Politehnica University of Bucharest

RabbitMQ: Phase 4 – Final Report

Name: Alexandru – Florin Ionescu

Date: 22/05/2025

Table of Contents

1. Platform Overview

RabbitMQ is an AMQP-based message broker that provides reliable, clustered messaging. In our setup we run a three-node cluster on Ubuntu VMs connected via a VirtualBox bridged network:

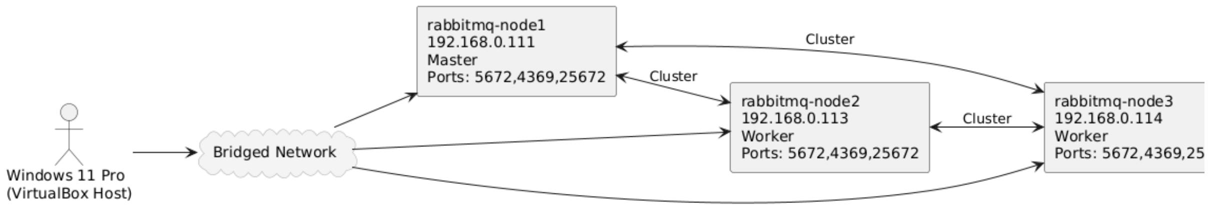

Figure 1: VM & Network Topology

We enabled:

- **Clustering:** all nodes share the same Erlang cookie and form a cluster.

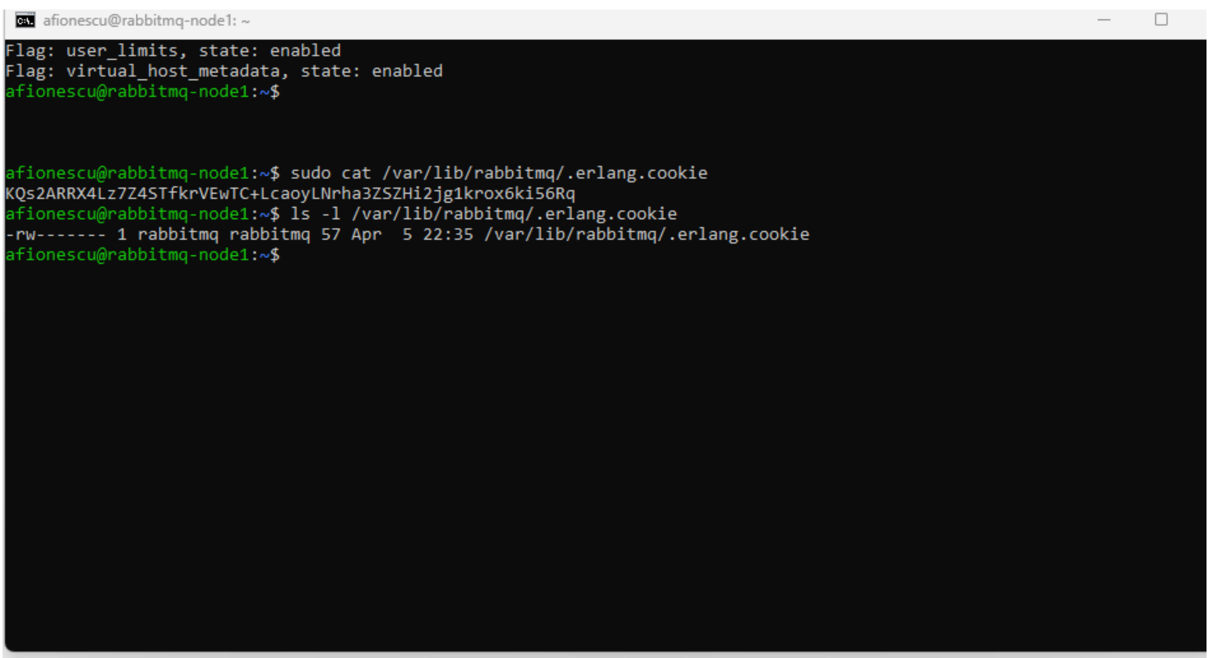

Figure 2: Contents and permissions of `/var/lib/rabbitmq/.erlang.cookie` on **rabbitmq-node1** (identical on all nodes)

