Adaptive Fault-Tolerant Strategy for Latency-Aware IoT Application Executing in Edge Computing Environment

Muhammad Mudassar, Yanlong Zhai, *Member, IEEE*, and Liao Lejian, *Member, IEEE*

***Abstract*—Edge computing has recently evolved that offers to execute jobs efficiently by pushing cloud capabilities to edge of the network, this improves the quality of services to latency-oriented Internet of Things (IoT) applications when compared with cloud computing. By using current smart devices as edge nodes, edge computing can provide elastic resources that allow distributed data processing in a decentralized way. Still these smart devices are resource constrained in nature and tends to face a high failure rate than traditional distributed systems, the implementation of a fault-tolerant system that ensures the reliability and application availability becomes a key requirement. In this article, we propose a fault-tolerance methodology based on checkpointing and replication for the edge computing. Our proposed system uses a smart checkpointing for the IoT application tasks executing in a distributed edge network, and by replicating the checkpoint files on alternative edge nodes in the vicinity allowed to increase the system reliability. The experimental results show that our approach is effective in terms of reliability and availability of tasks executing in the edge network along with meeting deadlines of an IoT application.**

***Index Terms*—Distributed computing, edge computing, fault tolerance, Internet of Things (IoT).**

I. Introduction

THE EVOLUTION of the Internet has transitioned from the Internet of content to Internet of Things (IoT) with a vision, where the Internet is extended to a vast network of interconnected smart objects. These objects are usually termed IoT devices that interact with physical world and gather data and information from their environment, which is normally processed on cloud. However, as the cloud systems are located within the core, the network may fail to resolve some needs of IoT applications, such as lower latencies, limited bandwidth, Quality of Service (QoS), availability, and reliability.

With drastic increase in the number of connected devices and increasing demand of real-time processing, the rise of a distributed processing paradigm is noticed in the form of edge/fog computing [1]. Distributed edge computing extends cloud computing facilities, such as computation power and storage requirements closer to the end user at the edge of the network [2]. This allowed mission-critical applications to be deployed at edge devices, which can result in real-time data analysis, reduce latency, high scalability, restrain network traffic, and improved QoS. As current smart devices are becoming more powerful in terms of processing capability, these devices being a part of the edge network can provide elastic resources for distributed data processing. Especially, for the IoT applications requiring high-end processing can be executed in the distributed edge computing environment. This distributed edge processing also protects data and application from the drawbacks of the traditional centralized architecture.

The efficiency of IoT applications relies on the trusted execution environment, attributes, such as the availability and reliability are of high priority while addressing IoT solutions related to the QoS. To provide a trusted IoT environment, faults and errors are key threats faced by low-power IoT end devices and also by edge nodes. These faults and errors can result in node failure or a service failure leading to a complete failure or degradation of the overall IoT system. An efficient fault-tolerance system can mitigate faults by detecting and recovering from faults during the runtime to prevent a complete failure.

Providing a reliable system is a significant challenge for IoT applications running in the edge network. For some scenarios, the fault tolerance becomes very important, such as self-driving cars, a hazard detection system, and airplane navigation systems. Failures in edge computing occur regularly because of low stability, power failure, hardware failure, network failure, or service failure. Edge devices are mobile and heterogeneous by nature, hence adding more difficulties to construct a reliable system.

The reliability at the edge network is related to provide successful services to the end user while considering the latency requirements of IoT applications. If the reliability of services running in the edge computing environment is low, then the service degradation or services outage happens frequently. On the other hand, if the reliability is high, then the resource-limited edge devices experience long service time, which introduces application delay. Therefore, both reliability and latency requirements are important issues while designing a fault-tolerance system for the edge network.

Manuscript received 26 May 2020; revised 23 February 2021; accepted 3 January 2022. Date of publication 18 January 2022; date of current version 25 July 2022. This work was supported by the Equipment Pre-Research Field Foundation under Grant 61400010104. *(Corresponding author: Yanlong Zhai.)*

Muhammad Mudassar was with the School of Computer Science, Beijing Institute of Technology, Beijing 100081, China. He is now with the Department of Computer Science, COMSATS University Islamabad (Vehari Campus), Vehari 61100, Pakistan (e-mail: muhammad.mudassar@cuivehari.edu.pk).

Yanlong Zhai and Liao Lejian are with the School of Computer Science, Beijing Institute of Technology, Beijing 100081, China (e-mail: ylzhai@bit.edu.cn; liaolj@bit.edu.cn).

Digital Object Identifier 10.1109/JIOT.2022.3144026

The most popular approaches for providing fault tolerance are checkpointing and replication [3]. In the replication, key components of a system are repeated on resources available in the network or hardware and software [4]–[6]. In the case of a failure, the backup of a failed component is available and the system continues to work with full efficiency. Checkpointing consists of saving specific data according to a predefined setting (e.g., after some time). In the case of failure, execution of the system can be resumed to a previously known working state of the system by using the latest checkpoint files, therefore reducing the time and amount of computations after an event of failure [7], [8].

These techniques of replication and checkpointing are widely used in distributed computing, cloud computing, and IoT systems for developing fault-tolerant systems. In [9], a smart checkpointing-based fault tolerance strategy is used for cloud computing tasks. The replication is also used as a fault tolerance strategy in the cloud computing [3]. Research work in [10] used the data replication technique to provide the maximum data availability for a high node failure rate in the IoT systems. Cherrier *et al.* [11] used the checkpointing methodology for failure recovery on large-scale IoT applications. A migration-based fault-tolerance strategy was proposed by Aïssaoui *et al.* [12] to migrate the software between gateways that will also help in geographic movement of IoT devices known as communication gateways. Some research works [13], [14] used checkpointing for memory backup of an individual IoT device. The techniques proposed in the literature are mostly concerned toward the reliability of a single IoT device, along with various other problems, such as high overhead, high resource consumption, growth in the network traffic, and increase in processing time. None of them has discussed fault tolerance of IoT applications running in distributed edge nodes.

Designing a fault-tolerance system for IoT application executing in a distributed edge computing environment is challenging as it implies that providing reliability has to be consistent with respect to the edge context, i.e., provide real-time analytics, with minimum latency and fewer network traffic. The fault tolerance in the context of edge computing has to handle the following challenges.

1) *Dynamic Devices:* Edge nodes can be mobile nodes so any type of synchronization cannot be achieved for the replication of multiple services or multiple nodes.
2) *Limited Resources:* Individual edge nodes have limited resources available to them; hence, tasks running on edge nodes have to be loosely coupled and small in size. In this case, the execution time (ET) is more delicate to the effect of checkpointing/restore process for large checkpoint files.
3) *Failure Rate in Edge Network Is High:* Edge devices have a high failure rate than traditional distributed systems and cloud infrastructure. This makes it difficult to rely on just a single device to provide reliability for the system.

The above reasons made it difficult to use traditional fault-tolerance techniques while handling faults in the edge computing environment. A combination of checkpointing and replication techniques copes with the challenges of an edge computing environment. In addition, minimizing the overload and resource consumption of the resources limited edge nodes.

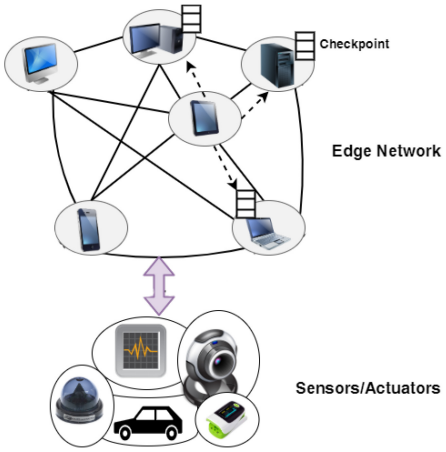


Fig. 1. Checkpointing and replication in the edge network.

