Fault Recovery using Checkpointing

Mihnea-Cristian Matei
SCPD
"POLITEHNICA" University of Bucharest

Click for github
(https://github.com/MihneaMatei/Fault-Recovery-using-Checkpointing)

Contents

1. Description

In today's digital landscape, the operation without interruption of complex distributed systems is mandatory, as the repercussions generated by system failures often lead to significant financial losses or compromise of critical services. The development of efficient and effective fault tolerance mechanisms is essential, as these mechanisms are meant to reduce the risks of failurile and to ensure as little interruptions as possible.

This project, "Fault Recovery using Checkpointing", addresses this task by implementing a fault tolerance mechanism based on checkpointing, a technique which involves periodically saving the state of a running system. This approach aims to reduce downtime and to enhance the reliability of the system.

Fault recovery is the process of restoring a system to a precedent, functional state after a failure has occurred. The failures can occur from various sources such as software errors, hardware faults or network issues. Based on the systems' complexity and size, fault recovery can become a very challenging task and a fault recovery mechanism that can scale is very difficult to achieve. The fault recovery mechanisms are designed to detect, troubleshoot and reduce these faults, allowing the system to continue to operate.

Checkpointing is the operation of periodically saving the state of a system (or process) that is running and is functional. The checkpoints are saved on a stable storage environment and, in case of failure, they are used to restore the system (or process) to a previous, stable state, minimizing loss of work and reducing the system's downtime.

Fault recovery using Checkpointing is a project focused primarily on providing system reliability by implementing a mechanism capable of fault-tolerance. Failures are inevitable in complex, distributed systems due to various reasons. The algorithm will seek a balance between checkpoints frequency, which can affect performance by creating too much overhead, and granularity, which can affect recovery time by having large checkpoints. One approach that can mitigate both these issues is incremental checkpointing, which implies saving only the changes done to the system since the previous checkpoint.

In order to achieve that, the project will most likely involve distributed systems frameworks such as Apache Hadoop or Apache Spark, because they provide a great infrastructure for implementing fault recovery mechanisms via their capabilities for

fault tolerance and scalability in distributed environments. The programming language most suited for this task is Python, due to its extensive capabilities for fast, complex development, especially effective in distributed environments. Python's high-level capabilities offer rapid development, thus accelerating this process.
In summary, this project seeks to address a critical aspect in distributed systems, fault tolerance, by using efficient and effective checkpointing algorithms in order to enable fault recovery. Through the integration of incremental checkpointing for fault recovery, the project aims to enhance system reliability and minimize the impact of failure on the system operation.

2. State of the Art

2.1. Existing Solutions

In recent years, due to the larger demand on distributed systems, fault tolerance and recovery mechanisms have seen a significant increase in attention from researchers and developers.
One notable algorithms is the Chandy-Lamport algorithm, proposed in 1985, that offers an approach based on coordinated checkpointing protocols that aim to achieve global consistency by coordinating the checkpointing processes across all nodes in the distributed system, offering a great theoretical foundation. This algorithm can provide a great method of checkpointing, should the system fail, but can also serve as a way to collect garbage, by removing unreferenced objects and can be a great method to detect deadlock, among other debugging. The paper implies that processes in a distributed system can determine the global state of the system during a computation. These processes can communicate between them by sending and receiving messages and a process can record the messages that passed through it and also its own state. By having all the processes record their own state and messages, at their own time, an overall snapshot can be created by piecing together all the snapshots from each process.[1] The algorithm works as it follows: initially, the snapshot process begins when a designated node (initiator) initiates a global snapshot request by marking its local state and sending markers through the communication channels. Each node then receives the marker and records its local state and forwards the marker again. The process of marker forwarding happens recursively until all nodes receive the marker. As the markers propagate, each node records its local state, allowing them to capture the state at a consistent point in time,

ensuring snapshot consistency. Once the propagation is over, a global snapshot is built by combining all the local snapshots from all the nodes, thus providing a consistent view of the system at a point in time. We must note, however, that snapshots will not be taken at the exact same instant, due to synchronization problems. The Chandy-Lamport algorithm ensures correctness and consistency of the snapshot by guaranteeing that all nodes and messages are included in the snapshot at the cost of message overhead, storage requirements and synchronization complexity.

