Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 12, 2025

SYNCHRONIZATION STRATEGIES FOR IOT AND EDGE INTELLIGENCE APPLICATIONS

Fauzia Talpur[*1], Mir Rahib Hussain Talpur[2], Shakir Hussain Talpur[3], Mishal Saima[4], Mir Sajjad Hussain Talpur[5], Khan Muhammad Maher[5]

[*1]*Department of Computer Science, University of Sindh Jamshoro Laar campus*
[2]*Information Technology Centre, Sindh Agriculture University Tandojam*

DOI: https://doi.org/10.5281/zenodo.19199684

Keywords

Article History
Received: 15 October 2025
Accepted: 02 November2025
Published: 14 December 2025

Corresponding Author: *
Fauzia Talpur

Abstract

The sentence we provided talks about the challenges and importance of synchronization in edge-based IoT systems, particularly in the context of new Edge AI applications. It explains that simply synchronizing clocks is not enough and that a synchronization system needs to consider factors like movement, network disconnections, and other related problems. This research proposes three task-based strategies and two redundancy-based strategies to address the synchronization problem. The main idea is to group devices into synchronized teams and identify synchronization points using a theoretical solution. The method uses a hierarchical design with controllers at different levels and devices at the leaf level. A late notification protocol is used by the device cluster to select the best synchronization point from pre-calculated options for time-aligned task execution. The suggested "fast synchronizer" solution improves performance compared to existing alternatives. The research focuses on creating a quick synchronization method using simulations based on real-world data. The algorithms' performance will be evaluated and compared to existing solutions, considering fault tolerance to ensure reliability. The findings and outcomes of the research will be thoroughly discussed, and suggestions will be made based on the results. The research proposes a synchronization system that addresses these issues because clock synchronization alone is not sufficient for such systems. It describes three task-based solutions and two redundancy-based strategies to achieve synchronization in edge-based IoT. Additionally, it suggests a hierarchical design with controllers at different levels and devices at the leaf level to coordinate IoT operations involving drones and the Internet of Vehicles. The primary goal of the "rapid synchronizer" is to group devices into synchronized teams and determine synchronization locations using a theoretical solution. A late notification protocol is used to quickly select the best synchronization point for time-aligned task execution from pre-calculated options. The research demonstrates that the fast synchronizer outperforms other solutions and offers significant performance benefits. The study aims to address synchronization challenges in edge-based IoT systems, specifically in Edge AI applications, with the aim of improving convergence, accuracy, and speed in distributed training processes while considering fault tolerance. A rapid synchronization approach is being developed and evaluated for its performance.

INTRODUCTION

The Internet of Things (IoT) is a platform that allows devices to become smarter and more efficient every day. This system connects physical and digital environments and has already made amazing progress as a global media solution. One of the challenges in IoT is the coordination of multiple devices that perform different functions such as detecting, verifying, triggering, interacting, and managing. Communication is also an important aspect of IoT. Recently, SpaceX acquired an IoT satellite strategy expert, and this has taken the competition for universal satellite connection for IoT devices to a new level.

As more people use the internet, the use of larger IoT devices that are managed by edge or cloud resident coordinators will be a crucial trend for developing IoT systems and smart systems. Such devices offer low latency and localizing services, while cloud-based devices offer a global perspective. However, the coordination of these devices in the presence of disconnections is a challenge. A synchronization system is required to coordinate the operations of several devices due to the enormous number of devices and the volume of data being gathered.

In order to accomplish collecting and processing data within predetermined proper time frames, on-the-edge technology is necessary, extraction and device activation. An essential issue that must be resolved at various phases of an application's development is device synchronization. The synchronization is the key to guaranteeing convergence, accuracy, and speed of distributed training methods over numerous edge machines.

Overall, device synchronization and communication are important aspects of IoT. To achieve synchronization, a variety of techniques are used including the energy-efficient reference broadcast synchronization algorithm, which aims to reduce energy consumption for nodes while achieving accurate synchronization.

1.2 Problem statement

Environmental rules that are becoming more stringent and the continued population growth are driving a desire for smarter cities, smarter buildings, and smarter transportation systems. As a result, devices managed by edge or cloud resident coordinators are emerging as a key trend in the development of sufficiently intelligent IoT systems. By way of all knows, likewise system, clock synchronization is required but not entirely adequate (Ameya 2020). A method of synchronizing an edge based Network disconnections, errors, failures, and mobility are all problems that an edge-based IoT system must be able to address system (Sun et al., 2012). As in the modern era of technology, the Applications that operate in real time that are mapped to edge computing must carry out data collection, Within a set of acceptable time limitations, processing of data, knowledge mining, & device actuating are required. For instance, drones hauling a weight must apply force simultaneously so the work to be sums up and operate the errands simultaneously synchronization primitives are required so that jobs might be launched at IoT devices with coordinated start timings clock synchronization, where IoT devices share a similar understanding of time, has been the subject of earlier efforts on synchronization in the IoT. That is, IoT devices keep their clocks as accurate as possible.

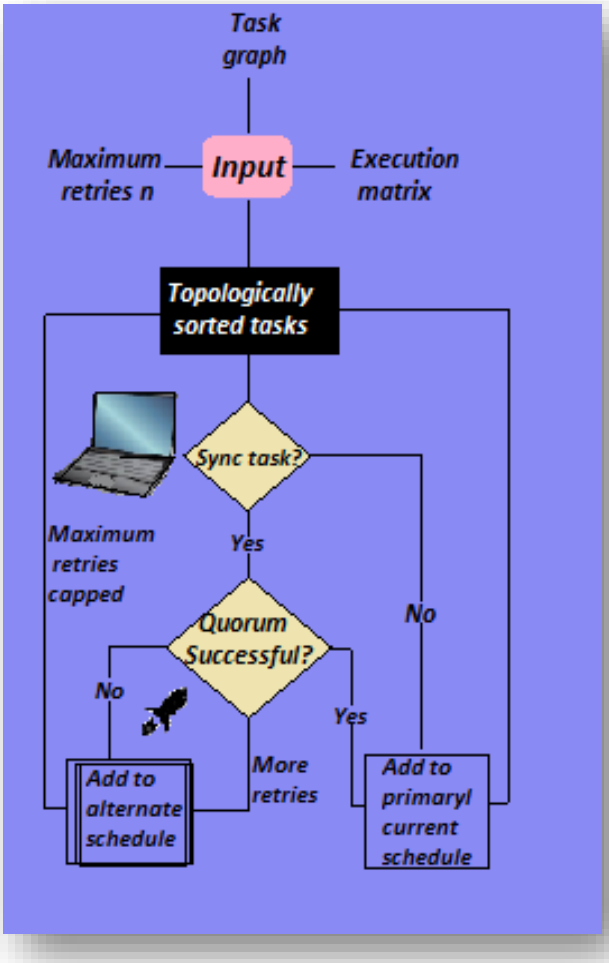

Figure 1.1 Process flow

The following are the contributions of this paper:

1.3 Background of synchronization strategies

Synchronization refers to the coordination of activities or processes to ensure that they occur at the same time or in a specified order. In computer science, synchronization usually refers to the coordination of concurrent processes or threads to prevent conflicts and ensure correct program behavior. Synchronization can be achieved using various mechanisms such as locks, semaphores, monitors, and barriers. These mechanisms provide a way for threads or processes to communicate with each other and coordinate their actions. For example, in a multi-threaded program, accessing a shared resource by many threads at once might result in race situations and unpredictable behavior. To ensure that only one thread uses the resource at a time, synchronizations methods like locks can be employed, avoiding conflicts and guaranteeing proper programmer behavior. In relation to edge intelligence and the IoT synchronization strategies play a crucial part in making sure that distributed devices and sensors can work together effectively. Here are some synchronization strategies that are commonly used in these domains.

Time synchronization

Time synchronization is critical for ensuring that devices and sensors can communicate and coordinate their actions effectively. In IoT and edge intelligence applications, devices may be distributed

across different time zones and geographies. Time synchronization protocols such as the Network Time Protocol (NTP) or Precision Time Protocol (PTP) can be used to ensure that devices are all operating on the same clock.

Data synchronization

Data synchronization is important for ensuring that data collected by different devices and sensors can be combined and analyzed effectively. In IoT and edge intelligence applications, data may be collected at different rates and in different formats. Synchronization mechanisms such as message queues, publish-subscribe systems, and distributed databases can be used to ensure that data is collected and processed in a consistent and timely manner.

