TOLERANCER: A Fault Tolerance Approach for Cloud Manufacturing Environments

Auday Al-Dulaimy*, Christian Sicari†, Alessandro V. Papadopoulos*,
Antonino Galletta†, Massimo Villari†, and Mohammad Ashjaei*

* Mälardalen University, Sweden, email{name.surname}@mdu.se
† University of Messina, Italy, email{name.surname}@unime.it

***Abstract*—The paper presents an approach to solve the software and hardware related failures in edge-cloud environments, more precisely, in cloud manufacturing environments. The proposed approach, called TOLERANCER, is composed of distributed components that continuously interact in a peer to peer fashion. Such interaction aims to detect stress situations or node failures, and accordingly, TOLERANCER makes decisions to avoid or solve any potential system failures. The efficacy of the proposed approach is validated through a set of experiments, and the performance evaluation shows that it responds effectively to different faults scenarios.**

***Index Terms*—Cloud computing, Edge computing, Cloud manufacturing, Fault tolerance, Reliability.**

I. Introduction

In a highly competitive environment, manufacturers need to enhance their infrastructure to increase revenue. Thus, they adopted the cloud manufacturing model as it offers an encapsulated variety of manufacturing resources as services to meet the demands of customers with lower costs and better performance. However, cloud manufacturing in its classical architecture has some limitations. The architecture of the cloud is of a centralized fashion that assumes stable connectivity to offer convenient services. But uninterrupted connection cannot be guaranteed. At the same time, the Industrial Internet of Things (IIoT) applications need to be able to work even when the connection is temporarily unavailable or under degraded conditions. In addition, cloud computing assumes that there is enough bandwidth to transfer data between the physical location of the manufacturers' devices and the cloud data centers, which is not guaranteed as well. Moreover, transferring massive data results in network bottlenecks and leads to latency issues for applications [1], and this may cause a deterioration in computing performance. Such limitations in the cloud layer may result in system failures. Thus, the manufacturers utilize the edge computing model to complement the cloud by decentralizing the compute and storage resources and moving them closer to the plants and factories aiming at improving the quality of service.

In the context of manufacturing, the Open Manufacturing Platform (OMP) stated in one of their white papers [2] that: "*Edge computing describes a system of decentralized edge nodes, which reside near the physical origin of the data. Edge nodes must be able to run arbitrary containers and are managed centrally. An edge node connects to both the cloud level and the production asset level and can temporarily run offline*". However, systems may fail at the edge layer as well, mainly due to the low scalability and limited resource capacity at this layer [3]. What makes providing reliable and fault tolerances services in manufacturing environments more complex is the relationships among manufacturers. The manufacturers need different types of services as the life cycle of their product development is composed of different stages. The products' dynamic, complex, and long life-cycle processes may result in service failure [4]. Thus, there is a need to manage the failure that may occur to cloud services offered to the manufacturing and industrial sectors. Without proper fault tolerance approaches, multiple manufacturing services will fail to lead to great losses.

Investigating service failure in cloud manufacturing and proposing a solution is the aim of this paper. More precisely, this work is trying to answer the following research questions:

(1) How to design a robust fault tolerance approach to avoid and/or deal with any possible failure in the nodes that host IIoT applications?

(2) How to design a hybrid (Proactive/Reactive) fault tolerance approach in the edge-cloud manufacturing environments?

The rest of this paper is organized as follows: Section 2 categories and gives a brief literature review of the works done to solve the problem of system failure. The proposed system model is presented in Section 3. In Section 4, we perform extensive experiments to evaluate the proposed approach. Section 5 concludes the paper.

II. Related work

This section presents the related works. Most existing fault-tolerance approaches can be classified into two categories: proactive approaches to avoid the expense of systems fault by predicting it in advance and reacting accordingly, and reactive approaches to handle the system's fault after happening by utilizing adequate techniques. However, the literature includes few hybrid approaches.

This work is supported by the framework of XPRES project at Mälardalen University.

There are several related existing proactive approaches. In [5], the authors proposed a proactive fault tolerance approach aiming at preventing system faults within the federated cloud environment. The environment is modeled as a multi-objective optimization problem that maximizes the profit and minimizes the VMs migration cost. The approach can re-distribute VM from faulty providers to non-faulty ones within the federation. However, this work considered applications that are served by VMs at the cloud layer and did not consider the features of the edge nodes. In [6], the authors proposed a fault-tolerant approach to maintain system availability. The approach includes the following components: fault manager, controller, and load balancer that work together to ensure a fault-tolerant environment via redundancy, optimized selection, and checkpointing. The work in [7] presented an approach that models the temperature of the CPUs in a virtualized cluster to expect a potential failure in a specific physical machine (PM), and accordingly, migrates VMs from the detected PM to be hosted on another PM. The selection of the new PM is represented and solved as an optimization problem. However, this work targeted VMs in the cloud environment, not containers at the edge. In [8], the authors proposed a fault-tolerant approach to work in the fog layer. The approach utilizes the checkpointing technique, and at the same time, it applies load balancing based on Bayesian classification to consider the energy efficiency of the fog devices. However, the approach was not evaluated in a real testbed. The work in [9] presented a preemptive migration prediction model, called PreGAN, to detect and classify faults in edge computing environments. PreGAN can migrate services from one node to another based on the features of the potential detected failure.