- **High-Availability Queues:** x-ha-policy = all mirrors every queue on all three nodes: *sudo rabbitmqctl set_policy ha-all "^" '{"ha-mode":"all","ha-sync-mode":"automatic"}'*

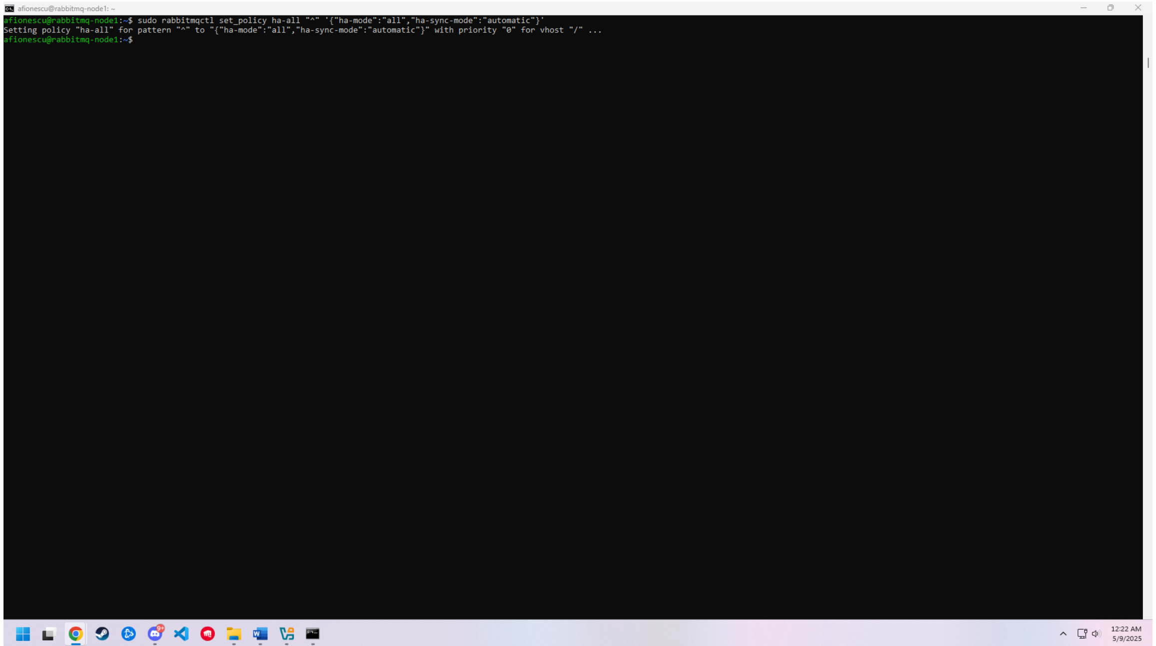

Figure 3: HA policy applied so that all queues are mirrored

- **Message Persistence:** delivery_mode=2 ensures messages are written to disk.

These features give us fault tolerance (no single point of failure) and durability (no lost messages on broker restarts).

2. Solution Design and Implementation

All source code, helper scripts, and configuration files are available in the public GitHub repository GitHub Link . Runtime log files are **not** stored in-repo; they can be recreated in a few seconds by rerunning the benchmark commands included in the README.

2.1. Logical Application Architecture

We built a simple producer–consumer app. A single **send.py** script running on node1 publishes messages to a durable queue named test-queue. Two **receive.py** consumers (on node2 and node3) pull from that queue and acknowledge each message.