Load balancing

Load balancing is important for ensuring that distributed devices and sensors can work together effectively without overwhelming any one device or sensor. Load balancing mechanisms can be used to distribute workloads across different devices and sensors, ensuring that each device or sensor is operating at an optimal level.

Resource management

Resource management is important for ensuring that distributed devices and sensors can operate within the constraints of their available resources. Synchronization mechanisms such as resource allocation policies, memory management, and power management can be used to ensure that resources are allocated effectively and efficiently across distributed devices and sensors.

Security synchronization

Security synchronization is critical for ensuring that distributed devices and sensors can communicate and exchange data securely. Synchronization mechanisms such as encryption and authentication can be used to ensure that data is exchanged securely between distributed devices and sensors. These are just a few examples of the synchronization strategies that are commonly used in IoT and edge intelligence applications. The selection of synchronization strategy depend on specific necessities of application, the underlying hardware and software platform, and the nature of the distributed devices and sensors involved.

Figure 1.2 Showing the concept of strategic synchronization

1.4 Role of synchronization in iot and edge intelligence applications

The IoT and edge-based applications rely heavily on synchronization to ensure that data and devices are working together in a coordinated and efficient manner. Here are some of the key roles that synchronization roles in edge-based and IoT applications. The Data Collection: In edge-based

applications and IoT, data is typically collected from distributed sensors and devices. Synchronization mechanisms ensure that data is collected consistently and at the appropriate time from all devices. This ensures that the data can be combined and analyzed in a meaningful way, The Data Processing: Once data is collected, it needs to be processed. Synchronization mechanisms are used to ensure that processing is done efficiently and that data is processed in a consistent and accurate manner. This helps to ensure that analysis and decision-making based on the data is reliable (Krishnamurthi et al., 2020). The Resource Management: In edge-based applications, resources such as processing power and memory may be limited. Synchronization mechanisms are used to ensure that resources are allocated in a way that maximizes their usage while minimizing their waste. This helps to ensure that applications can operate within the constraints of the available resources. The Device Coordination: In IoT and edge-based applications, devices must work together to achieve the desired outcomes. Synchronization mechanisms are used to ensure that devices are coordinated in their activities, which helps to reduce conflicts and ensure that the application operates as expected. The Security: Security is a major concern in IoT and edge-based applications. Synchronization mechanisms are used to ensure that devices are communicating securely and that data is protected from unauthorized access or modification. Overall, synchronization plays a critical role in ensuring the reliability and effectiveness of IoT and edge-based applications. It helps to ensure that data is collected, processed, and analyzed in a consistent and timely manner, and that devices are coordinated in their activities. This is essential for achieving the desired outcomes and ensuring the success of the application.

1.5 Fog and edge computing

The fog computing and Edge computing are both paradigms for distributed computing that aim to bring computational resources closer to the network of edge, where data's generated even consumed. Here is a brief overview of each. Edge computing refers to the practice of processing data locally, on devices or servers located at the edge of the network, rather than sending it back to a centralized data center for processing. Edge computing is often used in applications where low latency is important, such as in autonomous vehicles or real-time monitoring systems. Edge computing can also help reduce network bandwidth requirements and improve reliability by reducing the dependence on a centralized infrastructure.

Fog computing, on the other hand, refer to such practice of extending cloud's computing capabilities to edge networks, often through the use of intermediary devices known as fog nodes. Fog nodes are responsible for performing processing and storage tasks on behalf of edge devices, and can also provide network connectivity and security services. Fog computing is often used in applications that require more complex processing, such as in healthcare or industrial automation.

Both edge computing and fog computing are intended to improve the performance and efficiency of distributed computing systems by moving processing closer to where data is generated and consumed. The main difference between the two is the level of abstraction: edge computing is focused on local processing and devices, while fog computing is focused on extending cloud computing capabilities to the edge.

1.6 Clock synchronization

Clock synchronization refers to process for aligning the different device's clocks in a distributed computing system. In order for devices to communicate effectively, their clocks need to be synchronized so that they agree on the timing of events. (Jia et al., 2019). Clock synchronization is especially important in real-time systems, where precise timing is critical for correct operation (Geng et al., 2018).

There are several different techniques for clock synchronization, including:

1. Network time protocol (N.T.P)

The N.T.P is largely used protocol for clock's synchronization over the large internetworks and on internet. It works by exchanging time information between servers and clients and adjusting the local clocks to the matched time supported by the server.

2. Precision time protocol (PTP)

PTP is a more advanced clock synchronization protocol that is designed to provide sub-microsecond accuracy. It works by exchanging timing information between devices using a master-slave architecture, with the master device serving as the reference clock.

3. Global positioning system (GPS)

GPS can be used as a source of accurate time information for clock synchronization. Devices can use GPS signals to determine their location and the time offset between the GPS clock and their local clock.

4. Radio time signals

Radio time signals, such as those broadcast by national time and frequency standards organizations, can also be used for clock synchronization. These signals are typically transmitted using long-range radio waves and provide accurate time information to devices that can receive them.

Overall, clock synchronization is an important aspect of distributed computing systems, and there are several techniques available for achieving accurate and reliable synchronization. The choice of technique will depend on the specific requirements of the system and the available infrastructure.

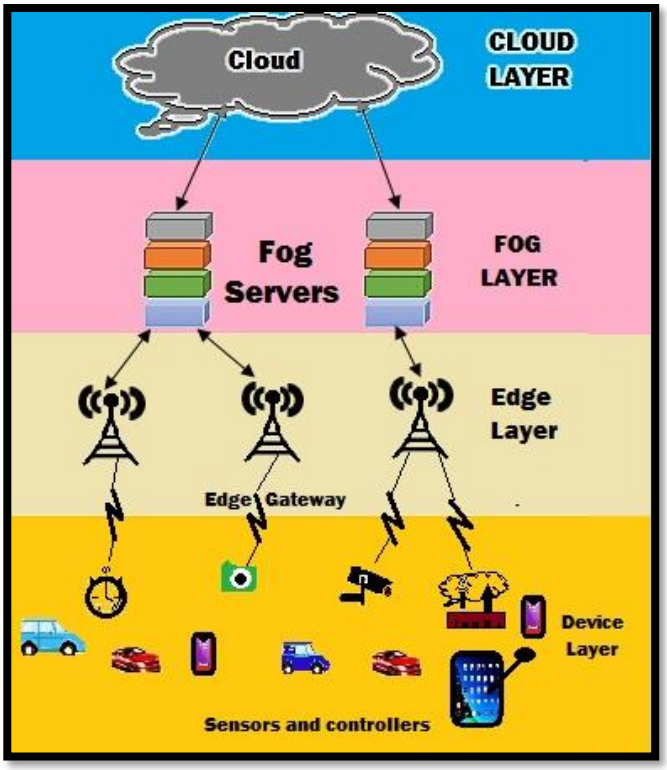

Figure 1.3 Showing fog and edge computing scenario

The requirement for synchronizations and control techniques in the IoT is motivated here by three application situations where synchronizations between devices is crucial. Hence, showing an example of Assisted Edge Autonomous Driving along with Capturing of Synchronized Data wherein shown the cars as workers for better understanding and showing the scenario of an best working synchronization.

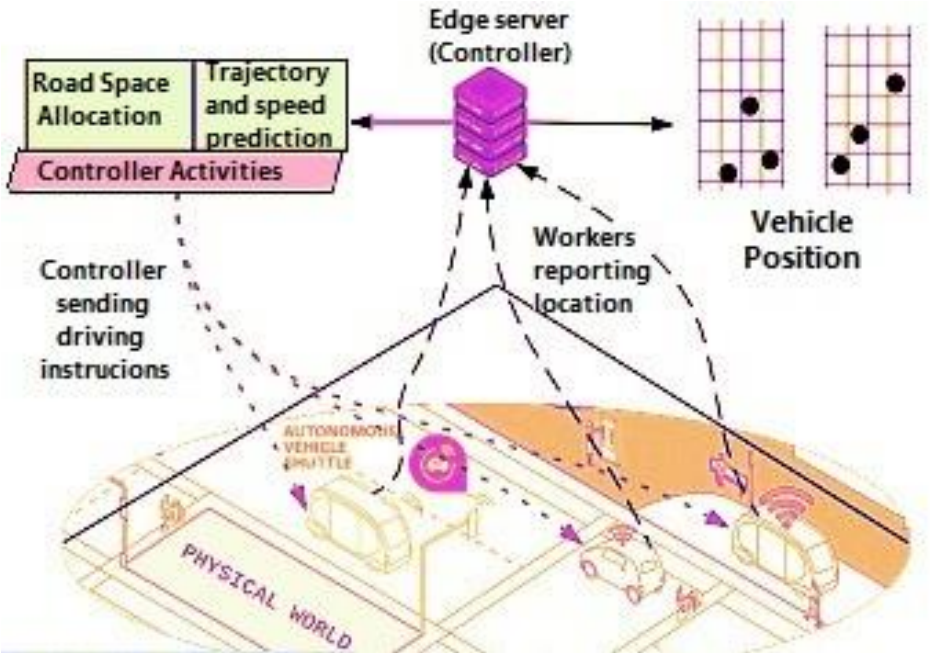

Figure 1.4 **Showing scenario of edge assisted auto driving controllers and cars as**