On the other hand, there are related reactive approaches. In [10], the authors presented a two-stage fault tolerance approach (off-line and online) to improve the reliability of the manufacturing network. The off-line stage ranks the manufacturing services according to their importance in fault tolerance, then the critical services are replicated. While the online stage performs a heuristic algorithm for replacing the failed services. The work in [11] presented a three-layer approach to solving the problem of system failure in cloud-edge environments. The three layers (Application Isolation, Data Transport, and Multi-cluster Management) work together to re-schedule failed processes on other available nodes. In [12], a fault-tolerant approach for recovering the failed IoT edge applications is presented. It manages and re-configures container-based IoT software in a reliable way upon software failure detection. However, the authors stated that the approach is not suitable to run on low-powered devices. The authors in [13] leveraged both Primary-Backup (PB) fault-tolerant and Deep-Q-learning-Network (DQN) techniques to ensure safe execution for the edge services. However, the approach was not evaluated in a real environment.

There are also a few hybrid approaches, combining both reactive and proactive approaches. In [14], the authors presented a hybrid model to take fault tolerance actions: proactive actions after predicting the failure probability, and reactive actions that employ replication and checkpointing techniques. The work in [15] presented a fault-tolerance approach that utilizes two directions: the first is performing a VM migration based on a failure prediction technique, and the second is by doing VM checkpointing.

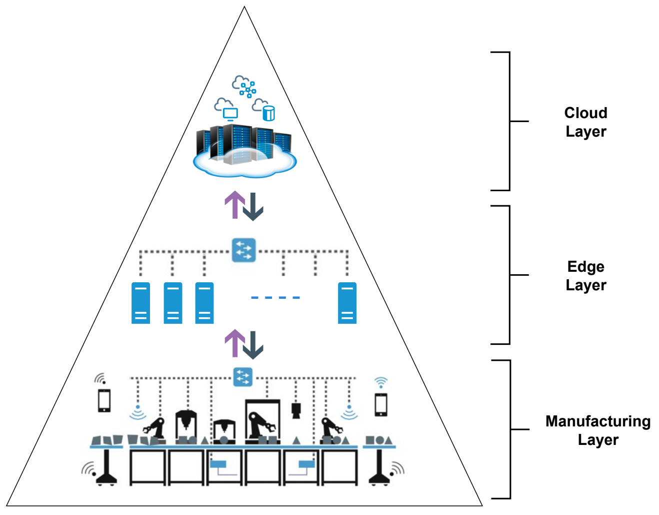


Figure 1: System model.

To the best of our knowledge, our work is the first in presenting a hybrid fault tolerance approach in cloud manufacturing environment.

III. System model

The system model targets the hierarchical edge-cloud computing architecture (Manufacturing Environment) in order to prevent and/or manage the potential system failures in such environments. The system is divided into three different layers: Manufacturing layer, Edge layer, and Cloud layer, as shown in Figure 1, and it includes N_T heterogeneous nodes that are prone to failure, which are distributed on the edge (N_E) and the cloud (N_C) layers, such that $N_T = N_E + N_C$.

As many manufacturers prefer to process their data on-site (mainly for security reasons), this work is focusing on investigating the system failure in the edge layer. The nodes at the edge layer N_E can host VMs and/or containers. Containers offer a lightweight, portable, and high-performance virtual entity compared to VMs. In addition, the size of container images is smaller than VM images. This is better to be adopted in the constrained devices at the edge layer, and also makes applications launching faster than VM-based applications [16], [17].

The manufacturing environment is heterogeneous. It includes many edge devices, installed at different times, with different configurations and operating systems. More devices could be added and integrated into the system anytime. Some systems adopt a *single-master multi-workers* architecture to maintain load balancing and high availability. Such systems are easier to be managed compared to full peer-to-peer systems. But at the same time, as the their management depends on a single master node, they come with a major issue, which is the potential single point of failure, and consequently, a high failure rate.

TOLERANCER approach, which works within the manufacturing environment, aims at avoiding any probable single point of failure. To do so, each node in the system model has the same role in monitoring and taking fault tolerance actions. Following this full distributed peer-to-peer architecture, where each node $i \in N_E$ is connected with the other nodes, we designed an approach that can be hosted and run on all edge devices. The approach's components are light entities, so that the nodes with low and medium computational capabilities (edge nodes) can host them.