Fault tolerance for latency-aware IoT applications executing in the edge computing environment is required to be handled efficiently, and checkpointing and replication are the possible techniques that can be used as per requirements based on the available resources in the edge network. In this article, first, we have proposed to measure the available resources smartly based on a subtask ET and use them for backup to achieve the required reliability level. Then, a decentralized backup strategy is implemented for the edge computing environment based on checkpointing and replication techniques. This methodology is self-managing in terms of performing the checkpointing and replicating it on a pool of available edge nodes. The checkpointing process will help to reduce the computations during the backup process and replication will help to achieve the smaller latency. The overall result is the efficient utilization of edge node resources and to achieve a better fault tolerance.

Our methodology uses adaptive uncoordinated checkpointing that will ensure the reliability and by replicating these checkpoint files to make sure the availability of the overall application in a network where the failure rate is high. The checkpoint files are stored on alternative edge nodes in the distributed edge environment as shown in Fig. 1, which can replace the failed nodes. This will help to reduce the recovery time. We will use incremental checkpointing and asynchronized replication to reduce the checkpointing overhead. An evaluation in terms of simulation shows the effectiveness of our proposed methodology with respect to reliability and availability of the system.

II. Related Work

Fault tolerance denotes the capability of the system to provide expected services regardless of the errors, failures, or faults caused within the system, and the core objective is to avoid an overall system failure in the presence of faults. For the heterogeneous IoT computing environment, a fault-tolerant system is critical to provide the reliability and QoSs. The popular techniques for fault tolerance are replication and

checkpointing. In replication, critical parts (e.g., process, data, communication path) of the system are replicated using redundancy techniques, and in the case of failure, the backup takes over to keep the system functional [15]. The checkpointing process involves storing a set of data (usually a snapshot of execution state) during normal working of the system, from which that execution can be resumed if a failure happens.

A. Replication-Based Fault Tolerance

Providing fault tolerance by using the replication methodology is commonly used by cloud and distributed systems, the replication process involves creating similar copies of the task or data on several resources (backup nodes), when a failure happens, the replica can replace the failed object. Zhang *et al.* [16] proposed to use $3F + 1$ active replicas to handle F simultaneous failures in a cloud computing system. A mixture of active and passive (in suspended state) replicas is used to tolerate F faults in the cloud system [17], they have proposed to make $2F + 1$ replicas, where the active replicas are $F + 1$ while remaining are passive replicas. The replication technique is resource demanding and is affordable in cloud-based systems where unlimited resources are available.

For IoT systems, research works exist that have used the replication technique as fault tolerance, but because of resource limitation, partial replication (in the form of data, process or communication) was used. Research work in [18] used the concept of duplicate service to make a replica of a service running on a smart device to provide a fault-tolerant dynamic IoT environment, they have used the heartbeat protocol to detect and recover from failure without any external intervention. Mohamed *et al.* [19] discussed the fault tolerance and reliability concerns for the fog computing to support smart city applications. Fault tolerance is achieved using a replica of services in the fog, upon failure of a fog node, the services that are processed by this node are replaced by similar service available on another working fog node available in the locality.

A study in [10] offers a fully distributed data replication technique by replicating data hop-by-hop for IoT systems. This guarantees maximum data availability even under a high node failure rate, but in a limited node environment, the number of distinct replicas of data that can be stored in the network decreases. It also suffers the overhead of extra messages transferred between nodes to create replicas on these nodes. Fault tolerance using the redundancy technique is used to meet the high reliability and availability requirement of SAN [20] they have evaluated the reliability of mesh SAN by using binary decision diagrams.

The research work [21] proposed to use an agent-based architecture for IoT by following a hierarchical architecture, to ensure reliability and fault tolerance. They have used the mobile agent to monitor the resources and network. The data replication at the edge of the network helps to provide the reliability, a redirection is performed when a failure happens. The possible level for redirection is among cloud, fog, mist or dew. To handle unexpected faults, the agent will get the priority index for all applications executing on the failed edge node and checks for the available nodes at the same level once it finds the application, migration will be performed and connection rerouting is done. One problem with replication is that it uses extra resources to create replicas, which makes it expensive. For the case of IoT systems, the resources are always limited so relying fully on replication methodology is not sensible.

B. Checkpointing for Fault Tolerance

Checkpointing stores the state of the system periodically as a snapshot at some reliable place. The checkpoint file allows recreating a program after fault detection. The checkpointing-based fault-tolerance protocols normally use three types [22] of checkpointing techniques: 1) coordinated checkpointing; 2) independent or uncoordinated checkpointing; and 3) communication induced checkpointing.

1) *Coordinated Checkpointing:* This technique is also known as global checkpointing. It requires all the system entities to coordinate to construct a globally consistent checkpoint [23], [24].
2) *Uncoordinated or Independent Checkpointing:* To avoid the synchronization overhead, entities construct checkpoint independently of each other. To recover from a fault the overall system restore can be performed using the available checkpoints [7]. Independent checkpointing can result better for the fault tolerance of IoT systems and specifically for the edge computing environment, because each device acts independent of other devices in a decentralized fashion.
3) *Communication-Induced Checkpointing:* This technique allows entities to construct their checkpoint individually (local checkpoint) and force some entities to construct additional checkpoints (forced) to guarantee the eventual process of recovery line [25].

In an edge computing environment, the synchronization overhead to perform coordinated checkpointing becomes very large as the application scales since synchronization is required among all entities. In addition, the nature of edge computing is decentralized where most of the devices operate individually; also, the failure rate of edge nodes is higher than normal distributed computing systems, this can result in higher mean time of using rollback as the recovery process contrary to mean time between failures (MTBFs). Moreover, global rollback cannot guarantee the particular restoration of the system like that of the prefailure state. For the communication-induced checkpointing, forced checkpoints cannot be performed as there is no centralized entity in the edge network to enforce the checkpointing mechanism. Uncoordinated checkpointing can result as better for the IoT applications running as distributed on the edge computing environment, because failed entities (previously executing on particular edge node) can recover individually.

Checkpointing has been used for fault tolerance in distributed computing systems, cloud computing and IoT [4], [7], [9], [11]. In [26], a fault-tolerance technique is presented by using software rejuvenation to handle aging obstacles for applications in the cloud. At first, they have used aging degree

evaluation and adaptive failure detection to predict which service components of cloud primarily deserve to be rejuvenated, latterly, they used checkpoints and trace replay to guarantee the smooth running of a cloud application system.

Some research proposed to build energy-efficient fault-tolerant approach for specific devices in IoT by checkpointing the program execution data to stable storage on the same device [27], [28]. The main point is to store selected states of a program to reduce the time overhead involved in writing checkpoint data to nonvolatile memory (NVM); they have used a well-known algorithm "max-flow min-cut" on the data flow model of the program. Their focus is on fault recovery for a single device. The checkpointing mechanism can be used flexibly to manage fault tolerance in an IoT system.

In the checkpointing-based methodology, the size of the checkpoint file is small, so this method can be used easily for the migration-based fault-tolerance methodology. Research work in [12] proposes an intelligent framework that is based on semantic reasoning. They have proposed to use the checkpointing mechanism to handle more flexibly an IoT system, where gateways are installed to collect data and connect to the cloud for user application. They handled dynamicity, which can occur due to new services or mobility by means of a checkpointing methodology, hence, migration of the software can be performed from one gateway to another gateway. This will help in cases, such as transportation, logistics, or applications where devices need to change their physical position to some other place or city. Additionally, this will also help when the gateway device has limited resources in terms of memory, processing, battery power, and other resources.