Workers (back support with edge server)

Furthermore, now take another example of bridge health system as this is a concrete illustration of the use case for data acquisition synchrony depicted on the right side of Fig. 1.4. To keep a careful eye on vital health over structure of bridge, strain monitoring at the joints of bridge and over other crucial spots and the structure is necessary. Coordination of the data capturing activities is required to ensure that high-quality measurements are collected when the loading is at a specific conjuration. The location and weight of the vehicles on the bridge at that precise moment, as collected by drones, would be used to calculate the loading arrangement. The most precise technique to do such a measurement is to simultaneously activate all relevant equipment (drones, vehicles and sensors) for the function of measurement. If many devices perform the measuring function at various times, a difficult reconstruction method must be carried out to ascertain the concurrent loading. Only when the cars are on the bridge are the finer measurements relevant. In other words, they are not required to keep taking and reporting negated position data while they are not on the bridge.

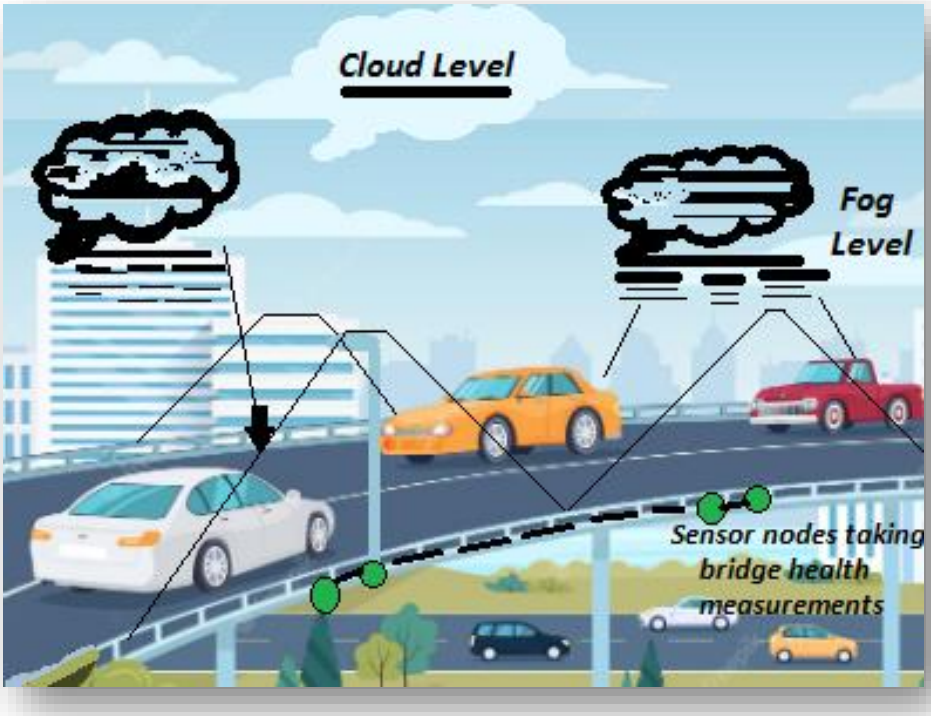

Figure 1.5 Showing delivery and bridge monitoring system (sample drone)

Objectives

1. To investigate fault tolerance in the synchronization strategies.
2. To design a novel and fast synchronization strategy.

CHAPTER 2
REVIEW OF LITERATURE

Papazachos and Karatza (2011) studied gang scheduling in multi-core clusters, discovering that synchronization in gang scheduling and co-scheduling is achieved by waiting for busy time when parallel project tasks require simultaneous scheduling and execution. Gang scheduling makes dependent tasks wait until processing becomes available for collective rescheduling, enabling job pausing and resumption.

Wang et al. (2016) proposed edge analytics framework for IoT including data collection, analysis, and synchronization mechanisms. Wang et al. (2017) presented an IoT-based health monitoring system incorporating resource allocation and synchronization techniques.

Majid et al. (2022) provided comprehensive review of time synchronization techniques in wireless sensor networks, examining both centralized and decentralized methods.

Lian et al. (2018) proposed distributed resource management framework for edge computing in IoT, including resource allocation, load balancing, and fault tolerance mechanisms.

Liu et al. (2019) provided comprehensive survey of data synchronization techniques for edge computing, including publish-subscribe systems, message queues, and distributed databases.

Ren et al. (2021) proposed synchronization mechanism for distributed edge intelligence, including consensus-based algorithm for coordination between distributed devices.

Sun and Zhang (2012) studied natural synchronization in fireflies, where male firefly flash ejections synchronize over time. Pulse-Coupled Oscillators (PCOs) model this behavior, where oscillators experience regular fluctuations and communicate by responding to neighboring pulses.

Steiner et al. (2014) proposed clock synchronization approach using time-triggered Ethernet standard. The TT Ethernet features switching and end systems connected through redundant channels for fault tolerance, with synchronization masters, clients, and compression masters.

Shi and Cao (2016) described edge computing as paradigm enhancing IoT system capacity for computation, storage, and information management by examining data at network edge, including smartphones and routes between information sources and fog layers.

Harris et al. (2017) identified explicit and implicit co-scheduling types. Firefly-inspired synchronization algorithms are used in IoT wireless sensor networks (Yadav et al., 2017), where sensors function as PCOs attempting phase synchronization through phase adjustment.

Xie et al. (2018) investigated clock synchronization in IoT, identifying critical parameters: clock skew difference, clock offset difference, and integration time for synchronizing out-of-synchrony system clocks.

Mani et al. (2018) identified clock synchronization challenges in IoT including varied environmental conditions, low device cost requirements, unstable network connections, constrained bandwidth, and limited device capabilities.

Fan et al. (2019) distinguished statistical synchronization methods providing statistical assurance on maximum clock offset versus deterministic approaches guaranteeing maximum offset without doubt. A blockchain-based time synchronization agreement mechanism was developed for IoT security requirements.

Amiri and Gunduz (2020) identified synchronization as significant issue in iterative methods like distributed real-time ML, affecting convergence rate and iteration times. The Bulk Synchronous Parallel (BSP) model ensures complete synchronized participation but struggles in diverse systems where devices complete calculations in divergent times. Asynchronous Parallel (ASP) requires no waiting between workers, reducing straggler impacts and communication costs but requiring more iterations for convergence.

This research develops synchronous scheduling methods for IoT jobs with fog control and AI applications where node availability verification is essential before job execution, particularly for asynchronous activities with strict timing restrictions. Unlike co-scheduling and gang scheduling, tasks are not preempted—once started, they continue until completion or failure. Unlike PCO-based synchronization seeking eventual synchronization, mixed task arrangement requires scheduling for local, asynchronous, and synchronous operations with quorum checking for availability verification before synchronous task execution.

RESEARCH METHODOLOGY

This research aims to explore synchronization techniques for edge assistance and IoT applications. In IoT and edge computing contexts, it is important to synchronize devices and systems for them to work effectively and reliably. This study proposes and evaluates different synchronization systems, considering factors like accuracy, scalability, energy efficiency, and delay.

Here is a list of the resources and methods used in this research:

Materials:

IoT Devices

Various types of IoT devices will be used, each with different computing capabilities, communication protocols, and synchronization requirements. These devices can include edge devices, sensors, and actuators.

Edge Servers

To assist with synchronization, edge servers or computing nodes will be installed. These servers can act as synchronization anchors and provide additional computing power.

Network Infrastructure

A network infrastructure will be established to connect the IoT devices and edge servers. This infrastructure may include switches, wireless access points, and routers.

Different synchronization techniques will be implemented and examined, such as the Network Time Protocol (NTP), the Precision Time Protocol (PTP), and reference broadcasts.