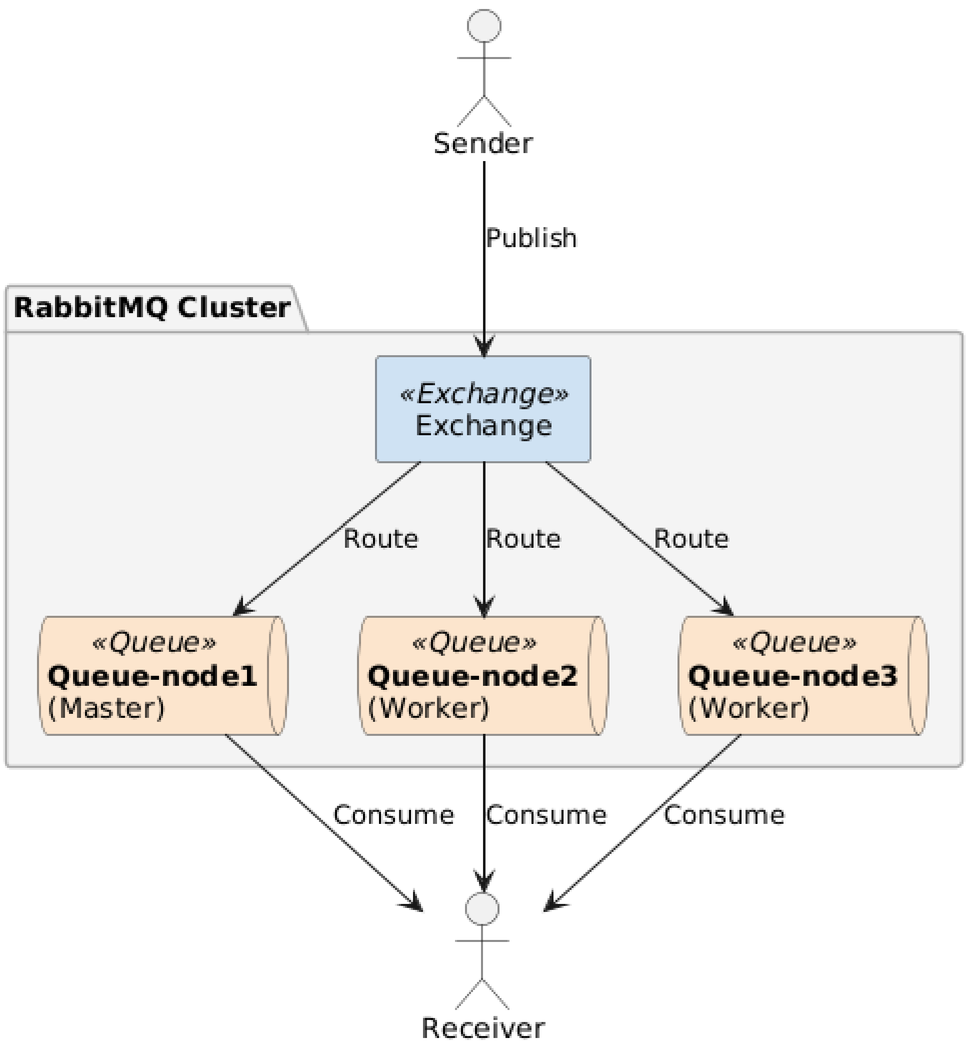

Figure 4: Logical Application Architecture

2.2. Physical Deployment & Scripts

Sender Script (send.py).