Research work in [29] has proposed a solution for handling long running IoT functions for the serverless paradigm by using the execution model of function as a service (FaaS). They have used checkpoint/restore in user space (CRIU) along with the Docker to preserve the function state using checkpointing. They demonstrated to checkpoint and restore sleeping functions using TCP/IP sockets, which is crucial for handling functions having longer execution. Results show that their technique can be used for live migrating of the container state between different devices, as well as to handle offloading and fault tolerance for IoT systems. Meroufel and Belalem [7] provided an evaluation of CRIU by using Docker-based microservices for a single machine. They showed that checkpoint and restore times increased linearly to the image size of the application. The research work [14] handled the live migration to increase the flexibility of the fog computing layer by using Docker and CRIU. They have claimed a 2.1× increase in efficiency using live migration contrary to the without migration-based algorithms. The live migration can face an overhead of increasing the system recovery, especially on lower connections.

To handle the overhead of the checkpointing that comes mainly from the time needed to save the state of a process, different solutions in the literature recommend to combine checkpointing with replication, live migration, or system logs. For example, Sun *et al.* [3] proposed a fault-tolerance model by combining checkpointing along with replication to provide a high serviceability model (HSCR) for a cloud system.

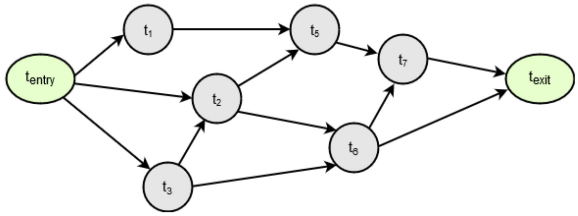


Fig. 2. Simple workflow.

Ozeer *et al.* [8] have used uncoordinated checkpointing along with message logs and keeping records of the function call to handle an IoT applications along with their state in the fog computing environment. They have used checkpointing to save the state of the application and used function calls and messages logs to model interactions of IoT devices and sensors with the physical world.

By using checkpointing along with replication allows faster task execution after a node crash in an edge computing environment. Our methodology uses checkpointing to ensure reliability for IoT application running distributed on edge nodes and replicating these checkpoint files in alternative edge nodes will ensure the availability in a high failure edge environment.

III. Methodology

For this research work, we assume that IoT application is executing the distributed mode on the federation of edge node (group) $G_i^e(W)$ [30] to reduce network traffic and latency. In each group, there exists an organizer node E^{org} where a user submits and IoT job. In our proposed methodology, we have used checkpointing and replication techniques for fault tolerance. A periodic uncoordinated checkpointing is performed for the IoT application tasks running distributed on edge nodes, the resulting checkpoint files are replicated on alternative edge nodes in the group that are currently idle. These alternative edge nodes can be used to replace a failed edge node in the group.

A. Application Model

To execute latency-oriented IoT applications distributed in edge computing environment, workflows can provide a common model for handling a number of applications in distributed systems [31]. The structure of a workflow directs the order of execution for the application tasks. Usually, a workflow W is represented by using the directed acyclic graph (DAG), $W\{T, E\}$ in which T represents the specific computational task which can consists of sub tasks, i.e., $T = \{t_1, t_2, \ldots, t_n\}$ and edge E denotes data or control dependency present between the tasks. Each task t_i is associated with a certain amount of workload wl_i. Fig. 2 represents a simple workflow in the form of a DAG consisting of seven tasks from t_1 to t_7 and two dummy tasks t_{entry} and t_{exit} showing the beginning and end of the workflow, respectively. The DAG also indicates precedence constraints between the tasks in the form of a vertex $V(t_i, t_j)$, which indicates that task t_i must be executed before the task t_j can start.

Fig. 3. Distributed edge computing with neighbor node for increasing replication in the vicinity.

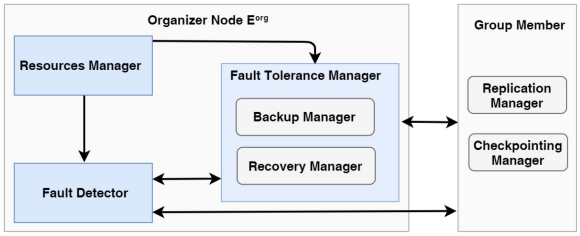


Fig. 4. Proposed system architecture.

B. Execution Model

To execute an application workflow in an edge computing environment, we will use the grouping methodology proposed in our previous work [30]. The distributed execution methodology suggests that edge nodes co-located in the proximity can potentially collaborate to execute a resource-intensive IoT application, as shown in Fig. 3. To manage a big task efficiently in the edge network, a group $G_i^e(W)$ is formed consisting of edge nodes e to complete a workflow W, given that i is the ith group executing a specific workflow. In each group, an edge node is specified as an organizer node (E^{org}), this node receives a request to execute a workflow and will be responsible to collaborate with other edge nodes in vicinity to complete the assigned job and deliver the required results by the user.

C. Reliability Model

When a workflow is executing in the error-prone environment of edge network, the reliability of the task is critically important. The reliability can be defined as a probability to successfully complete a task (t_i). We assume faults in the edge network are independent and uncorrelated for each node. We can model it according to a Poisson distribution with failure rate λ_e. The MTBF is a measure of how reliable the edge node is and can be calculated as

$$\text{MTBF} = \frac{\text{Total time}}{\text{Number of failures}}. \tag{1}$$

The failure rate λ_e is inverse of the MTBF

$$\lambda_e = \frac{1}{\text{MTBF}}. \tag{2}$$

The probability of a node failing r times with in time t is given by

$$p_r(t) = \frac{(\lambda_e t)^r e^{-\lambda_e t}}{r!}. \tag{3}$$

We will use replication to store checkpointing files, so the condition for the task failure is that there is no replica is available for functioning. Here, we are concerned with the probability of working backup, during time t. So, the probability of no failure with in time t is $e^{-\lambda_e t}$, this gives the probability for node failing during time t

$$p(t) = 1 - e^{-\lambda_e t}. \tag{4}$$

If there are n replicas of the task, the cumulative probability that no node will fail during time t will become

$$P(t) = \prod_{i=1}^{n} \left(1 - e^{-\lambda_i t}\right) \tag{5}$$

where λ_i is the failure rate for the ith node. Hence, the reliability for task t_i will become

$$R_{t_i}(t) = 1 - \prod_{i=1}^{n} \left(1 - e^{-\lambda_i t}\right). \tag{6}$$

Considering a task is executing on a specific edge node with its backup stored n number of nodes, the reliability can be increased through replication, if the replicas are placed on separate edge nodes. Furthermore, the failure of each node is independent of each other. To ensure the desired reliability (R_{req}) of the task fault-tolerance management, it has to ensure to replicate the task on n number of separate nodes.

D. System Model

The distributed edge computing environment consists of a group of edge nodes that collaboratively execute IoT application tasks. This group is comprised of two types of devices: 1) organizer node E^{org} where a user submits application, and this node collaborates with other nodes in the edge network to form a group to execute application tasks in distributed and 2) normal edge nodes (members) which provide computing, storage, and communication resources. Fig. 4 provides the architecture of our proposed system. A user submits the specific workflow that is to be executed in the edge computing environment.

The organizer is the node where job is submitted and results are delivered. This node will have the following modules.

1) *Resource Manager* collaborates with the nodes in the edge network to group them and then provide resources needed to execute the tasks. This module will also schedule tasks according to the workflow. A list of nodes belongs to the group (*list_G*) is provided to the fault-tolerance manager after tasks of the workflow are scheduled. This list is also provided to the fault detector.
2) *Fault Detector:* This module will monitor and detect faults in the edge node. Heart beat mechanism is used to detect the failure of a group member.
3) *Fault-Tolerance Manager:* This module has the following submodules.
   a) *Backup Manager* selects alternative edge nodes for each member of the group.

b) *Recovery Manager* will be responsible to ensure recovery in case of failure.