Regardless of the type, capabilities, configuration, or operating system of the device, TOLERANCER can be run on it if the device can run Dockers. Docker, and any general full container-based approach, is a perfect option to be considered in the targeted environment because Docker can encapsulate the system components and then run on different hardware and operating systems. Containerization helps in hiding such differences, automatically fetching and deploying containers on the nodes. By connecting each device with the other devices at the edge layer, a federation is created. The federation's members are the edge devices which can be configured at the federation bootstrap or at the run-time. When the federation is ready, the containers hosted on the edge devices are monitored, and the statuses of the devices are observed. Each member in the federation is responsible for the management of itself (no master node to be recognized in the federation). The member is also responsible for communicating and exchanging data with the other members in the federation. In this way, we can avoid the single (or n-points) point of failure situations. The events that TOLERANCER can monitor include: (1) high resource stressing (overloaded), (2) service down, and (3) device off.

The IIoT applications or services are deployed as Docker containers. The TOLERANCER tries to keep these services available at any time. When one or more of the previous events occur, TOLERANCER triggers fault a tolerance action(s) by involving the related peer or the other peers in the federation. These actions could be Proactive and/or Reactive actions The TOLERANCER approach comprises three light key units: Middleware Unit (MidU), Monitoring Unit (MonU), and Planning Unit (PlaU). Each node $i \in N_E$ hosts these units which are collaborating with each other aiming at avoiding system failures and resolving them upon the failure detection. In other words, it is a proactive/reactive approach. The TOLERANCER monitors the system periodically according to a predefined cycle, and the cycle interval is variable so it can be tuned based on the system status. The units are described as follows (Refer to Figure 2):

A. *Middleware Unit (MidU)*

This unit is composed of two components: the MESSAGE-BROKER and the SHARED-MEMORY. Both components are deployed in cluster mode where each edge device runs a single instance of both components.

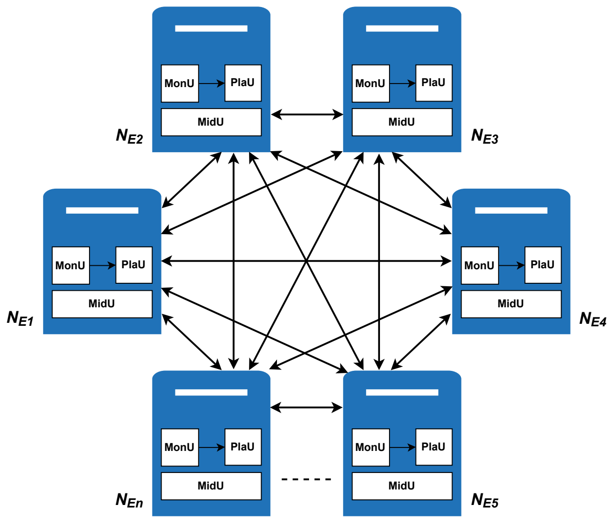


Figure 2: Nodes at the edge layer.

- MESSAGE-BROKER: It is used to exchange messages between the nodes at the edge layer. To exchange information, each node employs its own MESSAGE-BROKER to send messages to the other peers. It is needed to guarantee that the system works properly.
- SHARED-MEMORY: This component is used to store all the information generated by the federation, and to make it accessed by all nodes $\in N_E$. SHARED-MEMORY stores the information about the federation in general, and about the nodes themselves. For example, the number of the edge nodes and devices, their IDs, their health information, the running containers they host, and the migration processes carrying on.

All this information is generated and used by the Monitoring Unit's components.

B. *Monitoring Unit (MonU)*

The main functions of this unit are: collecting data about system status, and analyzing the collected data. To achieve these functions, MonU includes components that allow to log and monitor the services running on the edge nodes. The components are LOGGER and ANALYZER.

- LOGGER: it logs the status of the system and put the information into a written record periodically, based on a predefined interval. LOGGER records the following: (1) CPU usage, (2) Memory usage, (3) container status (running or failed) which is a SW-related failure that depends on the application itself, and (4) the device status (on or off) which is a HW-related failure.
  All the data is stored in the SHARED-MEMORY component, in order to be available to all nodes in the federation.
- ANALYZER: It is responsible for analyzing the data stored in the SHARED-MEMORY which are collected by LOGGER. ANALYZER checks the data related to each peer individually, and also the whole system status. When ANALYZER notices any cautionary data that may result

in system failure, it passes an alert to the PlaU to take an action (proactive/reactive reaction). The cautionary situation can result in the following cases:
- Case 1: over-utilized CPU and/or memory.
- Case 2: container with a failed state.
- Case 3: device with an off state.

C. *Planning Unit (PlaU)*

PlaU uses the analysis resulting from MonU (The resulted three cases), and accordingly, performs fault tolerance action(s). Case 1 necessitates a proactive action to maintain reliability and to avoid any potential failure, while Case 2 and Case 3 necessitate a reactive action as the failure already happened. PlaU includes two components, they are SCHEDULER and MIGRATOR.