Simulation Tools
Simulation tools like ns-3, OMNeT++, or MATLAB will be used to model and simulate the synchronization schemes. These technologies enable realistic assessment and performance analysis.

Methods:
Experimental Setup
The network infrastructure, edge servers, and IoT devices will be configured in accordance with the needs of the investigation. The distribution of the gadgets will take place in a controlled setting to simulate real-world situations.
Implementation of synchronizations strategies
The edge servers and IoT devices will use the chosen synchronization strategies. The firmware may need to be changed, synchronization methods may need to be implemented, and synchronization parameters may need to be set.

Data collection
During the experimental phase, data on synchronization accuracy, latency, energy use, and scalability metrics will be gathered. This information will be used to assess how well various synchronization techniques perform.

Performance evaluation
Appropriate statistical techniques will be used to analyze the data. The effectiveness of each synchronization approach will be compared and assessed based on predetermined indicators.

Comparative study
A comparative study will be carried out to determine the advantages and disadvantages of each synchronization approach. For comparison, elements like synchronization precision, latency, energy efficiency, and scalability will be considered.
To shed light on synchronization methodologies for IoT and edge assistance applications, the experimental assessment and simulation analysis results will be reviewed. In light of particular objectives and limits, the study will offer suggestions for choosing the best synchronization approach.
Synchronization scheduling schemes coordinate the execution of tasks in a distributed computing system, such as a network of sensors or IoT devices. These schemes aim to ensure that tasks are executed in a way that maximizes system performance while respecting resource constraints and synchronization requirements. This research will offer the synchronization strategies regarding IoT and edge assistance applications and enhance the previous work in the field by providing new algorithms and assumptions made after evaluating the results.
Here are a few examples of synchronization scheduling schemes

1. **Time-Division Multiplexing (T.D.M.)**
T.D.M. is a standard synchronization scheduling scheme in wireless sensor networks. It works by dividing the available time into fixed-length time slots and allocating each device a specific time slot for communication.

2. **Priority-Based Scheduling**
Priority-based scheduling is a scheduling scheme in which tasks are assigned priorities based on their relative importance or urgency. Tasks with higher priorities are executed first, while lower-priority tasks are delayed or skipped if necessary.

3. **Round-Robin Scheduling**
Round-robin scheduling is a simple scheduling scheme in which tasks are executed in a fixed order, each for a fixed amount of time, before moving on to the next task.

4. **Dynamic Scheduling**
Dynamic scheduling is a scheme in which task priorities and execution times are adjusted dynamically based on system conditions and resource availability. Dynamic scheduling can optimize system performance in real time, but it may be more complex than static scheduling schemes.
We create three synchronization scheduling techniques to handle various circumstances, depending on the degree of previous information about mission graphs (unsubstantial/average) & task frequency, regularity, or radio city of arrival. The following is a description of the notations used in the algorithms. The tasks in the queue, Tcurr, and T next, respectively, must be scheduled. After completing the execution of a previous job, a worker's available time is represented by T avail.

Before beginning to execute a synchronous task (c2ws), it is necessary to verify the presence of workers to achieve the requisite QoSync. We refer to this procedure as quorum checking. The Worker updates the controller on their accessibility to the synchronous job as part of the quorum check procedure. Depending on the degree of synchronization, we ran the majority was present to see if the control system had the required workers available to carry out the synchronized job. The start time of the quorum is determined by the degree of synchronization, the distribution of the execution time, and even the completion job time for all workers prior to the synchronization point. Table 3.1 shows the syn. Algo Symbols are primarily used in quorum checking. *Quorum checking* is used in synchronization strategies to ensure consistency and correctness in distributed systems. In distributed systems, where multiple nodes or processes work together to achieve a common goal, maintaining consistency and avoiding conflicts becomes crucial. Quorum checking helps in achieving these goals.

Table 3.1 Synchronization algorithm's symbols

Symbol	Description
T_{curr}	current task to be scheduled
T_{quorum}	quorum check task
T_{update}	status update task run by workers
t_{avail}	available time of a worker
λ	time delay before re-attempting quorum
α	ratio of workers required to pass quorum
τ	predicted quorum check time

3.1 Scheduling algorithm (static synchronization)

The SSSA displayed in Algorithm 1 run during build time presumes that we are already familiar with work graphs in G. Topological ordering of the job graph places asynchronous and synchronous jobs higher on the priority scale, respectively. Tasks with shorter predicted execution times are prioritized to break ties amongst jobs of the same kind. Precedence constraints are preserved, thanks to the topologically sorted set S. When quorum verification on the preliminary schedule fails, SSSA generates other backup schedules that are utilized instead. Schedules are created during build time. An input to the method is also the no. of a quorum (maximum) retries to try at each point of synchronization.

Algorithm 1: Static synchronization scheduling algorithm

```
1  Input: Task graph G and maximum quorum retries.
2  Output: Primary and alternative schedules.
3  set S = top_sort(G) and schedule counter m = 1
4  StatSchd(S, m, t_avail):
5    while S ≠ ϕ do:
6      if type(T_next) = c2w_s:
7        s_schedule(T_update, m, t_avail) and compute τ
8      if type(T_curr) = c2w_s:
9        s_schedule(T_quorum, m, τ) ;                ▷ calls QuorumCheck(m)
10       if retries not exceeded:
11         StatSchd(S, m, t_avail + λ)
12       else if retries exceeded:
13         remove(S, T_curr)
14         StatSchd(S, m, t_avail)
15       s_schedule(T_curr, m, t_avail)
16     else if type(T_curr) = c2w_a ∥ type(T_curr) = w_l:
17       s_schedule(T_curr, m, t_avail)
18 QuorumCheck(m):
19   if quorum = passed:
20     continue schedule m
21   else:
22     switch to schedule m++
```

Table 3.2 showing functions explanation in (S.S.S.A)

Symbol	Description
$top\_sort(G)$	topological sort of tasks in task graph G
$StatSchd(S, m, t_{avail})$	function that accepts topologically sorted set of tasks S, the schedule m, and time t_{avail}
$type(T)$	the type of task T: sync, async or local
$s\_schedule(T, m, t)$	schedule task T in schedule m at time t
$remove(S, T)$	remove task T from task set S
$QuorumCheck(m)$	perform quorum check on current schedule m

Lines 6 and 7 show how updated transmitting tasks have been scheduled to the worker's one task prior to the point of synchronization. Quick upgrades allow an administrator to grasp

communications while busy employees work, reducing wait times (wasted work cycles). The quorum check task Quorum is scheduled after assessing the predicted variance in execution progress among workers (Line No. 9). If a quorum is achieved, the sync. Tasks (Line No.15) are scheduled. The existing schedule (m, the schedule) is continued (Line No.20). On the other hand, the synchronization procedure is resumed after waiting for a delay if there remains the majority of repeated attempts provided after the majority of failures (Line No. 22), (Line No.11). If there are no more opportunities for retries, the synchronization job fails. We complete the following task (Line Nos. 13, 14). In S (Lines Nos. 16, 17), we and c2wa are scheduled as soon as they reach the front line.

3.2 Scheduling algorithm (dynamic sync.)

The controller executes the DSSA shown in Algorithm 2 at runtime. In the dynamic approach, presumptions established in the static synchronizations scheduling algorithm, likewise, previous knowledge to the expected execution times of topology's of tasks graph along with tasks, are loosened. Scheduling choices are made on the y as tasks become available because DSSA considers that we cannot reliably foresee the pattern of task arrival. The DSSA Flow is depicted in (Fig. 3.1) i.e. Flow of Algorithm (D.S.S.A). To represent the scheduling activities that take place at the worker, we utilize [W].