Member edge nodes are the edge nodes, which collaborate with organizer and execute tasks belong to a workflow. Each member will have the following modules.

1) *Checkpoint Manager:* This module will create and store a checkpoint file for the task executing on this edge node.
2) *Replication Manager:* This module will replicate the checkpoint file on alternative edge nodes to increase the reliability.

When an IoT application is submitted to the organizer node in the edge computing environment, the resource manager is activated, this module will group the edge nodes to execute the submitted workflow in distributed, and allocates the subtasks to available edge nodes in the group. A list for nodes present in the group is also provided to fault-tolerance manager. The fault-tolerance manager includes the backup manager and recovery manager, these modules are responsible for fault tolerance for the overall system. The replication manager and checkpointing manager modules are present on each edge node in the group. These modules help to provide reliability to the tasks running on each node in a decentralized way. The working details of each of the module are provided in Section IV.

IV. Fault Tolerance

Our approach for the fault tolerance combines the replication and checkpointing methodology to increase the overall system reliability without increasing the execution load on edge nodes. This section will present the working of each module of the proposed system to achieve fault tolerance for the edge computing environment.

A. Backup Manager

This module will select edge nodes where replication for checkpointing files is performed for each task. In order to minimize checkpointing cost, this module will select those edge nodes in the group which are currently idle. In the edge group, idle nodes can exist due to the dependencies between two tasks in the form of $V(t_i, t_j)$, which mean that the task t_j (here, we will call as child task) has to wait for the execution of the task t_i (here, we will name as parent task). The child has to wait because of some data generated by the parent and will be used as input to the child task, or there might exist some other activity like some interaction with some physical entity is required to accomplish the task. Idle nodes can also exist because of the partial use of its resources like communication resources are in use but processing and storage are idle which can be used to fulfill the requirements.

The backup manager starts working after the organizer node has allocated the tasks to the edge nodes in the group to execute in the distributed edge computing environment. After the task allocation, the estimation for the execution intervals between the start time and finish time of a task is possible. The edge network can consist of several different smart devices, each having different processing capabilities. The speed of an edge e node can be represented using S_e in millions of instructions per second (MIPS). Hence, the ET of the task t_i at a specific edge node e can be calculated as follows:

$$\mathrm{ET}_{t_i}^{e} = \frac{wl_i}{S_e}. \tag{7}$$

For this article, we assume all of the edge nodes are connected by the homogeneous network. The communication delay (DL) for each node and communication bandwidth (BW) between two nodes is the same. Hence, the data transmission time (TT) to send data (d_j^{i}) from task t_i to task t_j will be

$$\mathrm{TT}_{t_i}^{\mathrm{t_j}} = \mathrm{DL} + \frac{d_j^{\mathrm{i}}}{\mathrm{BW}}. \tag{8}$$

Other important parameters related to the task are the early start time ($\mathrm{EST}(t_i)$) and the latest finish time ($\mathrm{LFT}(t_i)$). The $\mathrm{EST}(t_i)$ refers to the time a task can start which is shown in (9), at the bottom of the page, which is determined as when all of its parent tasks (t_h) finish their execution.

Equation (10), shown at the bottom of the page, shows $\mathrm{LFT}(t_i)$ which is latest time a task can finish without missing the deadline, which happens when all the children of the task can finish their execution as late as possible.

The schedule time $ST_{t_i}^{e_i}$ of task t_i is the time for which the task is executing on edge node e_i in the group. This time lies between the calculated start time of a task $\mathrm{CST}(t_i)$) and calculated finish time of a task $\mathrm{CST}(t_i)$). This can take any value between $\mathrm{EST}(t_i)$ and $\mathrm{LFT}(t_i)$. This means the edge node e_i is free for all of the other time between 0 and overall deadline (DL), and we can use this time as a backup for other tasks running on other nodes. This will help to identify candidate nodes for the backup process.

The backup manager will also collaborate with the replication manager on other edge nodes in the group to get a list of neighbor nodes to increase the reliability. This is explained in the next section. The backup manager will keep a list of nodes for the task t_i used as backup nodes, and this list will be provided to the checkpointing manager for backup; the following algorithm explains the working of the backup manager.

$$\mathrm{EST}(t_i) = \begin{cases} 0, & \text{if } t_i = t_{\mathrm{entry}} \\ \min_{t_h \in \mathrm{parents}(t_i)} \left(\mathrm{EST}(t_h) + \mathrm{ET}_{t_h} + \mathrm{TT}_{t_h}^{t_i}\right), & \text{otherwise} \end{cases} \tag{9}$$

$$\mathrm{LFT}(t_i) = \begin{cases} \mathrm{DL}, & \text{if } t_i = t_{\mathrm{exit}} \\ \min_{t_k \in \mathrm{children}(t_i)} \left(\mathrm{LFT}(t_k) + \mathrm{ET}_{t_k} + \mathrm{TT}_{t_i}^{t_k}\right), & \text{otherwise} \end{cases} \tag{10}$$

Algorithm 1 Backup Node Selection Algorithm for Task t_i

Require: list_G; //list of all the nodes belong to group $G_i^e(W)$
Ensure: list_AN; //list of alternative edge nodes to execute t_i in case of failure

```
1: function BACKUP_NODE (t_i, list_G)
2:     List B = φ
3:     List N ← NEIGHBOR_REP(t_i, list_G)
4:     for i ← 1 to |list_G| do
5:         t_a ← GetTask(list_G[i])
6:         if t_a ≠ t_i then
7:             if (Begin (t_i) + ET (t_i) < CST(t_a)) OR
   (CFT(t_a)> Begin (t_i)) then
8:                 B ← B + list_G[i]
9:             end if
10:        end if
11:    end for
12:    for i ← 1 to |N| do
13:        z ← |B|
14:        B[z+1] ← N[i]
15:    end for
16:    for (i ← 1 to |N|) OR (Reliability (t_i) = R_req) do
17:        List_AN[i] ← B[i]
18:    end for
19:    return list_AN
20: end function
```

Algorithm 1 allocates a list of nodes to each subtask (t_i) that may replace the primary node on which t_i is executing. A list (list_G) is provided as input to the algorithm, this list contains all the nodes that belong to the group $G_i^e(W)$. The function Neighbor_Rep(t_i, list_G) which is executed by replication manager at each member of the group returns a list of neighbors to increase replication. The first for loop selects possible alternative nodes from the group that can provide fault tolerance. The second for loop adds the neighbor nodes at the end of the list obtained in the first for loop. The last for loop selects the desired n number of edge nodes to satisfy the required reliability level R_{req} for task t_i.

B. Replication Manager

To increase the reliability of task (t_i), one has to increase replicas for the task t_i. For every node (g) in the group $G_i^e(W)$, g will preserve a subset of selected nodes from neighbors. It will provide decentralization in the fault-tolerance process, as each node is selecting neighbor nodes for the backup process independently. To ensure the reliability, this set of neighbors should contain edge nodes capable of completing the significant task for any failure event. When selecting an edge node q for the set of neighbor nodes, that node $g \in G_i^e(W)$ has to fulfill the following.

1) The neighbor selection should check the exhausted nodes (in terms of battery and other resources). Such a node should avoided to become a backup.
2) Neighbors that are active nodes and already providing resources to the same workflow as that of g should not be selected in the subset.