- SCHEDULER: It is responsible for specifying the following operations when migration is needed: the service(s) to be migrated, the source device(s) that hosts the service(s), and the destination device(s) to host the migrated service(s).
  The SCHEDULER uses the data stored in the SHARED-MEMORY in order to take the decisions. It can employ different scheduling algorithms like Round Robin or even more complex ones.
- MIGRATOR: It receives three parameters as input: The IDs of the source devices, the IDs of the destination devices, and a list of services to be migrated. The MIGRATORs on the source and destination devices collaborate with each other to perform the migration process. After migration, the services run on the destination node and are removed from the source node. If the migration process does not perform correctly, the MONITOR can notice this in the next monitoring cycle in order to find a new destination. The new system status is stored in the SHARED-MEMORY components of all nodes, so the nodes in the system will be aware of what is the status resulted after the migration process.

D. TOLERANCER *description*

This section describes the communication and the main processes involved in TOLERANCER using some high-level pseudo-codes.

1) Communication: Communication inside the TOLERANCER system is done using a special distributed Message Oriented Middleware (MoM). This MoM is implemented using RabbitMQ, a very popular open-source project that in turn implements the Advanced Message Queuing Protocol (AMQP). The AMQP is very similar to a Publish/Subscribe protocol, where producers push messages in a queue distinguished by a key called “Topic”, and consumers connected to the same queue read them. AMQP adds a fourth element called ”Exchange” which, in a nutshell, ensures that every message arrives at the destination, finally guaranteeing a high QoS. These characteristics ensure stable communication among the TOLERANCER’s nodes and then good reliability to the entire system.

2) Processes: The main processes in TOLERANCER are Logging, Analyzing, Scheduling, and Migration.

The LOGGER works as described in Algorithm 1. It reads information from the nodes of the system aiming at maintaining a stable and balanced cluster. The algorithm takes a specific node as an input and outputs a stored and shared status about that node. The algorithm is activated for all nodes periodically based on a predefined period. The information is collected using an API as described in line 3, and then, in line 4 this information is stored in the SHARED-MEMORY together with the current timestamp.

The ANALYZER works in two phases. The first phase is described in Algorithm 2. Block 2-15 shows that the algorithm works over all nodes in the federation. For each node, it tries to get exclusive access using the distributed semaphore that is managed by the SHARED-MEMORY. Then, the SHARED-MEMORY gives the status information recorded by the LOGGER of the same node to the ANALYZER. The ANALYZER in step 8 examines the collected information to understand if the node is working or not. If the node is not working, The ANALYZER updates the node’s status to FAILED in step 9. The algorithm is also considering the case when the node is working but the LOGGER in not updating the node’s information for a while. In such a case, the ANALYZER will try to contact the node, and if no response, it will set its status as FAILED in step 12.

The second phase is described in Algorithm 3. The ANALYZER obtains a list of the failed nodes in the federation from the SHARED-MEMORY in step 3. For each of the failed nodes, the algorithm in step 6 tries to access the node’s information using the shared semaphore, and in step 7, it gets the list of the container(s) hosted on the failed node.Then, in block 8-14, the algorithm reschedules each container to be hosted on a healthy node based on a specific scheduling algorithm. In this work, we pick the new destination host randomly among the healthy nodes (step 11). The migration is done through the MIGRATOR API as it initializes a transaction for accepting the new container, as described in the Figure 3. The Scheduling process is repeated in block 9-13 until there are no more containers to be rescheduled.

The MIGRATOR described in Algorithm 4 is composed of two functions: $send\_request$ and $receive\_request$.

The $send\_request$ function is invoked by the ANALYZER and used to ask the destination node ($destination\_node$) to host the $container$ that was previously hosted in a failed source node (src_node). This function needs to use the MESSAGE-BROKER’s APIs to send a migration request to the message queue of the $destination\_node$ under the topic $/migration\_request$, as shown in step 3. There is an identifier for each migration request, we refer to it as migration ID, and it is generated in Step 3. Then, in step 4, the function will use it through the Message Broker’s APIs to wait for the request’s response. If the request receives a $SUCCESS$ response, it means that the container is hosted in the $destination\_node$. After that, in step 6, the function updates the SHARED-MEMORY with the new host of the

Algorithm 1: The LOGGER Algorithm.

```
Input: The ID of the node $N_{id}$
Output: The node's status information
Begin
    While $True$
        $x \leftarrow get\_system\_info()$
        $SHARED\_MEMORY.store\_node\_info(N_{id}, x, get\_current\_timestamp())$
End
```