In a paper published in 1998, researchers Om. Damani and V. Garg proposed a solution that revolves around the integration of a rollback-based optimistic fault-tolerance scheme with an optimistic distributed simulation scheme that aims to minimize the wasted computation in the event of failure. When a logical process fails, the computation in which it takes place comes to a halt. The simple restart of the process may harm the computation, as it may enter an inconsistent state. The solution proposes the creation of periodical checkpoints that are saved on stable storage by each logical process involved in a computation. The checkpoints are saved asynchronously while the computation is in progress. After a system failure, or a crash, the saved checkpoints are used to restart the computation, therefore minimizing the wasted computation.[2] In the proposed solution, failure is considered a straggler event (an event that takes very long to complete) with the receive time equal to the virtual time of the last checkpoint saved on the stable storage. The inconsistent states created by the failure of logical processes will be treated as orphan states. Similarily, messages that happened during a faulty logical process will be treated as orphan messages. To remove those inconsistent states and messages, a technique called transitive dependency tracking will be used, ensuring the rollback of all orphan states and orphan messages. By removing those elements, a decrease in overhead of fault recovery will be achieved and correctness of system will be ensured. To minimize storage access time and to further reduce overhead, Om. Damani and V. Garg proposed the grouping of logical processes into clusters, saving checkpoints for the entire cluster rather than for each individual logical process. This solution offers great efficiency by reducing the overhead of rollback by only rolling back the states dependent on a faulty logical process. At the same time, this method assures the correctness of the systems while providing a great possibility of scaling. These advantages come at the price of complexity added to

the system, synchronization challenges that may appear in tracking dependencies across multiple processes and increased resource consumption.

The paper *Incremental Checkpointing for Fault-Tolerant Stream Processing Systems: A Data Structure Approach* was published in 2020 by C. Lin, L. Wang and S. Chang. The motivation of the research is to address the challenges of in-memory databases in maintaining high throughput and data persistence simultaneously. The researchers discuss the limitations of existing methods for logging, such as command and ARIES logging and the highlight the need for a checkpointing mechanism that combine the functionalities of both methods, but reduce the overhead and storage costs faced by those methods. The proposed solution is to leverage a data structure based incremental checkpoint (DSIC) methods that takes advantage of the irrelevance of order in data structures, in particular sets, and uses incremental checkpointing concepts to create an efficient checkpointing mechanism. This solution aims to reduce logging time and storage costs while mainting ARIES and command logging features, such as write-ahead logging (writing logs to storage before transactions to database) or transaction logging (records of all changes made to database for recovery purposes). The components of DSIC (Data Structure-based Incremental Checkpointing) would be the metadata file, containing a mapping between original key and extended key, logs for checkpoints and a base set of initial members (collections of data) in the database. The paper describes two important schemes: DSIC-Undo scheme which implies that all operations in the add set and remove set are recorded at each operation and the DSIC-Redo scheme which implies that the base set remains unchanged. The method works as it follows: the DSIC maintains checkpoint logs of add and remove sets in order to track the changes at database level. Based on the performed operations, the base set will be updated, allowing for quick recovery to previous states, operation done by the DSIC-Undo scheme, while the base set will remain unchanged, based on the DSIC-Redo scheme. Checkpoints and rollback procedures will ensure data consistency and fault-tolerance by enabling recovery to previous states. The main advantages of this method are the reduced logging time and the reduced storage cost, other advantages being the adaptability of the method to both read intensive and write intensive applications and a rapid and efficient recovery. However, these advantages come at the cost of complexity added to the system and an increased storage overhead. Performance trade-offs can also become a major issue based on application workload characteristics.[3]