Algorithm 2 Neighbor Selection Algorithm to Increase Replication of the Task t_i Executing on g

Require: list_G; //list of all the nodes belong to a group $G_i^e(W)$
Ensure: list_N; //contains a subset of neighbors for node g executing t_i

```
1: function NEIGHBOR_REP (t_i, list_G)
2:     List N = φ
3:     List N' = φ
4:     g' ← Get_Node(t_i)
5:     N' ← neighbors of g'
6:     for i ← 1 to |N'| do
7:         if (N'[i] ∉ list_G) OR (N'[i] ≠ exhausted_node)
   then
8:             N ← N'[i]
9:             N'[i] ← N' − N'[i]
10:        end if
11:    end for
12:    return list_N
13: end function
```

The following algorithm provides a list of neighbors based on the above criteria to increase reliability of the task and ensure the overall availability of the workflow.

Algorithm 2 takes a list of group nodes as input and returns a subset of neighbors for each of the group members to increase the replication. At line 4, the algorithm gets the node on which the respective task t_i is executing, and then all of the neighbors of the node are selected. The for loop iterates over each neighbor node and using if conditions select only those nodes which are not a part of the same group, as well as those nodes which ensure the condition that nodes are not exhausted nodes. The list_N containing a subset of neighbor nodes is returned at the end of the algorithm.

C. Checkpointing Manager

Checkpointing functionality can be used to provide fault-tolerant systems. The distributed edge computing environment can benefit from checkpointing, where the checkpoint image of a task could be stored on alternative edge nodes and in the case of failure, the task can be retrieved. Algorithm 3 ensures the checkpointing process at each node g of the group $G_i^e(W)$ (which is executing the overall workflow in distributed). The node g is executing a specific task (t_i) that belongs to the workflow. The checkpointing process uses incremental checkpointing and stores the checkpoint file asynchronously on each of the nodes provided in the list_AN. This list is provided by the backup manager.

Algorithm 3 takes the list of alternative nodes as input to store the copy of the checkpoint file to increase the task availability. This algorithm executes until the task is executing on the group member node. The condition tests for a checkpoint time interval, once the interval expires, the ith checkpoint is performed for the task t_i. The for loop replicates the checkpoint file on the alternative edge nodes.

Algorithm 3 Algorithm for Making Checkpoint and Replicating the Checkpoint File

Require: list_AN; //list of alternative edge nodes
Ensure: Provide fault tolerance for task t_i

```
1: function CHECKPOINT_REP (t_i, list_AN)
2:     i ← 1
3:     while t_i is executing do
4:         if T_curr − T_last_chkpt = T_interval then
5:             CP_i^{t_i} ← Checkpoint (t_i)
6:             i ← i + 1
7:             for j ← 1 to |list_AN| do
8:                 Copy(CP_i^{t_i}, list_AN[j])
9:             end for
10:        end if
11:    end while
12: end function
```

Algorithm 4 Algorithm for Recovery of a Failed Task

Require: list_AN; //list of alternative edge nodes
Ensure: Recover a failed task t_i

```
1: function RECOVER (t_i, list_AN)
2:     g ← Get_Node(t_i) //node where t_i is executing
3:     g′ ← list_AN[1]
4:     g′ ← Assign (t_i)
5:     Resume (t_i, CP_i^{t_i})
6:     list_G ← list_G - g
7:     if (g′ ∉ list_G) then
8:         list_G ← g′
9:     end if
10:    for j ← i to |list_T| do
11:        Update (list_AN_j)
12:    end for
13: end function
```

D. Recovery Manager

The recovery manager is responsible for the recovery procedure for the task t_i which belongs to the workflow and is currently executing on a group member g. The recovery process is triggered after the failure is detected for the node executing t_i. The task is restarted from the last checkpoint stored in an alternative edge node. This checkpoint file is already stored in the alternative edge nodes by the checkpointing manager. The first node present in list_AN is selected to execute the failed task and this node will also be added to the group if already not a part of the group (the node belongs to list_N), additionally, the list_AN for remaining tasks is also updated. Algorithm 4 explains the working of the recovery manager in case of a node $g \in G_i^e(W)$ fails while executing the task t_i of the workflow.

Algorithm 4 recovers the failed task t_i executing on a group member g. When edge node g is failed, the first alternative node from list_AN is selected to resume the execution of t_i. This alternative node g is returned once the task is restarted. The last for loop updates the list of alternative nodes for the remaining tasks in the workflow.

Algorithm 5 Ensure Workflow Availability and Handling Node Failure

Require: list_G; //list of all the nodes belong to group G
Require: list_T; //list of the all tasks belongs to the workflow
Ensure: Fault tolerance to the distributed edge computing environment

```
1: function FAULT_TOLERANCE ( )
2:     for i ← 1 to |list_T| do
3:         t_i ← list_T[i]
4:         list_AN_i ← BACKUP_NODE(t_i, list_G)
5:         CHECKPOINT_REP (t_i, list_AN_i)
6:         if (t_i = Fail) then
7:             RECOVER (t_i, list_AN_i)
8:         end if
9:     end for
10: end function
```

We have explained the working of submodules and the following algorithm explains how the overall reliability is provided and how the node failure is handled in distributed edge computing.

Algorithm 5 provides the system for fault tolerance in the distributed edge computing environment. The for loop iterates through each task of the workflow to allocate the backup nodes and start checkpointing and replication process for every task. When a task failure is detected, it provides the recovery for that task.

V. EVALUATION

The key purpose of the evaluation is to validate the effectiveness of the proposed system, furthermore, ensuring the IoT application latency requirements, network load reduction, efficient energy consumption, and increasing the utilization of available devices in the edge computing environment. We have performed various experiments using the simulations by simulating the problems in the real-world applications. We use iFogsim [32] for this purpose, this simulator allows to simulate the different application scenarios under different configurations of the fog/edge computing network, additionally, it allows to measure different statistics, such as energy consumption, network traffic, and application delay.

A. Experimental Setup

We have simulated video surveillance for object tracking, which relies on the number of cameras generating video streams and some processing modules [32]. Video surveillance where a distributed system of camera surveilling an area that can be used for a lot of application areas, such as public safety, security, healthcare, and manufacturing. The key system requirements include low latency, reduce network traffic, high-end processing of the overall application, and, moreover, application has to ensure availability and reliability. As the application also resource intensive and requires low-latency executions, hence, the distributed edge computing environment provided by our edge node grouping methodology [30] is the best option to run the application.

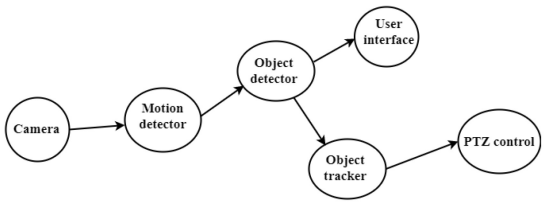


Fig. 5. Workflow for the surveillance application.

TABLE I
MODULE PROCESSING REQUIREMENTS

Module	CPU (MIPS)
Camera	100
Motion Detector	1000
Object Detector	800
User Interface	150
Object Tracker	500
PTZ Control	300

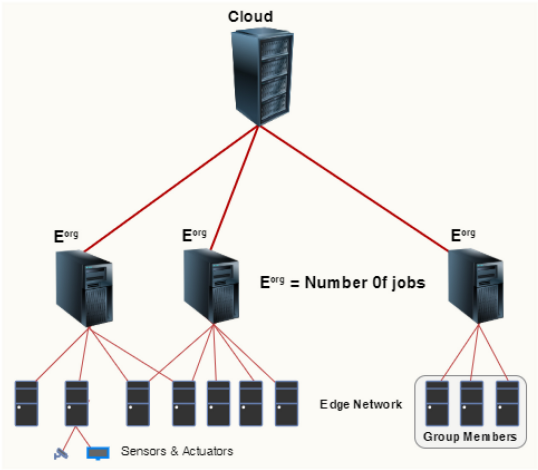


Fig. 6. Simulation topology with cloud, organized nodes, and group members in edge network.