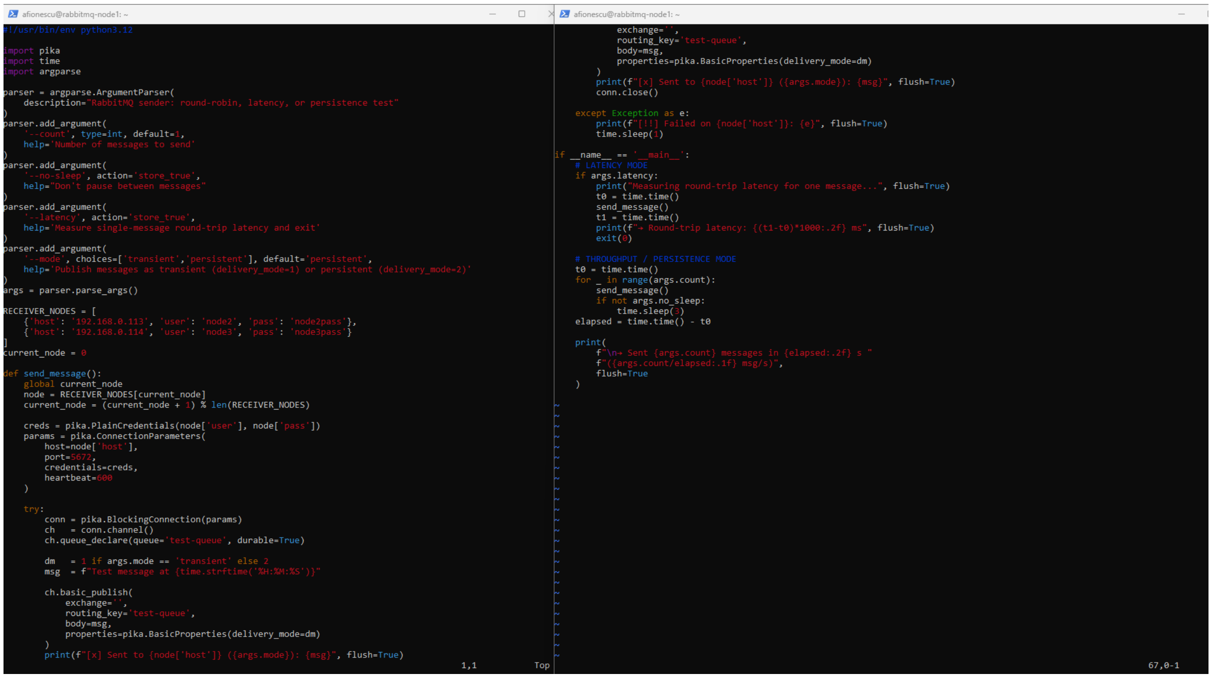

Figure 5: send.py script

- Maintains a Python list **RECEIVER_NODES** with node IPs and credentials.
- In **send_message(body):**

Builds a **ConnectionParameters** with host, **PlainCredentials**, and **heartbeat=600**.

Opens a new connection and channel.

Declares **test-queue** as durable.

Publishes **body** with **delivery_mode=2**.

Closes the connection.

Main loop calls **send_message()** then **time.sleep(3).**

(the pause is bypassed during throughput tests with the --no-sleep flag)

#send.py (snippet)

RECEIVER_NODES = [

```
    {'host':'192.168.0.113','user':'node2','pass':'node2pass'},
    {'host':'192.168.0.114','user':'node3','pass':'node3pass'}
]
current = 0


def send_message(body):
    node = RECEIVER_NODES[current % len(RECEIVER_NODES)]
    params = pika.ConnectionParameters(
        host=node['host'],
        credentials=pika.PlainCredentials(node['user'], node['pass']),
        heartbeat=600
    )
    conn = pika.BlockingConnection(params)
    ch = conn.channel()
    ch.queue_declare(queue='test-queue', durable=True)
    ch.basic_publish(
        exchange='',
        routing_key='test-queue',
        body=body,
        properties=pika.BasicProperties(delivery_mode=2)
    )
    conn.close()
```

Receiver Scripts (receive.py)

On the consumer side, I bind to the same exchange and consume messages asynchronously:

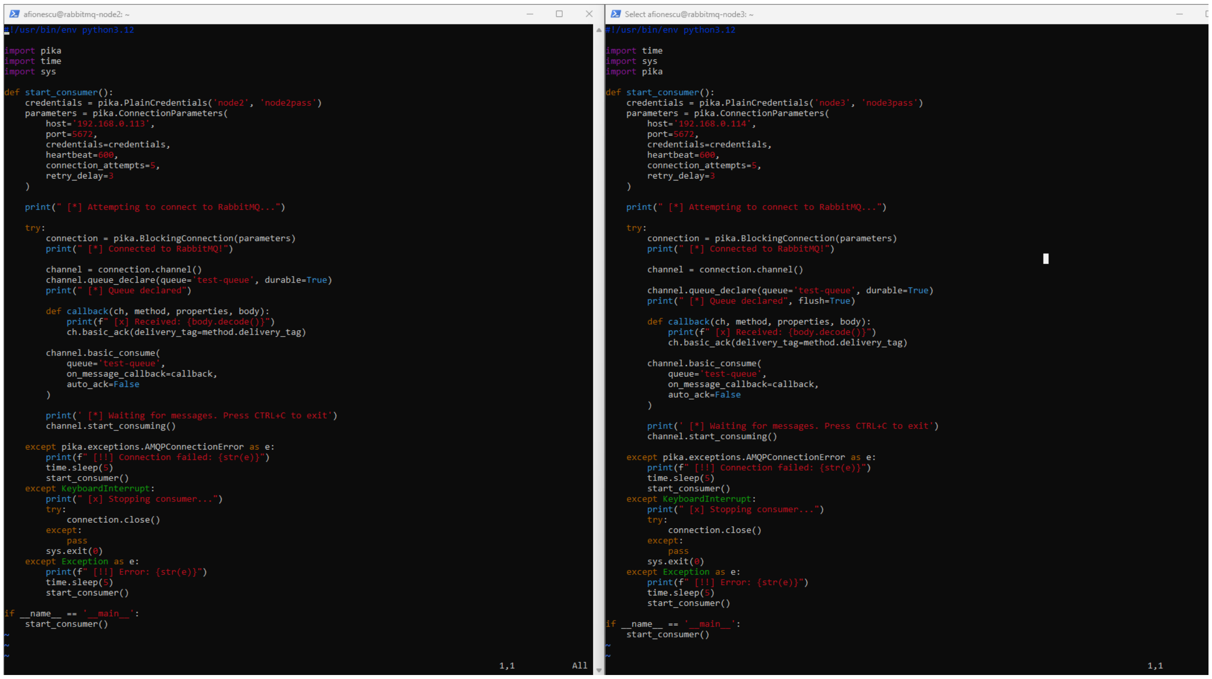

Figure 6: receive.py script

- Connects with **PlainCredentials** and **heartbeat=600**.
- Declares the same durable **test-queue**.
- Defines **callback(...)** to print **body.decode()** and call **ch.basic_ack()**.
- Uses **basic_consume(..., auto_ack=False)** and start_consuming().
- On **AMQPConnectionError**, waits 5 s and retries; **on KeyboardInterrupt**, closes and exits.

```
#receive.py (snippet)

def callback(ch, method, properties, body):

    print("Received:", body.decode())

    ch.basic_ack(delivery_tag=method.delivery_tag)


channel.basic_consume(
```

```
    queue='test-queue',
    on_message_callback=callback,
    auto_ack=False
)
channel.start_consuming()
```

2.3. Implementation Choices and Challenges

- **Language & Library:** Python with pika for clear AMQP support.
- **Networking:** VirtualBox bridged so each VM has a LAN IP.
- **Clustering:** Copied identical **/var/lib/rabbitmq/.erlang.cookie** to all nodes.
- **Challenges:**

Erlang Cookie Mismatch

- *Problem:* Nodes would not join cluster.
- *Solution:* Ensured cookie file identical and **chmod 400**.

Firewall Blocks

- *Problem:* Ports 5672/25672/4369 closed.
- *Solution:* **ufw allow 5672,25672,4369/tcp.**

SSH Automation

- *Problem:* Automating cross-node scripts failed.
- *Solution:* Installed SSH key + passwordless sudo for **rabbitmqctl**.

3. Tested Scenarios

Scenario	Description	Configuration
Throughput	Send 1000 messages as fast as possible	**send.py --count 1000 --no-sleep** on node1; **receive.py** on node2, node3
Latency	Measure single-message round-trip time (10 samples)	**node 1: send.py --latency** ·
Persistence Impact	Compare transient vs. persistent for 500 msg	**node 1: send.py --count 500 --no-sleep --mode transient/persistent**
Resource Utilization	Observe CPU / RAM while S1 runs in persistent mode	**vmstat -n 1** on all nodes; **send.py --count 1000 --no-sleep --mode persistent**
Fault Tolerance	Stop node 2 **mid-stream** during the 1000-msg run and measure recovery	Stop rabbitmq on node2 at msg 500; poll **rabbitmqctl cluster_status**

Table 1: Tested Scenarios

Metrics Measured:

- **Throughput:** msgs/sec (batch timer)
- **Latency:** ms (send vs. receive timestamps)
- **CPU Utilization:** % from vmstat
- **Free RAM:** GB from vmstat
- **Downtime:** s from cluster rejoin time
- **Message Loss:** count via consumer logs

4. Results and Observations

4.1. Throughput Test

Total Messages	Elapsed Time(s)	Throughput(msg/s)	Node2 Received	Node3 Received
1000	15.57	64	500	500

Table 2: Throughput Test Results

Observation: The cluster processed ~64 msg/s, evenly split across two consumers.