In 2021, a paper was published by a group of Japanese researchers. The purpose was to propose a solution that solves the problems associated with taking partial snapshots in distributed systems (see K. M. Chandy & L. Lamport *Distributed Snapshots: Determining Global States of Distributed Systems*). The algorithm is called Cooperative Partial Snapshot (CPS) and it works in a distributed system where nodes communicate via message passing, asynchronously. Communications links must be reliable and nodes must be allowed to join and leave freely. The nodes will take partial snapshots of the subsystem responsible with communication. Any node can initiate the algorithm by sending the marker message to its communication-related nodes. In case two or more nodes initiate the algorithm concurrently, CPS creates a virtual link between the initiators, forming an initiator network. By using an initiator network, the nodes can coordinate without the need of a leader, reducing communication overhead and solving the problem of collision or overlapping. The algorithm works in two phases. In phase 1, when initiators determine their snapshot groups and virtual links are created in order to avoid collisions. In phase 2, a spanning tree is constructed and termination signals are broadcasted to ensure algorithm completion. The CPS algorithm features a rollback mechanism that allows groups to restore their state in case of failure.[4] The main advantages of the CPS algorithm are the reduced communication overhead, by creating virtual links between collided initiators, efficient collision handling, high scalability and improved time complexity, thanks to its capability to handle multiple collisions concurrently. The downsides of this algorithm is its added complexity to the system, the dependency on the system model and its added complexity to communication.

2.2. Comparison and Conclusion

The Chandy-Lamport algorithm offers a solid foundation to global consistency in distributed systems and provides a great method for coordinated checkpointing. It can be utilized as a garbage collector or as a deadlock detector and guarantees the correctness and consistency of snapshots for fault recovery. However, it incurs message overhead and has high requirements for storage. Synchronization issues can occur and adds a high amount of complexity.

Damani and Garg solution integrates a rollback-based optimistic fault tolerance with distributed simulation, minimizing wasted computation in case of failure. It uses transitive dependency to track and handle orphan states and messages, reducing recovery overhead and minimizes access time and overhead by grouping logical

processes into clusters. It offers good scalability and ensures correctness for fault recovery. However, it adds a high amount of complexity and synchronization challenges, similar to Chandy-Lamport algorithm and increases resource consumption.

The DSIC approach utilizes data structures in order to improve incremental checkpointing. Doing so, logging time and storage costs are reduced. It is greatly adapted to database application, offering good performances for both read intensive and write intensive applications. As to both previous presented solutions, a high amount of complexity is added to the system. Performance trade-offs may appear due to application workload characteristics and storage overhead is increased.

The CPS algorithm reduces communication overhead by creating virtual links between collided initiators and handles the collisions efficiently. The algorithm improves time complexity by being capable to handle multiple collisions and scales well. Disadvantages include added complexity to the system, dependency on the system model and increased communication complexity.

Each approach offers an impressive set of advantages at the cost of some disadvantages. The choice among them depdens on the requirements of the systems, such as scalability, performance or complexity, as well as other constraints or characteristics.

3. Implementation Details

The fault-recovery system is designed to provide robust and automatic handling of service failure within a distributed virtual machine (VM) environment. The system utilizes uncoordinated checkpointing to capture and store the state of a number of services defined by the user at regular intervals. This approach ensures that we can quickly restore the services to the last known working state in the event of a failure or misconfiguration. The setup runs on 2 VMs, the first one, VM1, being the leader which runs the fault recovery script and the second one is a backup in case of VM1 failure. To be noted that the services are checked and checkpointed on both VMs. Alongside the checkpoint and recovery mechanism, on VM1 runs a lightweight HTTP server built using Flask framework, which server as both health check and monitor endpoint. This server allows VM2 to continuously monitor the status of VM1 (both the working state of the VM and the working state of the script running in it) by sending HTTP requests and evaluating the responses. In case of VM1

failure, VM2 is configured to automatically take over its duties, starting the fault recovery script and utilizing the most recent checkpoints to restore the services. The failover process is automatic and upon VM1 restoration, the leadership will be given back to VM1 and VM2 will return to backup state, stopping the script for fault recovery. To be noted that there are two services watched, nginx and httpd. The script checks both their working state and configuration files. To test the setup more easily, a fault injection script was created that randomly stops the services or corrupts the configuration files.