The surveillance application has six modules represented using the workflow shown in Fig. 5. The workflow consists of modules (tasks), including camera, motion detector, object detector, object tracker, user interface, and pan, tilt, and zoom (PTZ) control. The video streams from cameras are forwarded to the motion detector module, which filters those videos where motion is detected and forward these selected videos to the object detector module. The key task of the object detector module is to identify moving objects; once object identification is performed, it sends object identification information along with its position to the object tracker module. The object tracker computes desirable PTZ settings and sends required instructions to PTZ control. The application also includes a user interface module that displays a segment of the video streams containing each of the tracked object to the user device. The processing requirements of each module are given in Table I.

Fig. 6 shows the overall network topology used to simulate the above application workflow. We have tested for three different configurations by varying the number of workflows for each configuration, where each workflow represents a surveilled area. The number of workflows used are 3, 6, and 9 for configuration 1, 2, and 3, respectively. For each workflow, an organizer node is specified, which will group the edge nodes in the area, the organizer is also responsible for providing fault tolerance for nodes executing the workflow according to the proposed methodology. A total of ten organizer nodes (E^{org}) are placed in the edge network with 60 devices are added acting as normal edge nodes that can be used as group members in the topology. Table I lists the network latencies between devices in the topology, and device-related specifications are present in Tables II and III.

TABLE II
NETWORK LATENCIES

Network	Latency (ms)
Sensors&Actuators – Edge node	2
Edge node – Organized node	5
Organizer node – Cloud	100

TABLE III
DEVICES CONFIGURATIONS OF THE TOPOLOGY

Devices in topology	Upstream link (Mbps)	Downstream link (Mbps)	RAM (MB)	CPU (MIPS)
Cloud	1000	10000	40960	40000
Organized	10000	10000	4096	5000
Edge nodes	10000	10000	2048	500 – 1000
Sensors & Actuators	10000	10000	512	500

When the workflow arrives at organizer node (E^{org}), its resource manager performs the group formation and assigns the modules to the edge nodes. The fault-tolerance module coordinates with group members to ensure the application execution in the presence of failures. The objective of the evaluation is to analyze the performance of the fault-tolerance methodology regarding delay, task deadline missing ratio, cost effectiveness in terms of energy consumption and effect of failure rate. The different result comparisons are performed as follows.

1) *Adaptive Checkpointing and Replication (ACR):* It is our proposed methodology for fault tolerance in the edge computing environment; we represent our methodology as ACR.
2) *RM:* It is the replication-based decentralized fault-tolerance methodology for IoT applications executing on a group of edge nodes presented in [33]. This methodology selects a set of neighbor nodes for every node g present in the group, and replicated task executing at g to the nodes present in the set. When any group member fails, the first node in the set replaces the failed node to recover the failed task. We name this Rep-FT.
3) *No-FT:* If no fault tolerance is provided in the edge network, failed tasks will be transferred to the cloud to complete the application execution.

B. Results

1) Application Latency: Application latency is used to measure the efficiency of IoT application; the edge computing environment provides low-latency results. Specifically, the application latency will be low when some fault-tolerance

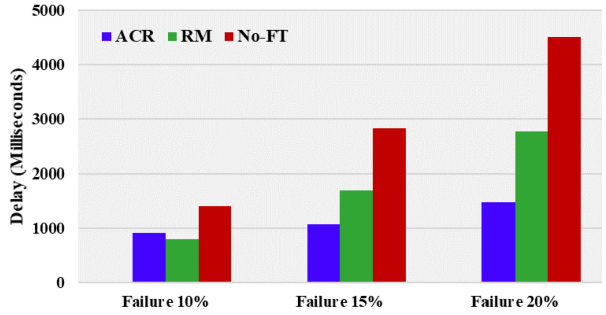


Fig. 7. Application latency comparison.

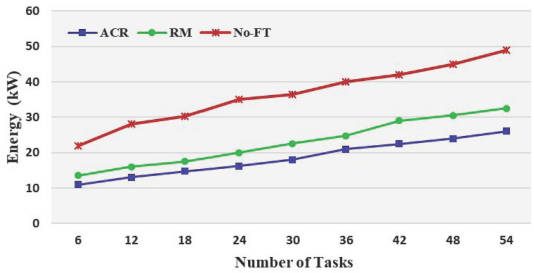


Fig. 8. Energy consumption for workload.

mechanism is active in the edge computing environment. Fig. 7 shows the application latency for three methodologies against different numbers of the failure rate percentages. For a lower failure rate, the RM performs better because more nodes are available in the edge network for replication, and the replication-based fault-tolerance techniques are faster in the recovery process when compared with checkpoint-based techniques. On the contrary, when the failure rate is increased, our proposed methodology performs better than other two approaches. For the higher failure rate, there will be fewer nodes available to perform replication in the RM approach, and if there is no replica available in the edge network, the task failure will result in more delay. Our methodology is a resource-efficient one, which can provide fault tolerance in a resource-limited edge network by using the free slots of available edge nodes for a recovery process.

2) Energy Consumption: The results in Fig. 8 show the energy consumption for a failure rate of 10%. The behavior for energy consumption is identical for the three approaches, i.e., energy increases by increasing workload in the form of subtasks. At start, the energy consumption level is smaller, especially when fault tolerance is provided in the edge network (ACR and RM), but over the time, by adding more workflows to the system, the resources consume more energy to execute the tasks and ensure fault tolerance. If no fault tolerance is provided, the No-FT approach consumes more power because the failed tasks are directed toward the cloud to complete their execution. Our approach reduces over all energy consumption, as it uses free time slot available on the members nodes of the group to execute the failed tasks, so the recovery is done immediately. The RM technique consumes more power because it uses extra nodes to replicate tasks. The results show the cost effectiveness of our approach in terms of energy consumption.

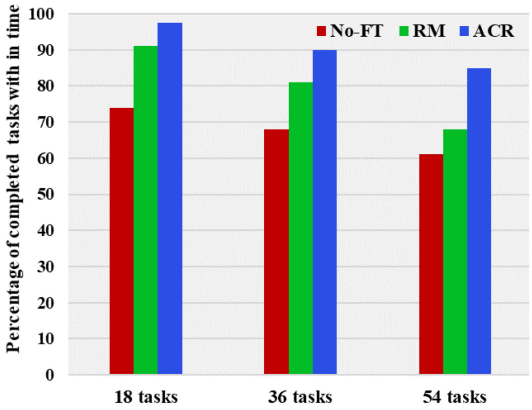


Fig. 9. Completed task comparison.

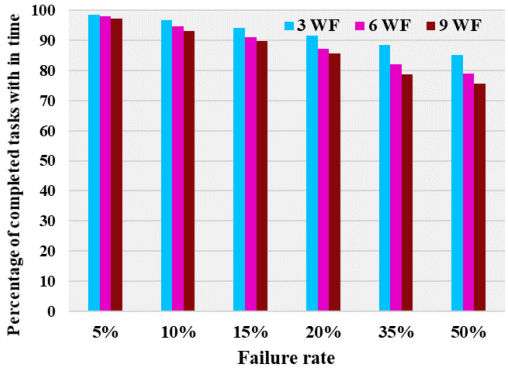


Fig. 10. Performance under different failure rate and flow size.

