeout in milliseconds
CHECK_INTERVAL_MS = 50  # Interval for checking the socket status
```

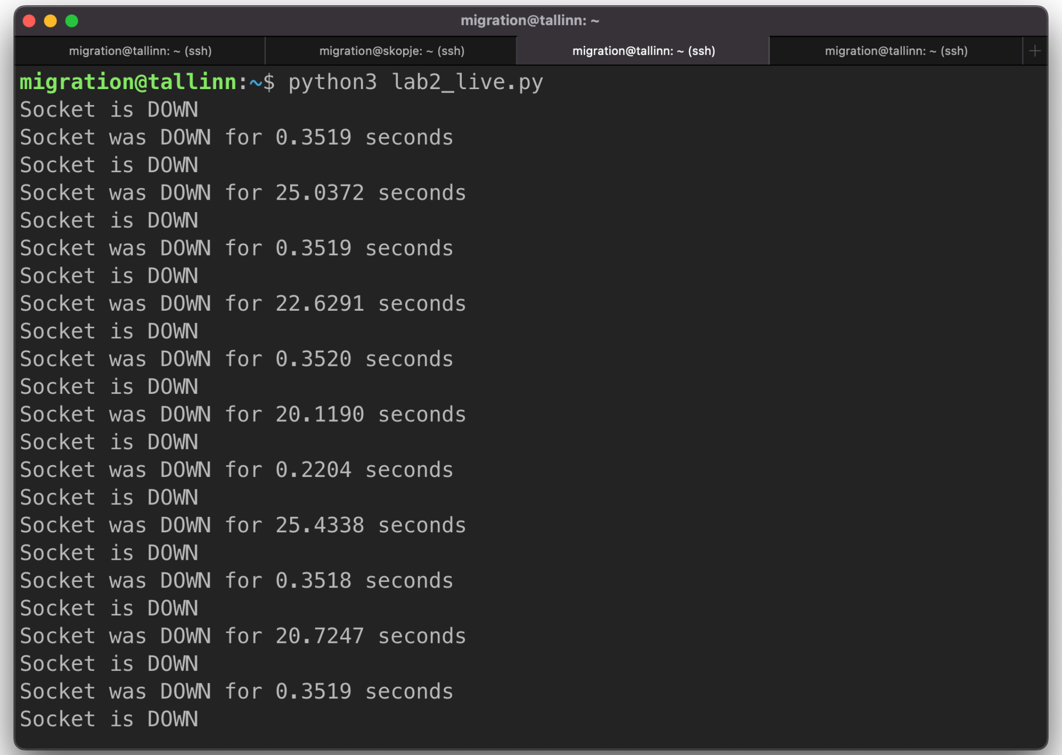
```python
def is_socket_up(host, port, timeout_ms):
    """
    Check if a socket at (host, port) is reachable within timeout_ms.
    Returns True if up, False if down.
    """
    timeout_sec = timeout_ms / 1000.0
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
         sock:
            sock.settimeout(timeout_sec)
            sock.connect((host, port))
        return True  # Connection succeeded
    except (socket.timeout, socket.error):
        return False  # Connection failed


# Monitor the socket status and log downtime
is_down = False
down_start_time = None

# run while not interrupted
while True:
    # if the socket is down
    if not is_socket_up(HOST, PORT, TIMEOUT_MS):
        # if the socket was not down before
        if not is_down:
            # Start tracking downtime with high precision
            is_down = True
            down_start_time = time.perf_counter()
            print("Socket is DOWN")

    elif is_down:
        # Calculate and log downtime duration with high precision
        down_duration = time.perf_counter() - down_start_time
        print(f"Socket was DOWN for {down_duration:.4f} seconds")
        is_down = False  # Reset the down flag
        down_start_time = None  # Reset the timer

    # Delay before the next check
    time.sleep(CHECK_INTERVAL_MS / 1000)
```

To run it:

```
python3 lab2_live.py
```

The measurements were coming from the python script’s output:

Downtimes for live migration:


migration@tallinn: ~
migration@tallinn: ~ (ssh)
migration@skopje: ~ (ssh)
migration@tallinn: ~ (ssh)
migration@tallinn: ~ (ssh)

```
migration@tallinn:~$ python3 lab2_live.py
Socket is DOWN
Socket was DOWN for 0.3519 seconds
Socket is DOWN
Socket was DOWN for 25.0372 seconds
Socket is DOWN
Socket was DOWN for 0.3519 seconds
Socket is DOWN
Socket was DOWN for 22.6291 seconds
Socket is DOWN
Socket was DOWN for 0.3520 seconds
Socket is DOWN
Socket was DOWN for 20.1190 seconds
Socket is DOWN
Socket was DOWN for 0.2204 seconds
Socket is DOWN
Socket was DOWN for 25.4338 seconds
Socket is DOWN
Socket was DOWN for 0.3518 seconds
Socket is DOWN
Socket was DOWN for 20.7247 seconds
Socket is DOWN
Socket was DOWN for 0.3519 seconds
Socket is DOWN
```

The longer downtimes are when we were running `reset.sh`.

1. 0.3519 seconds
2. 0.3519 seconds
3. 0.3520 seconds
4. 0.2204 seconds
5. 0.3518 seconds

- Average: 0.3256 seconds
- Median: 0.3519 seconds

Task 8

What would be the difference(s) when doing a live migration with shared storage?

Using shared storage for live migration significantly optimizes the process, leading to faster migrations, reduced downtime, and better use of network resources. This setup is ideal for environments that prioritize high availability and minimal disruption, making it a preferred method for VM migrations in enterprise and cloud environments.

Task 9

Optional: Perform live migrations using shared storage with your partner, and measure the down- time. Compute mean and median of your chosen metric. Hint: set up NFS, if you struggle with this contact the labteachers.

Install the package

```
sudo apt install nfs-kernel-server
```

Create the Shared Directory

```
sudo mkdir -p /mnt/shared_storage
```

Set Permissions for the directory

```
sudo chown -R nobody:nogroup /mnt/shared_storage
sudo chmod 777 /mnt/shared_storage
```

Configure the NFS Exports to specify the client allowed to connect.

```
sudo nano /etc/exports
```

Added the line **/mnt/shared_storage 145.100.104.36(rw,sync,no_root_squash,no_subtree_check)**

Export the Shared Directory

```
sudo exportfs -a
sudo systemctl restart nfs-kernel-server
```

The client was able to create the files

```
root@tallinn:/home/dmarques# ls /mnt/shared_storage/
images  testfile  test.yaml
```