Algorithm 2: Analyzer Algorithm Part 1.

```
Input: Set $N_E$ of the nodes in the federation.
Output: Analyzed system status, the updated nodes' status
Begin
    ForEach $Node_i \in N_E$
        try:
            $SHARED\_MEMORY.acquire\_node\_semaphore(Node_i)$
            $curr\_time \leftarrow get\_current\_timestamp()$
            $state \leftarrow SHARED\_MEMORY.get\_node\_info(Node_i)$
            $validate\_state \leftarrow is\_healthy(state)$
            If $validate\_state = False$
                $SHARED\_MEMORY.set\_node\_state(Node_i, "FAILED", curr\_time)$
            Else If $state.timestamp + INTERVAL\_CHECK < curr\_time$
                If $try\_contact(node) = False$
                    $SHARED\_MEMORY.set\_node\_state(Node_i, "FAILED", curr\_time)$
        catch:
            continue                    ▷ If the semaphore is held by other nodes, simply skip
        end
End
```

Algorithm 3: Analyzer Algorithm Part 2.

```
Input: Set $N_E$ of the nodes in the federation.
Output: Analyzed system status, the updated nodes' status
Begin
    While $True$
        $failed\_nodes \leftarrow get\_failed\_nodes$
        ForEach $Node_i \in failed\_nodes$
            try:
                $SHARED\_MEMORY.acquire\_node\_semaphore(Node_i)$
                $containers \leftarrow SHARED\_MEMORY.get\_containers\_in\_node(Node_i)$
                ForEach $container \in containers$
                    Repeat
                                    ▷ SCHEDULER process that randomly picks a healthy node for hosting the container
                        $destination\_node \leftarrow random\_choice(Nodes - failed\_nodes)$
                        $success \leftarrow MIGRATOR.send\_request(Node_i, destination\_node, container)$
                    Until $success = False$;
            catch:
                continue                ▷ If the semaphore is held by other nodes, simply skip
            end
End
```

container.

The $receive\_request$ function receives the requests. In step 9, the function waits for an incoming migration request message in its queue under the topic $/migration\_request$. When the message arrives, the function gets the faulty src_node, the container ID hosted on it, and the migration ID. In step 10, the function uses the SHARED-MEMORY for getting all information about the container (*i.e.*, the configuration), then it uses this information for verifying that the container is compatible with the node. If the container is reported as compatible, the function extracts the command needed for running the container and executes it in line 13. When it finishes, it sends a success message to the src_node message queue under the $/migration\_request$ topic using the migration identifier as a parameter. The message exchange process between a node that is analyzing another failed node and a node that may host a new container is described in Figure 3.

IV. Performance evaluation

This section evaluates TOLERANCER's ability to recover the faults and move all services deployed in the failed node to another node in the edge layer.

A. Testbed and experiments

Our evaluation testbed is a cluster of nodes that represents the edge layer for a specific manufacturer. We used five nodes: four Raspberry Pi4 and one Nvidia Jetson Nano. The cluster

Algorithm 4: Migrator Algorithm

```
Input: Set $N_E$ of the nodes in the federation.
Output: New container-to-host placement, The updated nodes' status
1  Begin
2      Function send_request($src\_node, destination\_node, container$): bool
3          $migration\_id \leftarrow MESSAGE\_BROKER.publish(destination\_node, "/migration\_request", src\_node, container)$
4          $response \leftarrow$ wait $MESSAGE\_BROKER.listen(Nid, "/migration\_request/ < migration\_id > ")$
5          If response = True
6              $SHARED\_MEMORY.update\_container\_map(src\_node, destination\_node, container)$
7      Function receive_request(): bool
8          While True
9              $src\_node, container, migration\_id \leftarrow$wait $MESSAGE\_BROKER.listen(Nid, "/migration\_request")$
10             $container\_info \leftarrow SHARED\_MEMORY.get\_container\_info(container)$
11             If $is\_compatible(src\_node, container)$
12                 $run\_command \leftarrow container\_info.run$
13                 $execute\_run(run\_command)$
14                 $MESSAGE\_BROKER.publish(src\_node, "/migration\_request/ < migration\_id > ", "SUCCESS")$
15 End
```