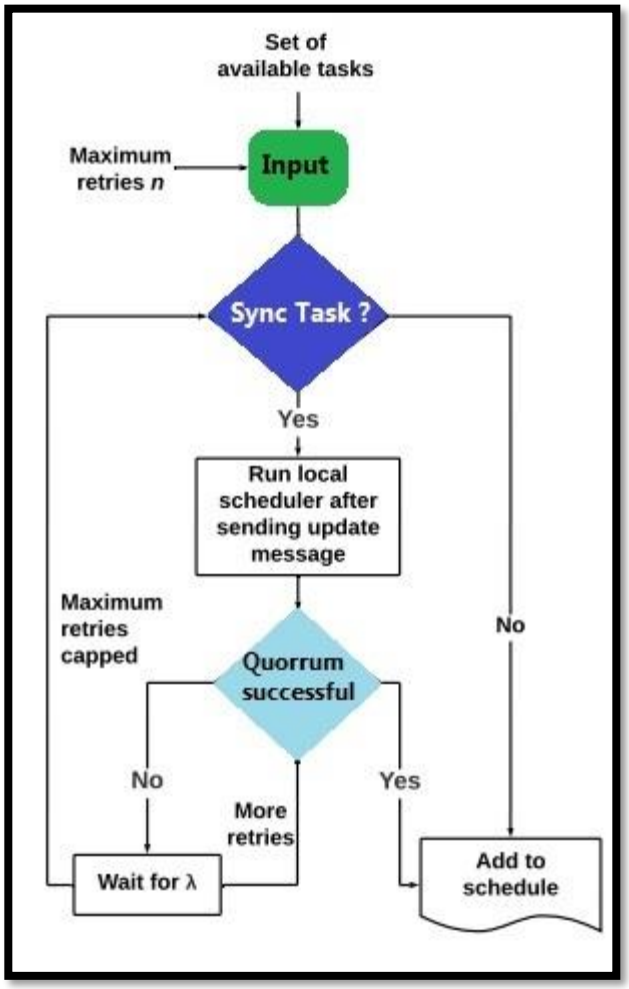

Figure 3.1 Showing flow of algorithm (D.S.S.A)

Algorithm 2: Dynamic synchronization scheduling algorithm

```
1  Input: Task graph G and maximum quorum retries.
2  Output: Execution of the tasks.
3  set A = {set of available tasks}
4  DynaSchd(A,t_avail):
5    while T_curr = get_tasks(A) and T_curr ≠ ϕ do:
6      if type(T_curr) = c2w_s:
7        d_schedule(T_update, t_avail)   [W]
8        δ = compute_slack()
9        LocalSchd(δ)   [W]
10       d_schedule(T_quorum, t_avail)   [W]
11       if quorum = failed && retries not exceeded:
12         DynaSchd(A, t_avail + λ)
13       else if quorum = failed && retries exceeded:
14         remove(A, T_curr)   [W]
15         DynaSchd(A, t_avail)
16       else if quorum = passed:
17         d_schedule(T_curr, t_avail)   [W]
18     else if type(T_curr) = c2w_a || type(T_curr) = w_l:
19       d_schedule(T_curr)   [W]
```


Table 3.3 showing function's explanation in (D.S.S.A)

Symbol	Description
$DynaSchd(A, t_{avail})$	function that accepts set of available tasks A and time t_{avail}
$get\_tasks(A)$	get the current task in A
$d\_schedule(T, t)$	schedule task T at time t
$compute\_slack()$	computes time gap between available time and expected quorum check time
$remove(A, T)$	remove task T from task set A

The controller must ask the workers to schedule the update transmitting job whenever we reach a synchronization point (Line-7). The controllers initiate native schedulers the employers are prepared ("Line No. 9 Algorithm, 3") before quorum checking. Find out if any specific roles need to be done may be executed if its time of execution (tl) is shorter than the computed slack (Line No. 8) in terms of workers'

available times. It is done to reduce the number of work cycles wasted on the employees arriving at the synchronization point first.

The workers do quorum checking (Line No. 10) By probing the controller to determine whether or not there is a quorum successfully made. The c2ws task is scheduled to run simultaneously on all available workers if a quorum is achieved, as algorithm two defines the step in the process well. The workers wait a while before retrying the synchronization operation if such quorum fails, and there may still be a retry available (Line No. 12). While algorithm two already defines the algorithm of Dynamic Scheduling Sync. The synchronization job failed (Line No-14), and execution continues if all retries are unsuccessful (Line No.15). The employees' wl and c2wa tasks are scheduled to be run at the soonest as possible (Line No. 19).

3.3 Scheduling algorithm (Micro Batch Sync.)

Micro Batch Synchronization Scheduling Algorithm (MBSSA) is a synchronization scheduling scheme used in distributed computing systems to achieve synchronization between tasks. It is beneficial in systems where data is generated in small batches or bursts, such as in Internet of Things (IoT) applications.

In MBSSA, tasks are grouped into micro-batches based on data processing requirements and timing constraints. Each micro-batch is assigned a deadline by which it must be processed, and tasks within each micro-batch are scheduled based on their priority and estimated execution time. The algorithm considers the expected arrival time of data batches and the processing time required for each micro-batch to make sure / ensure the overall timing requirement of the system is met.

MBSSA is designed to be flexible and adaptable to changing workload patterns, making it well-suited for dynamic and unpredictable systems. Using micro-batches allows for efficient use of computing resources and reduces the risk of overloading the system with a large batch of data. It also allows for efficient use of network bandwidth, as data can be transmitted in small batches rather than large chunks.

Overall, MBSSA is an applicable synchronization scheduling scheme for distributed computing systems that process data in small bursts or batches and can help ensure efficient use of computing resources while meeting timing constraints. According to Fig. 4.3, the MBSSA inherits characteristics from both SSSA and DSSA. SSSA makes more educated scheduling recommendations, which eventually become stale, especially for long-running applications, nevertheless presumes knowledge of the task graph and execution time distribution. At the same time, DSSA bases its scheduling decisions on the premise that we need more knowledge about the jobs entering the system or their distribution of execution times. It helps to get rid of the SSSA's scheduling decisions becoming stale. A collection of incoming tasks are dynamically combined into a micro-batch by MBSSA, which then schedules them statically. When a micro-batch is generated, more tasks are gathered into a fresh micro-batch and readied for scheduling. A time slice (tasks within a specific period) or a predefined size might be used to create micro-batches.

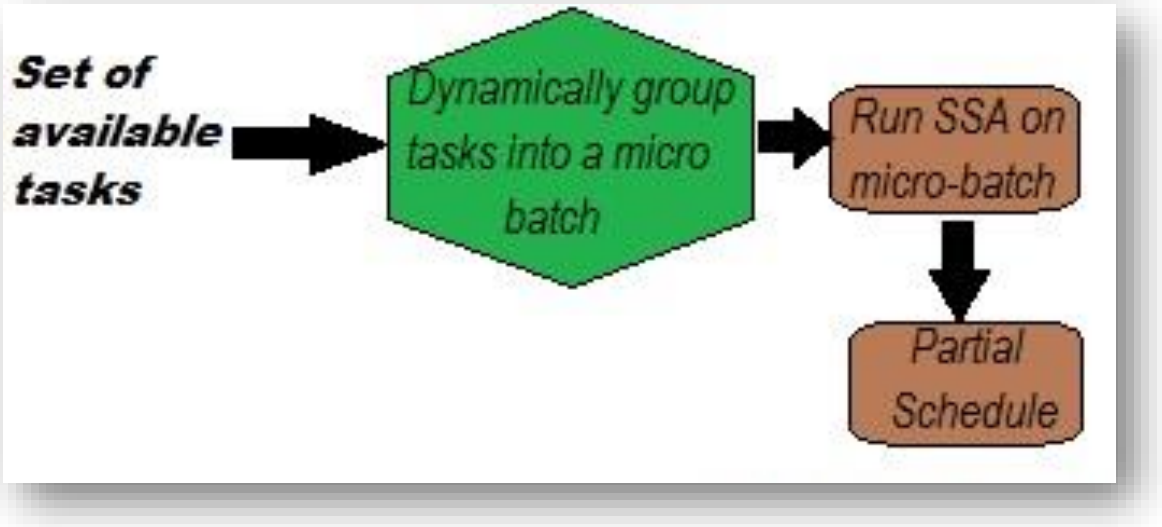

Figure 3.2 Showing process flow of scheduling algorithm (micro-batch)

The static scheduling technique is applied to the group of dynamically incoming jobs via MBSSA. When task arrival patterns can be roughly predicted and communication costs between employees and controllers are high, MBSSA works remarkably well.

3.4 Scheduler (Local)

An optimization strategy called local scheduling was created to reduce the amount of time that employees who arrive at the synchronization point sooner than other workers must wait. Since they only communicate with the controller rather than directly, employees at a synchronization point are unaware of one another. Therefore, a worker executes a local scheduler Local Schd() to compare its current time to the projected finish time for all workers as specified in Algorithm 3.

Algorithm 3: Local scheduling algorithm for task–based synchronization

```
LocalSchd(δ):
L = {local worker task queue}
while get_tasks(L) = w_l do:
  if t_avail + t_l ≤ δ:
    schedule(w_l)
    revise_available_times(t_avail, max_t_avail)
  else:
    continue
```

If the time variance among the other workers' predicted availability and the particular node's availability criterion was more than the total time of the local job, the local schedule would execute the local task (if it has one). The employee would be left idle in this scenario, and the neighborhood scheduling couldn't assign any nearby jobs.