3.1. The checkpoint and recovery script – script.py

The setup.py script is the main script of this setup and is the python code where the checkpointing and recovery in case of failure happens. It utilizez paramiko library to manage SSH connections and perform operatios on the remote machines. It includes several custom functions that interact with the services and files, creating a basic automated system to monitor, checkpoint and restore services as needed.
There are 4 libraries used: paramiko (for SSH connections), time (for added delays), os (for path manipulations) and posixpath (to handle Unix paths). This script file must be present on all the VMs in the setup (in this case, the two of them).

The “establish_ssh_connection(hostname, username, password)” function has the purpose of establishing an SSH connection to the specified host by creating a new SSH client, setting a policy to add the host key to the known hosts, then connects to the specified host using the given credentials and returns the connected SSH client object.

The “execute_ssh_connection(ssh_client, command)” function executes a command via SSH on the specified client. It sends a command to the SSH client, reads and decodes the output of the command and then returns both the output and the error.

The “check_httpd_status(ssh_client)” function sends a executes the status of the httpd service on a remote machine by executing systemctl status httpd command. If the output contains that the service is active, the function returns true.

The “create_checkpoint(ssh_client, files, service_name, checkpoint_dir)” function creates the checkpoint by constructing a path for the service within the checkpoint

directory, copies the configuration files into this directory and announces completion of the process.

The “restore_files(ssh_client, files, service_name, checkpoint_dir)” function restores the files from the checkpoint directory, overwriting each target file with its corresponding version in the checkpoint directory.

The “file_has_changed(ssh_client, file_path, checkpoint_file_path)” function checks if a configuration file has changed since the last checkpoint by comparing it with the current version of it.

The “check_and_restore_service(ssh_client, service_name, files, checkpoint_dir)” checks if a service is active and if the configuration files have changed since the last checkpoint. If the files have changed, it attempts to restart the service to check if the configuration is compatible with the service. If the service cannot restart, it restores the files and tries again.

The script body continuously monitors services on remote machines, creates checkpoints and restores services if needed. The list of monitored VMs is taken from a text file, vms.txt and the list of services is taken from a text file, services.txt.

```
192.168.0.94 VM1 root osboxes.org
192.168.0.143 VM2 root osboxes.org
```

Fig. 1. Sample of vms.txt

```
httpd /etc/httpd/conf/httpd.conf /usr/lib/systemd/system/httpd.service
nginx /etc/nginx/nginx.conf /usr/lib/systemd/system/nginx.service
```

Fig. 2. Sample of services.txt

```python
def check_and_restore_service(ssh_client, service_name, files, checkpoint_dir):
    output, _ = execute_ssh_command(ssh_client, f"systemctl is-active {service_name}")
    if output.strip() != "active":
        print(f"{service_name} is down. Checking for file modifications...")
        service_changed = False

        for file in files:
            checkpoint_file_path = posixpath.join(checkpoint_dir, service_name, posixpath.basename(file))

            if file_has_changed(ssh_client, file, checkpoint_file_path):
                print(f"{file} has been modified.")
                service_changed = True

        if service_changed:
            print("Changes detected, attempting to restart the service...")
            _, error = execute_ssh_command(ssh_client, f"systemctl restart {service_name}")
            if error:
                print("Restart failed, restoring from checkpoint...")
                restore_files(ssh_client, files, service_name, checkpoint_dir)
                execute_ssh_command(ssh_client, f"systemctl daemon-reload")
                execute_ssh_command(ssh_client, f"systemctl restart {service_name}")
            else:
                print(f"{service_name} restarted successfully.")
        else:
            print("No changes detected, restoring files from checkpoint...")
            restore_files(ssh_client, files, service_name, checkpoint_dir)
            execute_ssh_command(ssh_client, f"systemctl restart {service_name}")
    else:
        service_changed = False

        for file in files:
            checkpoint_file_path = posixpath.join(checkpoint_dir, service_name, posixpath.basename(file))
            if file_has_changed(ssh_client, file, checkpoint_file_path):
                print(f"{file} has been modified.")
                service_changed = True
        if service_changed:
            print("Changes detected, attempting to restart the service...")
            _, error = execute_ssh_command(ssh_client, f"systemctl restart {service_name}")
            if error:
                print("Restart failed, restoring from checkpoint...")
                restore_files(ssh_client, files, service_name, checkpoint_dir)
                execute_ssh_command(ssh_client, f"systemctl restart {service_name}")
            else:
                print(f"{service_name} restarted successfully.")
```