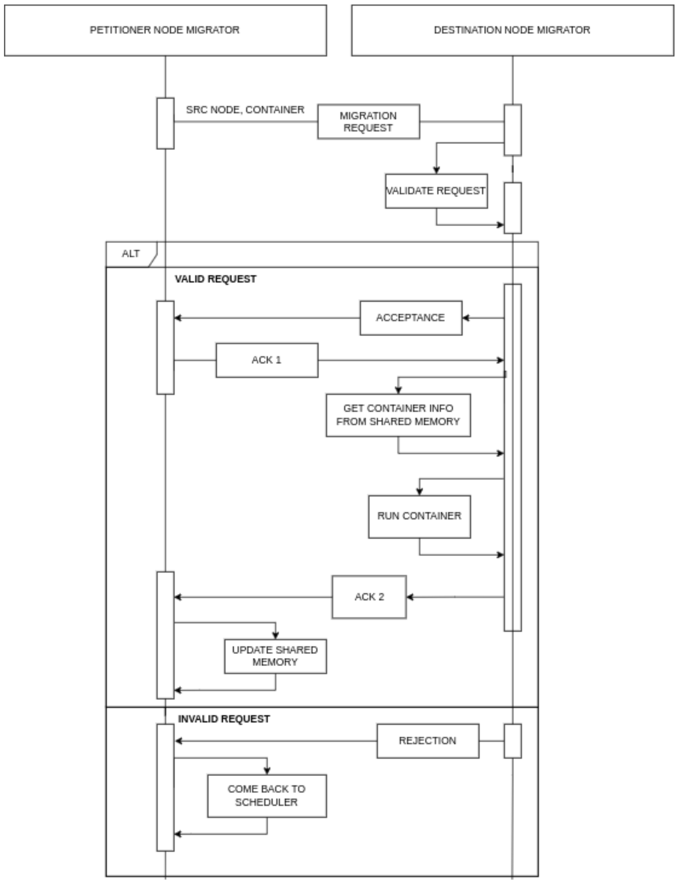


Figure 3: Migrator message exchange

information is summarized in Table I. To evaluate the proposed approach, we targeted an edge cluster, deployed services as containers, run the containers, caused a failure in a specific node (or nodes) of the cluster by disconnecting it (or them) from the network, and then examined TOLERANCER ability to recover the faults. The examination is done by monitoring the capability of the other active nodes to notice the failure and start moving all containers hosted on the failed node to another healthy one. We performed three experiments. In each experiment, we consider a different cluster, as shown in Table II. The cluster in experiment 1 consists of 5 nodes, the cluster in experiment 2 consists of 4 nodes, and the cluster in experiment 3 consists of 3 nodes. With every experiment, we run a different number of containers (*m*), each container can run Nginx web servers, and then, we caused a failure in one node. We considered that the failed node hosts and run a different number of containers as follows: 3, 30, 60, 90, and 120 containers. Then, we checked if the containers on the failed node migrate to another healthy node. In addition, we calculated the time to detect the failure and the time to restore all the containers hosted in the failed node.

Table I: Cluster's nodes characteristics.

Name	Node	CPU	Memory
Edge1	Nvidia Jeston Nano	4-core (ARM v8) 64-bit SoC 2 GHz	4 GB
Edge2	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	4 GB
Edge3	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	4 GB
Edge4	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	8 GB
Edge5	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	8 GB

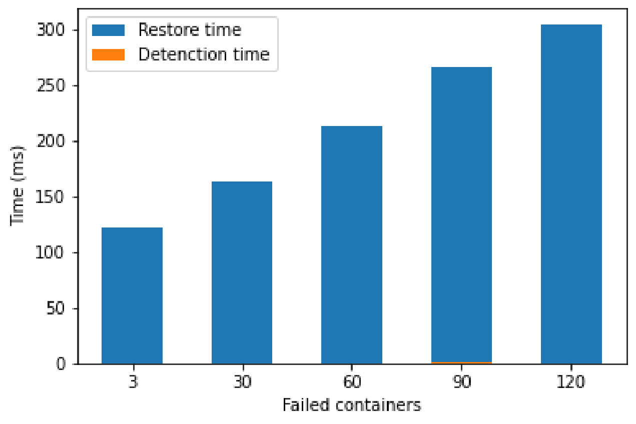
Table II: Clusters configurations.

Cluster size	Edge1	Edge2	Edge3	Edge4	Edge5
5 nodes	✓	✓	✓	✓	✓
3 nodes	✓	✓	✓	✗	✓
3 nodes	✓	✓	✗	✗	✓

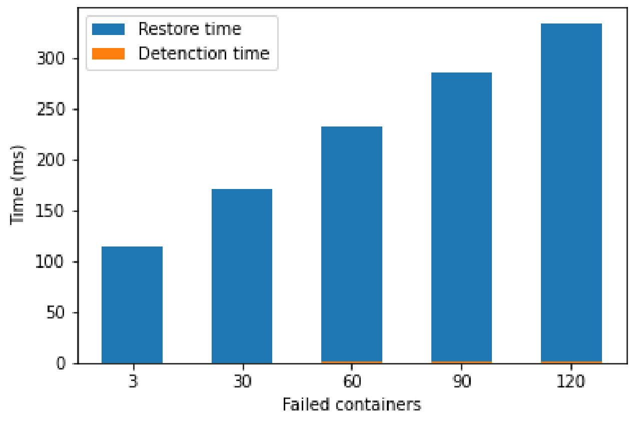
B. Result discussion

In this section, we discusses the performance evaluation of TOLERANCER from the service maintainability perspective.