3.5 Algorithm of fast synchronization

The synchronizations algorithm displays the actions and choices taken from the workers and controllers depending on the runtime configurations, the algorithm generates various runtime actions that the clusters can implement. The available timings for C1 and C2, the fast and slow clusters that are t1av & t2av, respectively. Fig. 3.3 further displays the runtime synchronizations. Equations 6.7 and 6.9 are solved by the controller to produce the synchronizations schedule and axes the three synchronizations alternatives for each synchronizations point. As soon as the workers become available, asynchronous activities are executed.

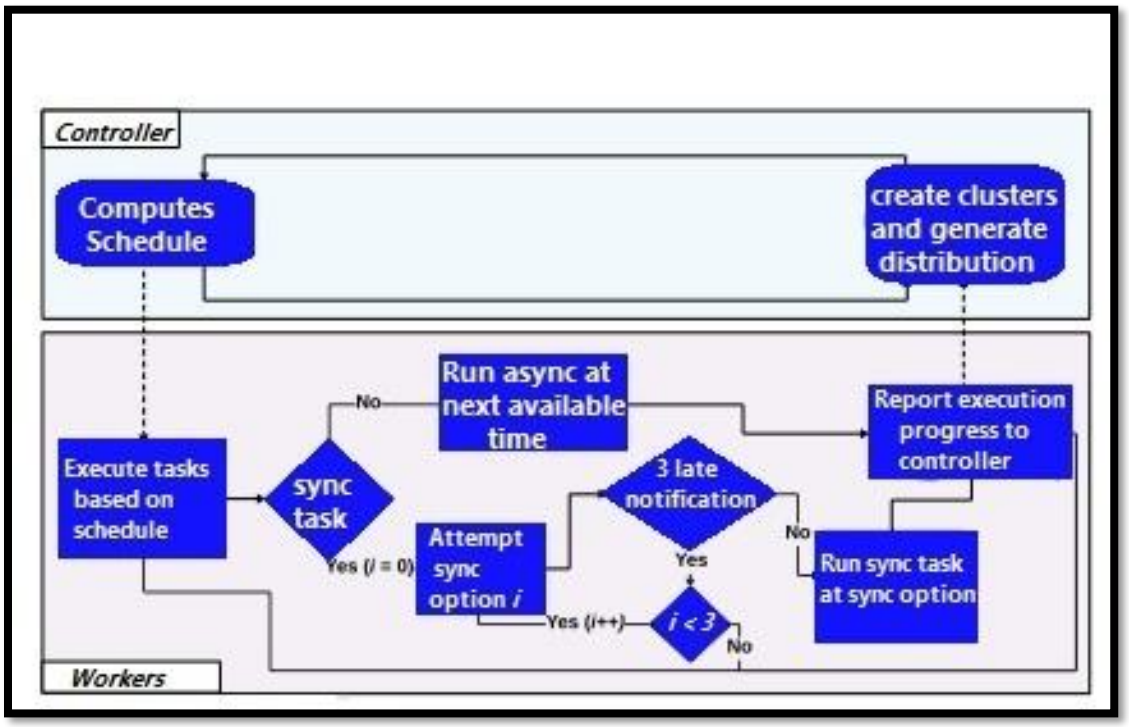

Figure 3.3 Showing actions by controllers and workers during run time in fast sync.

Algorithm 4: Synchronization algorithm

```
1  Controller:
2  Fix the three sync options t_s^1, t_s^2 and t_s^3 by solving Equations 6.7, 6.8 and 6.9
     respectively
3  forall workers do:
4    First sync option:
5    if (t_av^1 ≤ X_1) and (t_av^2 ≤ X_2):
6      execute(T_sync, t_s^1);
7      end;
8    elif (t_av^1 ≤ X_1) and (t_av^2 > X_2) and send(C_2, late_notify):
9      if t_l^1 ≤ t_s^1 − t_av^1:
10       execute(T_local, t_av^1);
11     proceed to line 18;
12   elif (t_av^1 > X_1) and (t_av^2 > X_2):
13     proceed to line 18;
14   elif (t_av^1 ≤ X_1) and (t_av^2 > X_2) and no_late_notify:
15     abort(sync);
16     end;
17   Second sync option:
18   if (t_av'^1 ≤ X_1') and (t_av'^2 ≤ X_2'):
19     execute(T_sync, t_s^2);
20     end;
21   elif (t_av'^1 > X_1') and (t_av'^2 ≤ X_2') and send(C_1,late_notify):
22     if t_l^2 ≤ t_s^2 − t_av^2:
23       execute(T_local, t_av^2);
24     proceed to line;
25   elif (t_av^1 ≤ X_1) and (t_av^2 > X_2) and no_late_notify:
26     abort(sync);
27     end;
28   Third sync option:
29   if (t_av''^1 ≤ X_1'') and (t_av''^2 ≤ X_2''):
30     execute(T_sync, t_s^3);
31     end;
32   elif (t_av''^1 ≤ X_1'') and (t_av''^2 > X_2''):
33     abort(sync);
34     end;
```

Likewise, Algorithm 4 also shows the Execute (T sync; tns) shows that sync option n should be used to start the sync task Tsync at time tns and finish it there. If both clusters become available for the option before the projected availability times (Line 5), the sync job will be carried out by staff members at the initial synchronization option (Line 6). Both units go on to their next synchronization point if either of the clusters arrives early at the initial sync option. The quicker clustered can perform a local job before moving on to the second option, though, if that slower clustered is tardy and tells it (Lines No. 8–10). Hence, algorithm 4 is a much better algorithm with proper synchronization, and in this series, the synchronization ends when a late cluster fails to transmit a late message.

3.6 Synchronization approach in Edge & WSN Networks

Network analysis examines synchronization-related message exchanges during recent synchronization period, adapting subsequent period duration to reduce overall energy consumption. Sensor-Initiated (SI) mode conserves network resources during infrequent traffic when synchronization latency is not critical. Sensor clocks may desynchronize during no sensing events.

The proposed synchronization strategy is adaptable and self-contained, requiring physical broadcast channel (naturally provided by wireless media) and connected network for synchronization ripple propagation. Reference nodes transmit strobe information regularly, initiating synchronization waves. Sensor nodes dynamically select closest reference nodes for clock synchronization.

Network Time Protocol (NTP) and Precision Time Protocol (PTP) are commonly used in edge networks. In WSNs, Time Synchronization Protocols (TSPs) like Reference Broadcast Synchronization (RBS) use distributed approaches where nodes communicate with neighbors to estimate time differences

RESULTS

It was important for us to conduct some experiments to evaluate the feasibility and performance of our idea. To describe the setup for these experiments, we used a Directed Acyclic Graph (DAG) consisting of synchronous, asynchronous, and local tasks. This task graph is similar to techniques used in synchronous machine learning and parameter synchronization methods like Bulk Synchronous Parallel (BSP), Stale Synchronous Serial (SSP), and Dynamic Stale Synchronous Parallel (DSSP).
The results of the experiments can be divided into four phases:

1. Calculating global weights

In this step, local data is used to calculate weight gradients. These gradients are then sent to the parameter server, which adjusts the local weights using the global weights determined earlier.

2. Model training and updating

During this stage, workers participate in model training and updating. They collect data, use the model, and perform training tasks. Local tasks are used to describe the specific calculations performed by workers to ensure smooth operation of the current program. These tasks are started based on application requirements and parameters.

3. Task completion time

A mixed distribution is used to determine how long it takes workers to complete specific tasks. The distribution used in the experiment mimics both short jobs (25 ms) and long jobs (80 ms) based on traces from cluster research. Task completion times are categorized into two groups: those for quick execution and those for planned performance.

4. Simulation parameters

The parameters of the simulation include:

- Synchronization step: Represents the number of computers required to reach a consensus during the synchronization step.
- Number of Workers: Reflects the maximum number of workers engaged in system activity at any given time.
- Number of Continuous Task Graph Runs: Specifies the quantity of Continuous Task Graph Runs.
- Frequency of Clustering: Indicates how often the controller re-clusters the data.

By adjusting these simulation parameters and observing the performance of the distributed computing system, we can assess the feasibility and performance of the proposed idea.