Fig. 3. Section of code from script.py file; the check_and_restore_service function

3.2. The server script – server.py

The server script sets up a simple web server using Flask, a popular Python web framework. The server provides a single route to check if a specific Python script (script.py) is currently running on the host machine. This script must be executed in the leader VM (in this case, VM1). It utilizes 3 libraries: Flask, subprocess and os.

The function “is_recovery_script_running()” checks if the script is running on the current host by running a pgrep command that searches for processes that include the script.py file. The function returns True or False based on whether the script is running or not.

The Flask route “@app.route(“/”)” is the route to the root endpoint of the web server and its “index()” function calls the script checking function presented aboce to see

whether the script is running or not. If the script is running, it returns code 200, if not, error code 503 is thrown.

```python
def is_recovery_script_running():
    try:
        output = subprocess.check_output(["pgrep", "-f", "script.py"])
        return output.decode().strip() != ""
    except subprocess.CalledProcessError:
        return False

@app.route("/")
def index():
    if is_recovery_script_running():
        return "Recovery script is running.", 200
    else:
        return "Recovery script is NOT running.", 503
```

Fig. 4. Section of code from server.py

3.3. The monitoring script – monitoring.py

The monitoring script is designed to monitor the status of a remote server by regularly checking a specific URL, like a heartbeat, and if needed, to run the fault recovery script on the local machine. This script should bbe started on the backup machine, VM2. The script uses 3 libraries: requests, time and subprocess.

The function “check_server(url)” is the main function here and checks if a server is active and if a specific service (the recovery script) is running on it by making a HTTP request to a provided URL.

The function “run_recovery_script()” runs a local python script, the fault recovery script, as a separate process, when the main server is not functioning correctly.

```python
def check_server(url):
    try:
        response = requests.get(url)
        if response.status_code == 200:
            print("VM1 is active, and the recovery script is running.")
            return True
        else:
            print("VM1 is active, but the recovery script is NOT running.")
            return False
    except requests.RequestException:
        print("VM1 is down.")
        return False

def run_recovery_script():
    print("Running the recovery script...")
    process = subprocess.Popen(['python', 'script.py'], start_new_session=True)
    return process
```

Fig. 5. Section of code from monitoring.py

3.4. The fault injection script – fault_injection.py

The fault injection script provides a mechanism to automatically inject faults into a set of services running on the VMs, simulating potential real-world system failures. This script can be started on any machine. It utilizes libraries like paramiko, random and time.

The "establish_ssh_connection(hostname, username, password)" and "execute_ssh_command(ssh_client, command)" are similar to the ones in the script.py description.

The "inject_fault(ssh_client, service_name, files)" function randomly injects faults into services or their configuration files to simulate failure scenarios. It randomly chooses a type of fault to inject: either stopping the service or corrupting its configuration files.

The script reads the list of VMs and the list of services from the same vms.txt and services.txt files described earlier.

```python
def inject_fault(ssh_client, service_name, files):
    fault_type = random.choice(['stop_service', 'corrupt_file'])

    if fault_type == 'stop_service':
        print(f"Stopping {service_name}...")
        execute_ssh_command(ssh_client, f"systemctl stop {service_name}")
        print(f"{service_name} stopped for fault injection.")
    elif fault_type == 'corrupt_file':
        # Randomly select a file to corrupt
        file_to_corrupt = random.choice(files)
        print(f"Corrupting file {file_to_corrupt}...")
        execute_ssh_command(ssh_client, f"echo 'Corrupted content!' > {file_to_corrupt}")
        print(f"Injected corruption into {file_to_corrupt}.")
```