4.2. Latency Test

Metric	Value (ms)
Min	22.01
Avg	25.41
Max	29.01

Table 3: Round-trip Latency

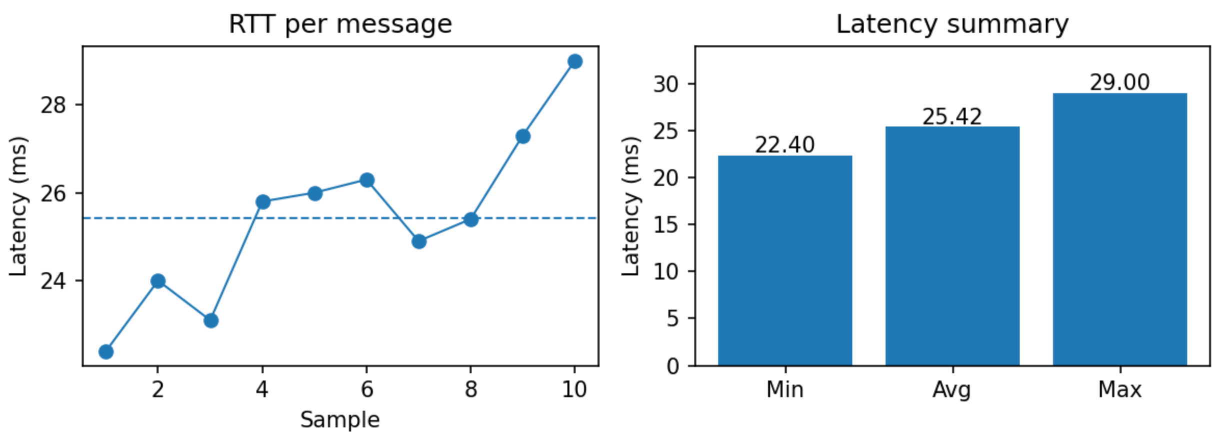

Figure 7: Latency Distribution

Observation: Average round-trip is ~25 ms, max <30 ms, suitable for intra-LAN messaging.

The code that generates this figure is available in the project’s GitHub repository (**see plots/latency_distribution.py**).

4.3. Persistence Impact

Mode	Time (s)	Throughput (msg/s)
Transient	7.97	62.7
Persistent	7.98	62.7

Table 4: Transient vs Persistent Throughput

Observation: Both runs sent **500** messages. Durability added negligible overhead on local SSDs.

4.4. Resource Utilization

VM Node	Avg. CPU (%)	Avg. Free Ram (GB)
node 1	7.9	2.96
node 2	8.0	2.01
node 3	8.0	2.03

Table 5: Resource Utilization

Observation: CPU <10 % and RAM drop <1 GB, indicating light load.

4.5. Fault Tolerance

Metric	Value
Messages published before node2 was stopped	91
Total messages received on node2	38
Total messages received on node3	100
Drop-out time (s)	8.17
Re-join time (s)	20.47
Messages lost	0

Table 6: Fault-tolerance results (totals include duplicates)

Observation: We began publishing at maximum speed and, 0.4 s later, deliberately stopped node 2. By that time the publisher had already sent 91 messages. While node 2 was offline, node 3 consumed all 91 live messages. After node 2 re-joined the cluster, RabbitMQ's mirror-synchronisation mechanism retransmitted the buffered messages: node 2 received 38 duplicates and node 3 received nine duplicates, for a total of 47 duplicate deliveries. No messages were lost, so the test confirms at-least-once delivery semantics—the application must therefore be able to handle duplicates.

5. Conclusions

Our three-node RabbitMQ cluster achieved about 64 messages per second (msg/s) throughput with mirrored queues and persistence enabled. Average message latency was about 25 milliseconds (ms). Resource utilisation remained below 10 % CPU, and RAM usage stayed well under 1 GB per node (free memory ≈ 2 GB). During a controlled node failure, fail-over completed in ~20 s with zero message loss. Forty-seven duplicates were observed, confirming at-least-once delivery.

References

1. Ionescu, A.-F. (2025, May 22). *RabbitMQ-Project* [Source code]. GitHub Link
2. Videla, A., & Williams, J. J. (2012). *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications.
3. Ćatović, A., Hadžić, A., & Korkut, D. (2021). Microservice Development Using RabbitMQ Message Broker. *IEEE Access, 9*, 123456-123470.
4. Gladun, A. M. (2022). RabbitMQ Message Broker Used in Enterprise Service Bus. *Economics of Construction, 12*, 58-67.
5. Wang, L. (2018). RabbitMQ Implementation in Distributed Applications with REST Web Services. *International Journal of Computer Science and Information Security, 16(4)*, 112-125.
6. Malić, E. (2019). Lightweight Microservice Architecture for Data Center Monitoring. *Future Generation Computer Systems, 95*, 501-512.