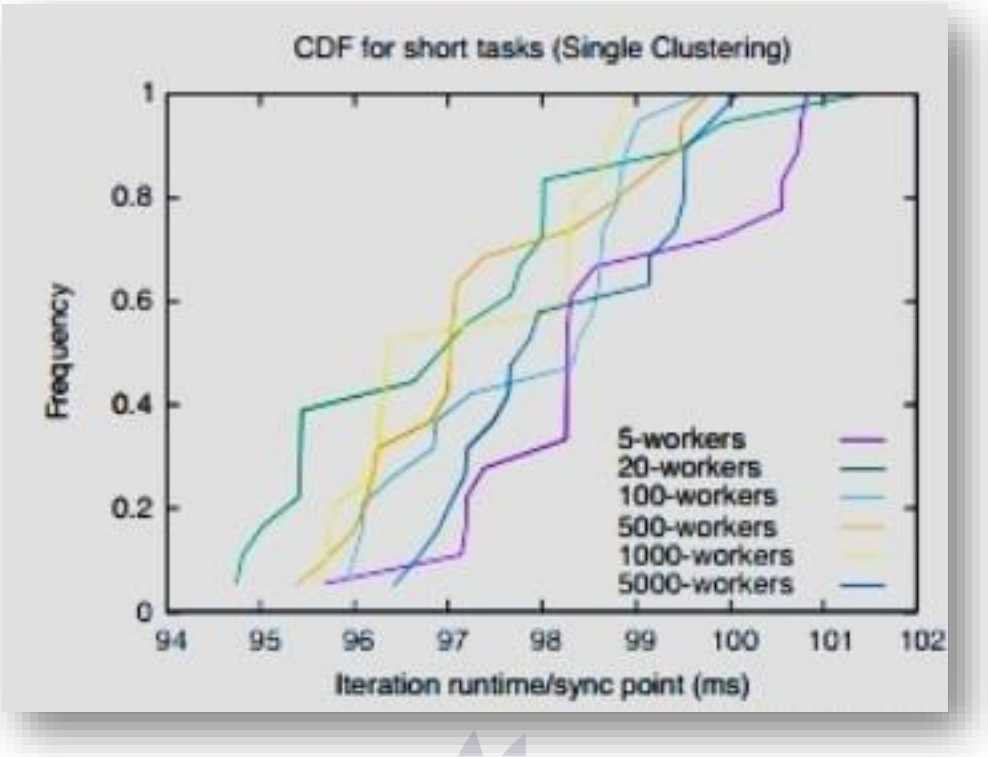

Figure 4.1 short tasks runtime (fix clusters)

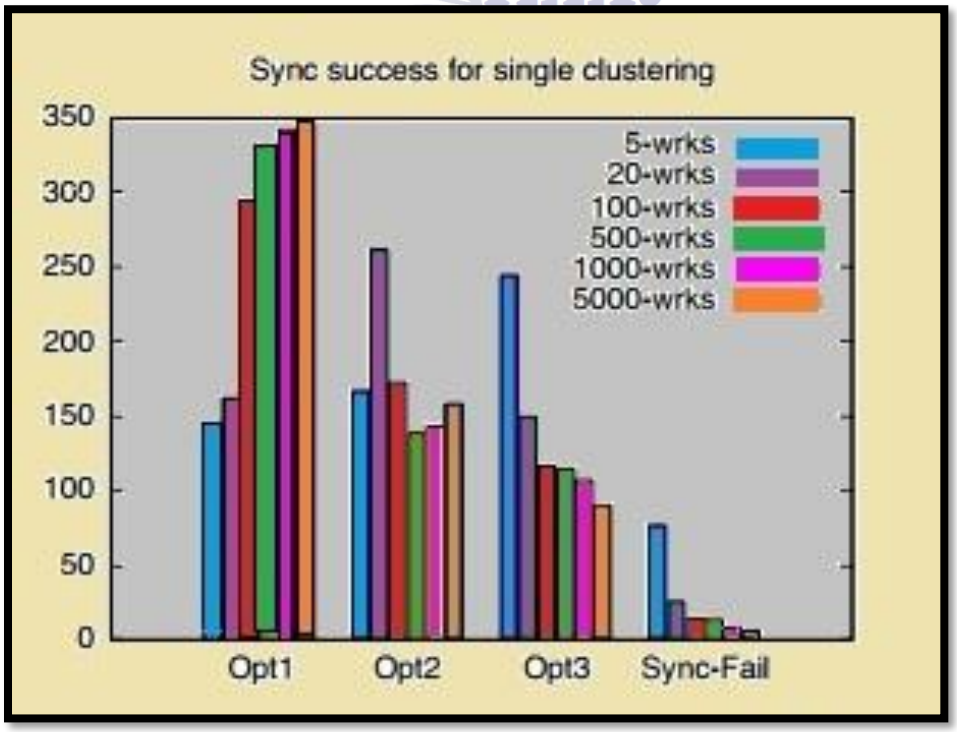

Figure 4.2 Successful and failed sync. Events.

We change the execution time variation of jobs among workers to examine the effects. We consider the effect that employee variability has on duration per syncing endpoint and minority attendance to gauge its effect on our method. The likelihood of having stragglers grows as the standard deviation of a job among numerous employees rises. For 100 employees and temporary jobs, the execution time varies from 1:5 to 6 milliseconds, as illustrated in Fig. 6.16Each sync fact gets longer when the worker job execution time variance increases from 1:5 to 6 milliseconds. The average sync participation for all execution time variants is 0:75, with an average higher for execution time variants 1:5 when the execution time variance widens.

Table 4.1 showing comparison of Fast Sync with related works

Metric	Fast_Sync	ASP	BSP	SSP	DSSP
Straggler mitigation	Yes, using clustering, quorum and late notification	No	No	Yes, using fixed bounded staleness	Yes, using flexible bounded staleness
Message overhead	Low	Very low	Very high	Moderate	High
Sync slack	Not allowed	Allowed	Bounded	Flexible but bounded	
Adaptable to dynamic systems	Highly adaptable	Adaptable	Not adaptable	Adaptable	Adaptable

There are a few challenges in results and simulation as heterogeneity on edge: The interoperability of heterogeneous devices and technologies is one of the primary problems with distributed or decentralized edge systems. Even though our technique reduces rapid synchronization will be hampered by the impact of laggards, devices that are diverse and have a wide variety of job times for execution. Node faults and failures: Our technique uses quorum requirements, clustering, and the late notification protocol to integrate fault tolerance. Rapid synchronization becomes increasingly challenging in a dispersed border system with an elevated frequency of node errors, failings, and recovery. An unstable system will make it challenging to achieve synchronization, even with the best possible fixed synchronization choices. Network Connectivity: For quick synchronization, node connectivity is crucial. The controller must have message-sending capabilities (current cluster composition, partial schedules. Also, employees must be able to interact with the controller and one another. Our algorithm will operate much more slowly on an unreliable network.

By comparing our algorithm's performance to that of the ASP, BSP, and SSP parameter server models (each with a different staleness threshold), we can assess how well it performs. We utilize Ray, a Python framework for developing distributed applications, to create all the frameworks, including our synchronized distributed training method. We track all models' training durations, training iterations, and testing precision for various runtime configurations. We perform the trials with various worker counts. The number of training iterations necessary for the trained ResNet20 model to achieve a 70% testing accuracy is shown in Fig. 4.2. According to Fig. 4.2, BSP requires the fewest

training iterations for all groups of workers. However, each iteration for BSP takes a lot longer. It is so that any modifications made by workers may be applied at the limitation server before workers go on to the next iteration, which is a barrier used by BSP. Compared to ASP implementation for homogeneous cluster arrangement, SSP3, and SSP5 variations of the (SSP) use fewer training iterations and longer training times to attain 70% testing accuracy with staleness value of 3 and 5, respectively.