Fig. 6. Section of code from fault_injection.py

4. Performance Analysis

4.1. Tested Scenarios

In the tested fault scenarios, the focus was on simulating service crashes and configuration corruption to assess the resilience of the recovery mechanisms for the system. Service crash scenarios involved intentionally stopping the nginx and httpd services to observe how the system detects and responds to their failure, aiming to ensure minimal downtime. For configuration corruption, errors were deliberately introduced into configuration files to evaluate the system's ability to detect and revert these changes back to their last known good state.

```
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
192.168.0.143 - - [30/May/2024 07:51:42] "GET / HTTP/1.1" 200 -
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
nginx is down. Checking for file modifications...
No changes detected, restoring files from checkpoint...
Restoring checkpoint...
Checkpoint restored and applied.
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
```

Fig. 7. The fault recovery script is running; an error was detected on nginx

```
VM1 is active, but the recovery script is NOT running.
Running on backup VM
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
VM1 is active, and the recovery script is running.
Resuming on VM1
```

Fig. 8. Switchback to leader after failover

4.2. Metrics

The most important metrics in a fault-recovery setup are time and size on disk. The fingerprint left by the checkpoints director for the two service, httpd and nginx, are very small, of just 55 bytes.

```
drwxr-xr-x.  4 root    root      55 May 29 17:54 checkpoints
```

Fig. 9. Checkpoints directory size on disk

Time-wise, when the cluster is healthy, a checkpoint creation time is between 250ms and 450ms, which is fine considering the small sizes of the files. The checkpoints are set to be created every 10 seconds, so the frequency is pretty high, generating a high amount of overhead.

The Recovery Time Objective (RTO) when there is no recovery to be made sits between 600ms and 900ms. Again, due to high frequency, there is a lot of generated overhead and a lot of communication happening internally.

```
Checkpoint created at: /tmp/checkpoints/httpd
Recovery Time Objective (RTO): 0.6274888515472412 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
Checkpoint creation time: 0.34679508209228516 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
Recovery Time Objective (RTO): 0.7406914234161377 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
Checkpoint creation time: 0.29553914070129395 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
Recovery Time Objective (RTO): 0.6730427742004395 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
Checkpoint creation time: 0.4246079921722412 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
Recovery Time Objective (RTO): 0.6426575183868408 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
Checkpoint creation time: 0.32279419898986816 seconds
Recovery Time Objective (RTO): 0.7565791606903076 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/httpd
Checkpoint creation time: 0.30874085426330566 seconds
Recovery Time Objective (RTO): 0.6369011402130127 seconds
Creating checkpoint...
Checkpoint created at: /tmp/checkpoints/nginx
Checkpoint creation time: 0.25743985176086426 seconds
Recovery Time Objective (RTO): 0.9014303684234619 seconds
```

Fig. 10. Checkpoint creation and no recovery times

When one of the services requires recovery, the recovery time increases to over a second, in this case, when the httpd service failed, the required time was 1.353 seconds to recover.