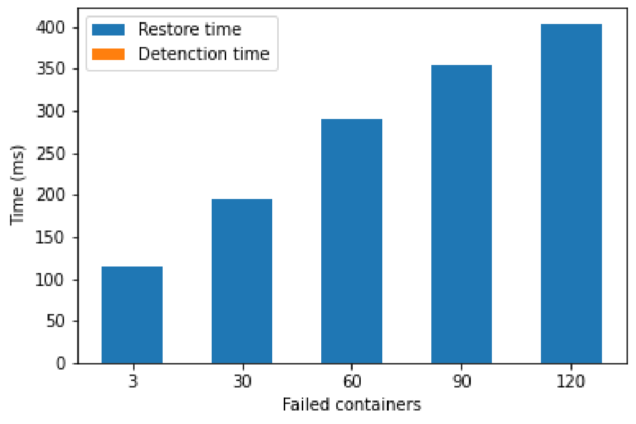
We set the timing to activate the interval value of the LOGGER equal to 5 seconds. Time selection is crucial in meeting the objective of the approach's design. If the interval period is too short, it overloads the system performance by performing more actions (*e.g.*, migration), and if it is too long,



(a) Time to restore in a 5-node cluster



(b) Time to restore in a 4-node cluster



(c) Time to restore in a 3-node cluster

Figure 4: Time to restore in three different clusters.

the approach may not immediately respond to the faults (or to the possible faults).

After deploying the services and running them, we switched off one node in the three different clusters. We noticed that all TOLERANCER's components responded efficiently to such

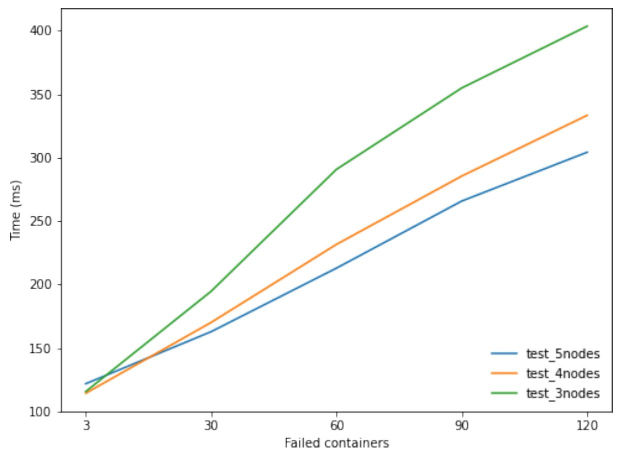


Figure 5: Comparing the *Time to restore* in three different clusters.

fault. A node failure leads to the following actions:

- The LOGGER of the failed node stops writing the health status information of the failed node,
- The ANALYZERs of the healthy nodes notice that there is no information from the failed node, so they try to contact it. After the no-response, the healthy nodes mark the status of the off node as failed.
- After detecting the failed node, the ANALYZERs of the healthy nodes get a list of the containers hosted in the failed node from the SHARED-MEMORY.
- The SCHEDULERs of the healthy nodes are triggered by the ANALYZERs, and the fastest of them reschedule the failed container to be hosted on a new node.
- The node that runs the scheduling process triggers the MIGRATOR of the destination node to accept the incoming container.
- The MIGRATOR of the destination node accepts the incoming container, runs it, and updates the SHARED-MEMORY.

However, as most industrial applications are real-time applications, in each experiment we measured the time needed to rerun the containers after any system failure. In Figure 4, the blue bars represent the time needed to restore the failed containers in the cluster with a fixed number of nodes. We can see that the time needed to restore the containers is directly proportional to the number of containers. This is expected as the migration requests are queued among all the remaining healthy nodes and hosted by the destination node(s) one by one. The time required to restore any container is calculated as in Equation 1:

$$T_{Restore} = T_{Detect} + T_{Fcontainers} + \frac{T_{Rerun}}{N_{Hdevices}} \quad (1)$$

where:

- $T_{Restore}$ is the time required to restore the failed container,
- T_{Detect} is the time required to detect the failure,

- $T_{Fcontainer}$ is the time required to find the failed container,
- T_{Rerun} is the time required to rerun all the failed containers,
- $N_{Hdevices}$ is the number of healthy devices.

In the same figure, we can see the time needed to detect the failure, which is calculated as in Equation 2:

$$T_{Detect} = DetectFailure_{TS} - Failure_{TS} \tag{2}$$

where:

- $DetectFailure_{TS}$ is the detected failure timestamp.
- $Failure_{TS}$ is the failure timestamp.

It is clear from the figure that the failure detection time (which is represented by orange color) is quite less than the actual time needed to restore the services. The detection time is a small part of the total time needed to restore service. In other words, the figure shows that the longest period to restore the container is consumed by the scheduling and migration operations of the PlaU, not by the logging and analysis operations in the MonU.

To understand the effects of the cluster size (*i.e.*, the number of nodes in the cluster) on the performance, refer to Figure 5. It compares between the time required to restore the containers in the three clusters presented in Figure 4. Each line describes a single cluster behavior. If we ignore the network failures, we can conclude that the larger cluster results in better performance, as the time needed to restore the services is less. The last part of Equation 1, $\frac{T_{Rerun}}{N_{Hdevices}}$, props this conclusion, as increasing the number of healthy nodes leads to decrease the value of this part, and consequently, decrease $T_{Restore}$.