3) Success Rate: Success rate is an important factor used to measure reliability of the fault-tolerant systems. A reliable system has higher success rate, we have measured the success rate by using a ratio of completed tasks in a workflow with in the required time to the total number of tasks submitted to the system during a certain period

$$\text{Success rate} = \frac{\text{Number of tasks completed}}{\text{Total number of tasks}}. \tag{11}$$

Fig. 9 shows the comparison of three techniques in terms of success percentage for different number of tasks submitted to the system. It is evident that the success rate for the ACR methodology is higher than other two methods when the failure rate is higher even when there are more number of tasks (more workflows) submitted. This is because our methodology provides better response time by using smart checkpointing methodology and mostly recovery is performed by the group member nodes. On the contrary, when more tasks are submitted, it decreases the number of replicas for the RM methodology, which affects the success rate. For the Cloud methodology, the number of completed tasks within due time is always low because of network latency is present.

Fig. 10 shows the performance of ACR against a different number of workflows (WF) under different failure rates, we have calculated success percentage of different number of tasks completing their dead line submitted to the system. The results depict that for a low failure rate, most of the tasks have achieved their deadline, this was because a lot of free

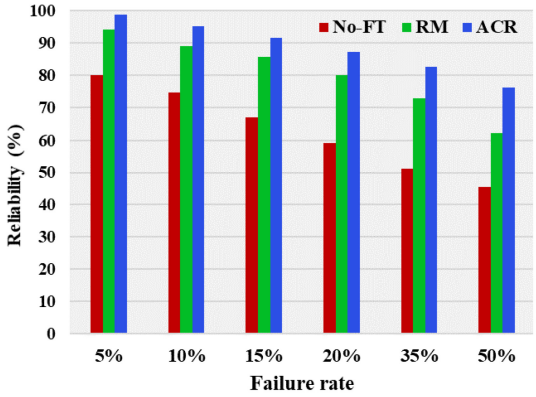


Fig. 11. Reliability versus failure rate.

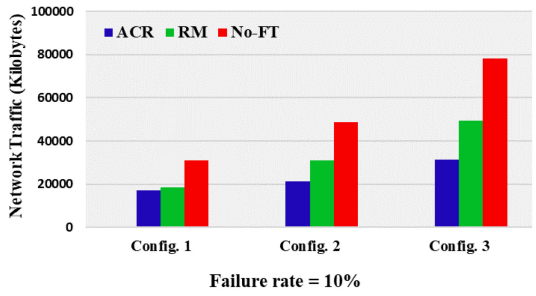


Fig. 12. Network traffic generated for a lower failure rate.

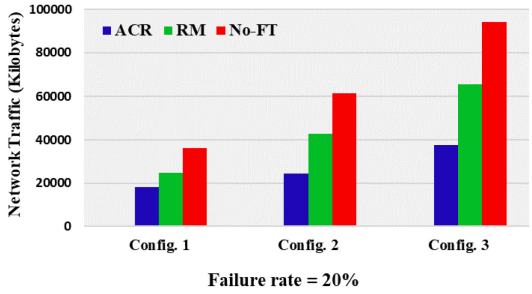


Fig. 13. Network traffic generated for a higher failure rate.

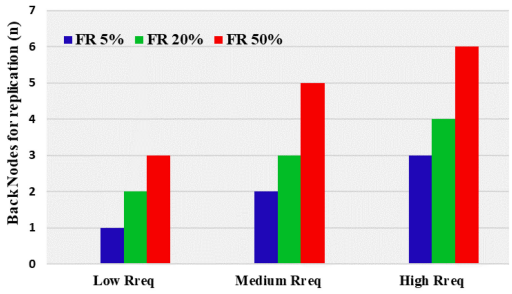


Fig. 14. Replication attained against different reliability and failure rates.

resources exist in the edge network to provide the reliability during the smooth working of nodes. When the number of tasks increases in the case of more workflows, the system, and a lower failure rate, the fault-tolerance system is able to complete the tasks ensuring deadlines, and even for an average failure rate. For a higher failure rate and more workload, the system does not fail rather it shows good performance to achieve a good percentage in terms of task completion.

4) Reliability: Fig. 11 shows results for measuring the reliability for the three techniques in case of increased failure rate. The No-FT is more vulnerable to increased failure rate as the task execution and its data will be lost if edge node failure occur, making it impossible to perform a recovery for the failed task. By using redundancy in form of replication in the RM performs better; however, the reliability of the task depends on the factor that currently the system has how many replicas. Our fault-tolerance methodology uses replication of the checkpoint files to the other nodes present in the group executing the workflow. In addition, our methodology also uses the neighbor replication mechanism, which will ensure minimum required replicas to fulfill the application reliability requirements.

5) Network Traffic: Providing fault tolerance for the edge network impacts on the total network use, a staggering decrease is observed in the data transferred over the network. We have compared three configurations of workflows, and by using a failure rate of 10% and 20%. Fig. 12 shows results for the lower failure rate and less number of tasks, the data generated for both fault-tolerance techniques are same, but when the number of tasks increases, the RM technique shows higher network traffic as more replicas are generated. The accumulated data transfer in the network results to be far less for the ACR methodology even for the higher failure rate Fig. 13, because the checkpoint file size is less as compared to complete process replication on the number of edge nodes in the RM technique. It is also observed when the failure rate is increased, network traffic for the RM technique rises, as more replicas are required to provide the required reliability level. This shows that our methodology is also effective in terms of network utilization for higher failure rates. Moreover, if there is no fault tolerance provided in the edge network, the No-FT approach, the execution for failed tasks are performed on the cloud, this increases the network traffic.

6) Reliability and Replication: Fig. 14 shows replication achieved against different failure rates and different required reliability R_{req} levels. It is evident that for lower R_{req} fewer replicas were present in the edge network, this helped to achieve resource efficiency along with ensuring reliability. For the case of medium and high R_{req}, less replicas are created by our methodology, specifically when there is a lower failure rate; however, for high failure rates, more replicas are created to provide efficient fault tolerance to the system and even more replicas are created when the required reliability is set to high. The more replicas for a higher failure rate will help to complete the tasks inside the edge network, resulting in meeting the deadlines. But more replicas will result in resource consumption, one has to tradeoff resource utilization to achieve higher system reliability.

VI. Conclusion and Future Work

To obtain full advantage of the edge computing environment, an efficient and cost effective fault-tolerant system was required. In this article, we have proposed to combine two

fault-tolerance techniques to minimize overhead on the edge nodes. We have used checkpointing of tasks in execution to reduce the burden of backup on resource-limited edge devices, and by replicating these checkpoint files on the selected alternative nodes to improve the overall application availability. The alternative nodes are selected based on free resources as well as the free nodes available in the vicinity. This will ensure the application reliability requirements. The performance of the proposed methodology was evaluated by comparing with pure replication-based technique, where process and data both are replicated on neighbor nodes, in contrary to our approach, where only checkpoint file is replicated. Experimental results indicated that our fault-tolerance methodology improves the overall performance of IoT application when executed in the distributed edge computing environment. The outcomes reveal that the end user can enjoy the service quality concerning application response time and overall system cost.

The edge computing has already attracted the research community and IoT industry, and it is predicted to be the major driving force for the latency-oriented IoT application. In order to cope with future requirements, a reliable and optimized edge computing system is required that can dexterously handle both processing and communication. The fault tolerance in the edge computing environment is still in its embryonic stage, more extended research is required to error free and smooth execution of real-time applications in the edge computing environment. Moreover, the real test scenario for fault tolerance is required to be implemented on real edge nodes.

REFERENCES

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[2] H. El-Sayed *et al.*, "Edge of things: The big picture on the integration of edge, IoT and the cloud in a distributed computing environment," *IEEE Access*, vol. 6, pp. 1706–1717, 2017.

[3] D. Sun, G. Chang, C. Miao, and X. Wang, "Building a high serviceability model by checkpointing and replication strategy in cloud computing environments," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst. Workshops*, Macau, China, 2012, pp. 578–587.