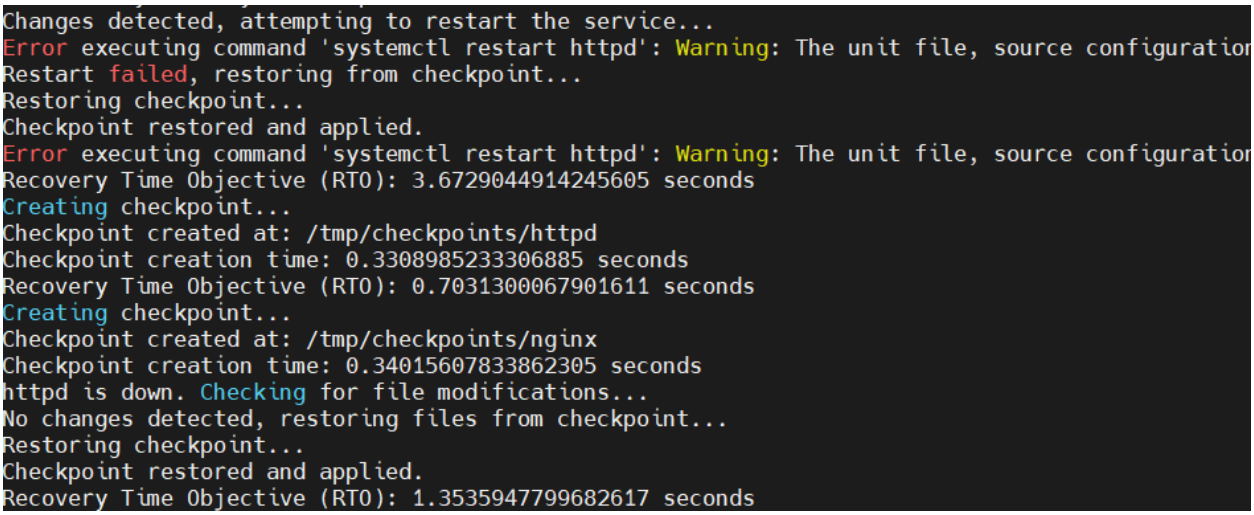

Fig. 11. Recovery time when one service requires recovery

Finally, there is the response time from the HTTP server to the backup VM, which is around 20-40ms. A low response time, due to a small body of response.

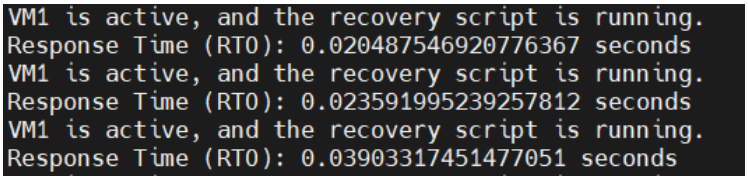

Fig. 12. HTTP server response time to backup VM

In terms of resource consumption, on the backup node, the CPU idle usage sits below 0.5% on average. When the script for monitoring is running, the average usage is around 10%, with a spike of 17.67% when the failover occurs.

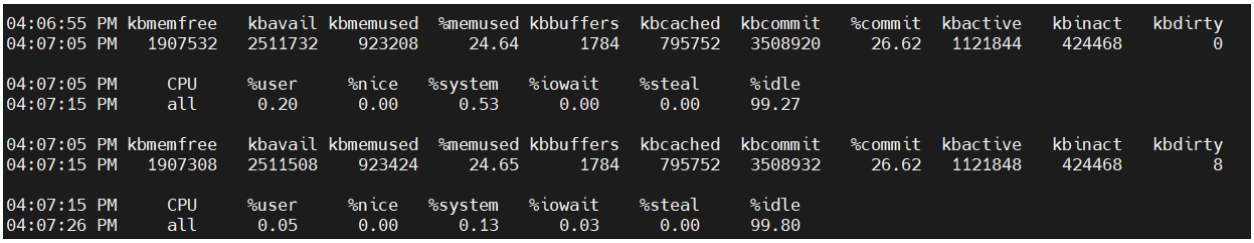

Fig. 13. CPU usage while idling on backup node

```
04:07:36 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:07:46 PM   1998056   2602260    832696     22.22      1784    795752   3409292     25.86   1051948    424472         4

04:07:46 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:07:56 PM     all      6.77      2.73      7.52      0.66      0.00     82.33

04:07:46 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:07:56 PM   1867972   2481456    950688     25.37      1784    804696   5010120     38.00   1185256    428772         4

04:07:56 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:06 PM     all      1.92      0.34      2.78      0.03      0.00     94.93

04:07:56 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:08:06 PM   1953856   2567364    864768     23.08      1784    804700   4873512     36.97   1088360    428788        16

04:08:06 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:16 PM     all      1.76      2.44      2.83      0.03      0.00     92.95

04:08:06 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:08:16 PM   1882308   2495876    936020     24.98      1784    804740   5010752     38.01   1185616    428816         8

04:08:16 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:26 PM     all      4.62      0.98      4.02      0.00      0.00     90.39
```