Moreover, the figure also shows that as the number of failed containers increases, the gap between the clusters increases as well. This is because increasing the number of migration requests creates system congestion that leads to performance degradation.

V. Conclusion and Future Directives

The paper presents an approach, called TOLERANCER, to solve the software and hardware-related failures in cloud manufacturing environments. TOLERANCER makes decisions to avoid or solve any potential system faults. Experiments on a real testbed show that proposed approach provides on-the-fly automatic actions to handle both hardware and software-based failures. To mention, this version of the paper includes the performance evaluation of the reactive part of TOLERANCER, while the performance evaluation of the proactive part will be shown in an extended version.

Currently, we are working on expanding our approach to include more results by testing larger clusters, and by trying different time intervals to find its effects on the system. More constraint(s) could be considered (*e.g.*, deadline of the running services). Besides, the nodes at the cloud layer could be integrated with the nodes at the edge. In addition, managing other failure types (such as security-related failures) is a future direction of this work.

References

[1] A. Al-Dulaimy, W. Itani, J. Taheri, and M. Shamseddine, "bwslicer: A bandwidth slicing framework for cloud data centers," *Future Generation Computer Systems*, vol. 112, pp. 767–784, 2020.

[2] Open Manufacturing Platform (OMP), "Edge computing in the context of open manufacturing," https://open-manufacturing.org/wp-content/uploads/sites/101/2021/07/OMP-IIoT-Connectivity-Edge-Computing-20210701.pdf, 2021, accessed: 2021-10-20.

[3] W. Du, X. Zhang, Q. He, W. Liu, G. Cui, F. Chen, Y. Ji, C. Cai, and Y. Yang, "Fault-tolerating edge computing with server redundancy based on a variant of group degree centrality," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 198–214.

[4] F. Tao, L. Zhang, Y. Liu, Y. Cheng, L. Wang, and X. Xu, "Manufacturing service management in cloud manufacturing: overview and future research directions," *Journal of Manufacturing Science and Engineering*, vol. 137, no. 4, 2015.

[5] B. Ray, A. Saha, S. Khatua, and S. Roy, "Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment," *IEEE Transactions on Cloud Computing*, 2020.

[6] B. Mohammed, M. Kiran, K. M. Maiyama, M. M. Kamala, and I.-U. Awan, "Failover strategy for fault tolerance in cloud computing environment," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1243–1274, 2017.

[7] J. Liu, S. Wang, A. Zhou, S. A. Kumar, F. Yang, and R. Buyya, "Using proactive fault-tolerance approach to enhance cloud service reliability," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 1191–1202, 2016.

[8] A. Sharif, M. Nickray, and A. Shahidinejad, "Fault-tolerant with load balancing scheduling in a fog-based iot application," *IET Communications*, vol. 14, no. 16, pp. 2646–2657, 2020.

[9] S. Tuli, G. Casale, and N. R. Jennings, "Pregan: Preemptive migration prediction network for proactive fault-tolerant edge computing," *arXiv preprint arXiv:2112.02292*, 2021.

[10] Y. Wu, G. Peng, H. Wang, and H. Zhang, "A two-stage fault tolerance method for large-scale manufacturing network," *IEEE Access*, vol. 7, pp. 81 574–81 592, 2019.

[11] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th world forum on internet of things (WF-IoT)*. IEEE, 2018, pp. 813–818.

[12] K. Olorunnife, K. Lee, and J. Kua, "Automatic failure recovery for container-based iot edge applications," *Electronics*, vol. 10, no. 23, p. 3047, 2021.

[13] T. Long, P. Chen, Y. Xia, N. Jiang, X. Wang, and M. Long, "A novel fault-tolerant approach to web service composition upon the edge computing environment," in *International Conference on Web Services*. Springer, 2021, pp. 15–31.

[14] M. Amoon, "A framework for providing a hybrid fault tolerance in cloud computing," in *2015 Science and Information Conference (SAI)*. IEEE, 2015, pp. 844–849.

[15] Y. Sharma, W. Si, D. Sun, and B. Javadi, "Failure-aware energy-efficient vm consolidation in cloud computing systems," *Future Generation Computer Systems*, vol. 94, pp. 620–633, 2019.

[16] B. I. Ismail, E. M. Goortani, M. B. Ab Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe, "Evaluation of docker as edge computing platform," in *2015 IEEE Conference on Open Systems (ICOS)*. IEEE, 2015, pp. 130–135.

[17] A. Celesti, D. Mulfari, A. Galletta, M. Fazio, L. Carnevale, and M. Villari, "A study on container virtualization for guarantee quality of service in cloud-of-things," *Future Generation Computer Systems*, vol. 99, p. 356 – 364, 2019.