[4] W. Chen and J. TSA, "Fault-tolerance implementation in typical distributed stream processing systems," *J. Inf. Sci. Eng.*, vol. 30, no. 4, pp. 1167–1186, 2014.

[5] A. Sari and B. Necat, "Securing mobile ad-hoc networks against jamming attacks through unified security mechanism," *Int. J. Ad Hoc Sens. Ubiquitous Comput.*, vol. 3, no. 3, p. 79, 2012.

[6] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, pp. 1–44, 2008.

[7] B. Meroufel and G. Belalem, "Adaptive time-based coordinated checkpointing for cloud computing workflows," *Scalable Comput. Pract. Exp.*, vol. 15, no. 2, pp. 153–168, 2014.

[8] U. Ozeer, X. Etchevers, L. Letondeur, F.-G. Ottogalli, G. Salaün, and J.-M. Vincent, "Resilience of stateful IoT applications in a dynamic fog environment," in *Proc. 15th EAI Int. Conf. Mobile Ubiquitous Syst. Comput. Netw. Serv.*, 2018, pp. 332–341.

[9] B. Meroufel and G. Belalem, "Optimization of checkpointing/recovery strategy in cloud computing with adaptive storage management," *Concurrency Comput. Pract. Exp.*, vol. 30, no. 24, p. e4906, 2018.

[10] W. B. Qaim and O. Ozkasap, "DRAW: Data replication for enhanced data availability in IoT-based sensor systems," in *Proc. IEEE 16th Int. Conf. Depend. Auton. Secure Comput. 16th Int. Conf. Pervasive Intell. Comput. 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Athens, Greece, 2018, pp. 770–775.

[11] S. Cherrier, Y. M. Ghamri-Doudane, S. Lohier, and G. Roussel, "Fault-recovery and coherence in Internet of Things choreographies," in *Proc. IEEE World Forum Internet Things (WF-IoT)*, Seoul, South Korea, 2014, pp. 532–537.

[12] F. Aïssaoui, G. Cooperman, T. Monteil, and S. Tazi, "Intelligent checkpointing strategies for IoT system management," in *Proc. IEEE 5th Int. Conf. Future Internet Things Cloud (FiCloud)*, Prague, Czech Republic, 2017, pp. 305–312.

[13] Z. Ghodsi, S. Garg, and R. Karri, "Optimal checkpointing for secure intermittently-powered IoT devices," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, Irvine, CA, USA, 2017, pp. 376–383.

[14] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *Proc. 5th Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Daegu, South Korea, 2016, pp. 1–6.

[15] A. Sari and M. Akkaya, "Fault tolerance mechanisms in distributed systems," *Int. J. Commun. Netw. Syst. Sci.*, vol. 8, no. 12, p. 471, 2015.

[16] Y. Zhang, Z. Zheng, and M. R. Lyu, "BFTCloud: A Byzantine fault tolerance framework for voluntary-resource cloud computing," in *Proc. IEEE 4th Int. Conf. Cloud Comput.*, Washington, DC, USA, 2011, pp. 444–451.

[17] Y. Tan, D. Luo, and J. Wang, "CC-VIT: Virtualization intrusion tolerance based on cloud computing," in *Proc. 2nd Int. Conf. Inf. Eng. Comput. Sci.*, Wuhan, China, 2010, pp. 1–6.

[18] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin, and Y.-C. Wang, "Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware," in *Proc. IEEE World Forum Internet Things (WF-IoT)*, Seoul, South Korea, 2014, pp. 45–50.

[19] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Towards fault tolerant fog computing for IoT-based smart city applications," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Las Vegas, NV, USA, 2019, pp. 0752–0757.

[20] L. Xing, M. Tannous, V. M. Vokkarane, H. Wang, and J. Guo, "Reliability modeling of mesh storage area networks for Internet of Things," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 2047–2057, Dec. 2017.

[21] J. Grover and R. M. Garimella, "Reliable and fault-tolerant IoT-edge architecture," in *Proc. IEEE SENSORS*, New Delhi, India, 2018, pp. 1–4.

[22] A. Khunteta and K. Praveen, "An analysis of checkpointing algorithms for distributed mobile systems," *Int. J. Comput. Sci. Eng.*, vol. 2, pp. 1314–1326, Jul. 2010.

[23] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Int. Workshop Parallel Distrib. Algorithms*, 1988, pp. 215–226.

[24] B. Meroufel and G. Belalem, "Service to fault tolerance in cloud computing environment," *WSEAS Trans. Comput.*, vol. 14, no. 1, pp. 782–791, 2015.

[25] D. Manivannan, Q. Jiang, J. Yang, and M. Singhal, "A quasi-synchronous checkpointing algorithm that prevents contention for stable storage," *Inf. Sci.*, vol. 178, no. 15, pp. 3110–3117, 2008.

[26] J. Liu, J. Zhou, and R. Buyya, "Software rejuvenation based fault tolerance scheme for cloud applications," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, New York, NY, USA, 2015, pp. 1115–1118.

[27] T. Xu and M. Potkonjak, "Energy-efficient fault tolerance approach for Internet of Things applications," in *Proc. 35th Int. Conf. Comput.-Aided Des.*, Austin, TX, USA, 2016, pp. 1–8.

[28] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, Cambridge, MA, USA, 2013, pp. 29–40.

[29] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of IoT edge functions," in *Proc. 2nd Int. Workshop Edge Syst. Anal. Netw.*, 2019, pp. 60–65.

[30] M. Mudassar, Y. Zhai, L. Liao, M. N. Zahid, and F. Afzal, "Decentralized and secure cooperative edge node grouping to process IoT applications in heterogeneous smart cyber-physical systems," in *Proc. 17th Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST)*, Islamabad, Pakistan, 2020, pp. 395–400.

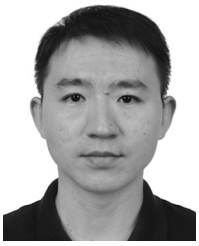
[31] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Gener. Comput. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.

[32] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments," *Softw. Pract. Exp.*, vol. 47, no. 9, pp. 1275–1296, 2017.

[33] M. Mudassar, Y. Zhai, L. Liao, and J. Shen, "A decentralized latency-aware task allocation and group formation approach with fault tolerance for IoT applications," *IEEE Access*, vol. 8, pp. 49212–49223, 2020.

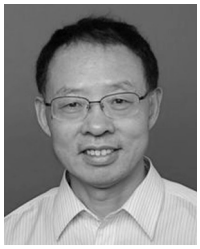
Muhammad Mudassar received the master's degree from the School of Electrical Engineering and Computer Science, SEECS-NUST, Islamabad, Pakistan, in 2011, and the Ph.D. degree in computer science from Beijing Institute of Technology, Beijing, China, in 2020.

He is working as a Lecturer with the Department of Computer Science, COMSATS University Islamabad (Vehari Campus), Vehari, Pakistan. His research interests include distributed computing, big data processing frameworks, fog computing, fault tolerance, and edge computing.



Yanlong Zhai (Member, IEEE) received the B.Eng. and Ph.D. degrees in computer science from Beijing Institute of Technology, Beijing, China, in 2004 and 2010, respectively.

He is an Associate Professor with the School of Computer Science, Beijing Institute of Technology. He was a Visiting Scholar with the Department of Electrical Engineering and Computer Science, University of California at Irvine, Irvine, CA, USA. His research interests include cloud computing, big data, and edge computing.



Liao Lejian (Member, IEEE) received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 1994.

He is currently a Professor with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, where he also served as the Vice Dean of the School. He has published numerous papers in several areas of computer science. His main research interests are machine learning, natural language processing, and intelligent network.