Fig. 14. CPU usage while the setup is running on backup node

On the leader node, the average idle usage is similar, but the running usage is a little bit different. When the scripts are running and are just conducting checks, the usage is less than 5% usage. Things change when a fail occurs, with spike hitting close to 20% usage (19.9%).

```
04:07:31 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:07:41 PM     all      0.05      0.00      0.53      0.81      0.00     98.61

04:07:31 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:07:41 PM   1968864   2583412    852136     22.74      1784    805708   3462716     26.26   1093624    427916        12

04:07:41 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:07:51 PM     all      5.81      2.19      7.75      4.15      0.00     80.10

04:07:41 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:07:51 PM   1851356   2472460    961808     25.67      1784    811596   3629396     27.53   1198080    430752         4

04:07:51 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:01 PM     all      2.00      0.05      2.23      1.43      0.00     94.30

04:07:51 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:08:01 PM   1891744   2512924    920928     24.58      1784    811660   3583096     27.18   1164528    430820        28

04:08:01 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:12 PM     all      2.97      2.77      3.43      0.09      0.00     90.74

04:08:01 PM kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
04:08:12 PM   1841948   2463192    970860     25.91      1784    811712   3632380     27.55   1201112    430868        16

04:08:12 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
04:08:22 PM     all      4.48      1.31      4.34      0.78      0.00     89.09
```

Fig. 15. CPU usage on leader node when a spike occurs

4.3. Results and Observations

In the evaluation of the fault recovery system using checkpointing across two virtual machines, a range of system behaviors under different operational conditions can be observer. Under idle conditions, both VMs exhibited exceptionally low resource usage, consistently below 1%, indicating minimal overhead when the system is not actively handling tasks. During normal operations with active scripts, the primary node (VM1) maintained resource usage below 5%, demonstrating the efficiency of the recovery and monitoring scripts under typical workload conditions. Notably,

during failure scenarios, resource usage on VM1 spiked to between 10-20%, reflecting the system's increased computational demand to manage error detection, logging, and attempts to restore system state. The secondary node (VM2), tasked primarily with monitoring, showed a higher baseline resource utilization at around 10%, likely due to the continuous polling and status checking processes. This usage spiked briefly to 16% during failover events, as VM2 engaged in more intensive tasks to assume control from VM1. These findings highlight the system's robust response capabilities and resilience, though they also suggest potential areas for optimization, particularly in reducing the monitoring load on the secondary node to enhance overall system efficiency.

5. Conclusions

The development and deployment of the fault recovery system using checkpointing across two virtual machines have demonstrated substantial resilience and efficiency in managing system failures, achieving minimal downtime and maintaining data integrity under various conditions. The primary node consistently exhibited low resource utilization, reflecting the minimal operational overhead of our mechanisms, while resource spikes during failures underscored the system's active engagement in robust recovery protocols. The secondary node effectively managed failover with great efficiency, though its higher baseline consumption suggests potential areas for optimization in monitoring. Challenges such as optimizing monitoring loads and ensuring data consistency were addressed, providing significant learning opportunities. Future developments should explore coordinated checkpointing to potentially reduce recovery times and expand scalability. This project serves as a foundational model for similar high-availability environments, offering scalable and adaptable solutions for critical system operations.

6. References

[1] K. M. Chandy, L. Lamport (1985), *Distributed Snapshots: Determining Global States of Distributed Systems*. Available: https://lamport.azurewebsites.net/pubs/chandy.pdf Accessed: April 2024

[2] Om. P. Damani, V. K. Garg (1998), *Fault-Tolerant Distributed Simulation*. Available: https://dl.acm.org/doi/pdf/10.1145/278008.278014 Accessed: April 2024

[3] C. Lin, L. Wang, S. Chang (2020), *Incremental Checkpointing for Fault-Tolerant Stream Processing Systems: A Data Structure Approach*. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9076291 Accessed: April 2024

[4] J. Nakamura, Y. Kim, Y. Katayama, T. Masuzawa (2021), *A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems and experimental evaluations*. Available: https://arxiv.org/pdf/2103.15285.pdf Accessed: April 2024