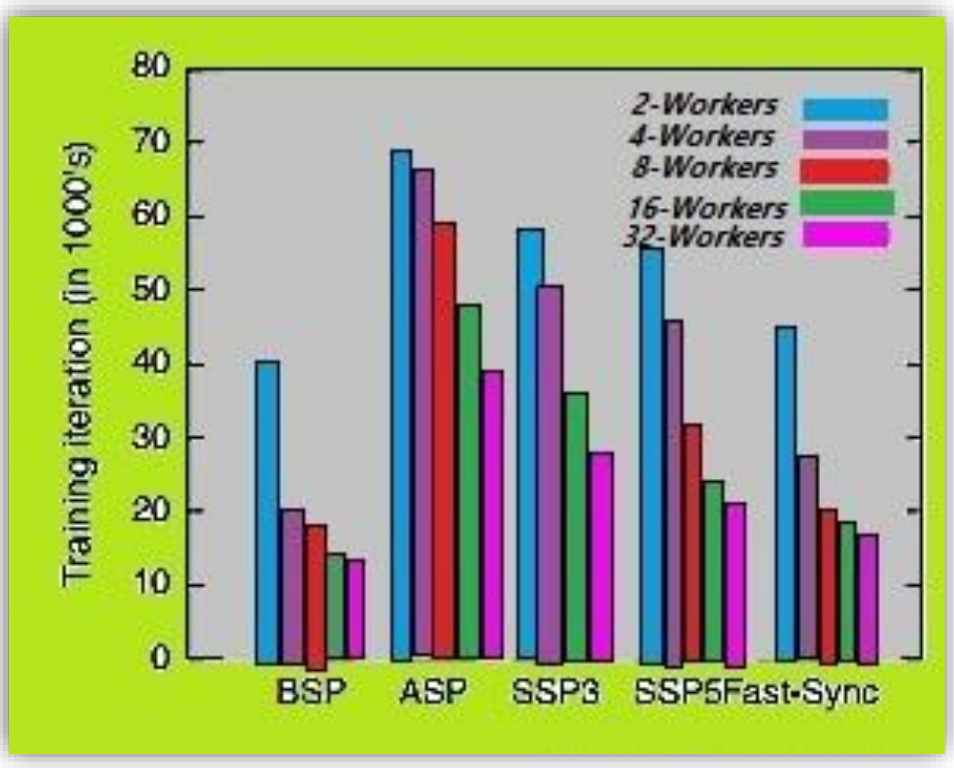

Figure 4.3 Showing variation of homogenous workers with 70% accuracy

We build the ResNet20 generator in both a mixed and a mixed configuration to study the effects of variation and to integrate some dropouts among the employees. Figures 4.3 and 4.4, respectively, Show the training period needed to obtain 70 percent accuracy for the homogeneous and heterogeneous cluster designs. For all structures to reach a 70% testing accuracy, extra training period was needed. with exception of ASP, which saw a smaller increase in training time, 8% on average and our method, which saw a 12% increase. With a 22% rise, BSP was the most affected, followed by SSP3 and SSP5, respectively.

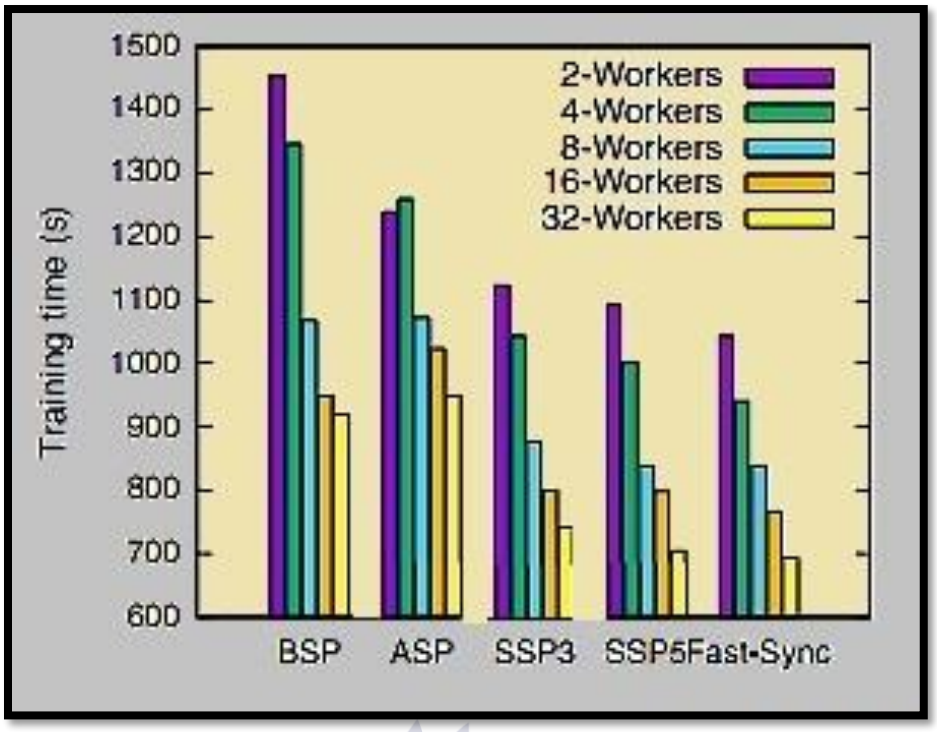

Figure 4.4 Showing amount of time required by workers

Then, for 8 employees in both homogeneous and heterogeneous cluster configurations, the testing accuracy vs. training time is calculated. At 200s of training time, BSP achieves 45% accuracy more quickly than other frameworks for the homogenous cluster arrangement. Beyond this, all other frameworks outperform BSP in terms of testing accuracy. In terms of performance, our approach is comparable to for initial training, ASP is used, and for further training, SSP implementations are used. Our approach, such shown in Fig. 4.2, provides an accuracy for the heterogeneous cluster arrangement that is greater or comparable to SSP and ASP for all training periods. This is due to the fact that our method employs clustering to put employees together, which significantly reduces worker communication compared to competing models. Our method of synchronization and the accompanying Table 6.2 compares the training models for parameters servers.

Here, it is very much necessary to highlight the difficulties in implementing quick synchronizations techniques in a (actual) edge AI setting. Because implementing a quick synchronizations strategy in a genuine (edge-AI) system can provide a number of difficulties, including, Communication Overhead: A fast synchronization scheme requires frequent communication between nodes to ensure consistency. This can result in a high communication overhead, leading to increased network congestion and latency. Then comes, Network Bandwidth Limitations: The communication between nodes requires network bandwidth. However, edge devices typically have limited bandwidth, which may not be enough to handle the increased traffic from synchronization. Far ahead comes, Hardware Constraints: The CPU, memory, and battery life of edge machines are often restricted. Fast synchronization schemes may require additional resources that may not be available on edge devices. Real-time Constraints: Real-time systems require immediate responses to events. However, fast synchronization schemes may require time to propagate updates, leading to potential delays in responding to events. Scalability: As the system's number of edge-AI nodes grows, the complexity's synchronization also increases, making it more challenging to deploy and maintain. The Security:

Fast synchronization schemes require frequent communication, which increases the risk of security breaches, such as man-in-the-middle attacks and data tampering. Therefore, ensuring data security is crucial when deploying synchronization schemes in edge-AI systems. Moreover, Algorithmic Complexity: Some synchronization schemes can be computationally expensive, requiring significant processing power, which may not be available on edge devices. Therefore, the choice of synchronization algorithm needs to be carefully considered to ensure that it can be executed efficiently on edge devices. Finally the deploying fast synchronization schemes in real edge-AI systems requires careful consideration of the above challenges to ensure successful implementation

Table 4.2 Comparison with related works of Fast Sync. Schemes (Reproduced)

Metric	Fast_Sync	ASP	BSP	SSP	DSSP
Straggler mitigation	Yes, using clustering, quorum and late notification	No	No	Yes, using fixed bounded staleness	Yes, using flexible bounded staleness
Message overhead	Low	Very low	Very high	Moderate	High
Sync slack	Not allowed	Allowed	Bounded	Flexible but bounded	
Adaptable to dynamic systems	Highly adaptable	Adaptable	Not adaptable	Adaptable	Adaptable

Distribution of Time and Executions: - Using checkpoints included into the software, the controller keeps track of the employees' execution progress. The device that controls generates distributes from the two clusters' most recent app task executions for the task's estimated completion time (nish times) for each cluster before to the sync point. A mixed distribution of two Gaussian distributions is used to represent each cluster, as we may clarify / stated the statements and for the sake of clarifying much better, as special symbols were needed hence, I used the image version to represent the distribution of execution times by showing/representing special symbols as under:-:-

The first distribution, $\mathbb{D}_{early} = G(\mu_{ea}, \sigma^2_{ea})$ represents the early execution times distribution of the cluster while the second distribution, $\mathbb{D}_{late} = G(\mu_{la}, \sigma^2_{la})$ represents the late execution times distribution of the cluster. We assume that the distribution of the execution times of local tasks on workers in both clusters are known. The distribution is a mixture of models, and is defined as $\mathbb{D}^{lo}_{early} = G(\mu_{lo\_ea}, \sigma^2_{lo\_ea})$ and $\mathbb{D}^{lo}_{late} = G(\mu_{lo\_la}, \sigma^2_{lo\_la})$.

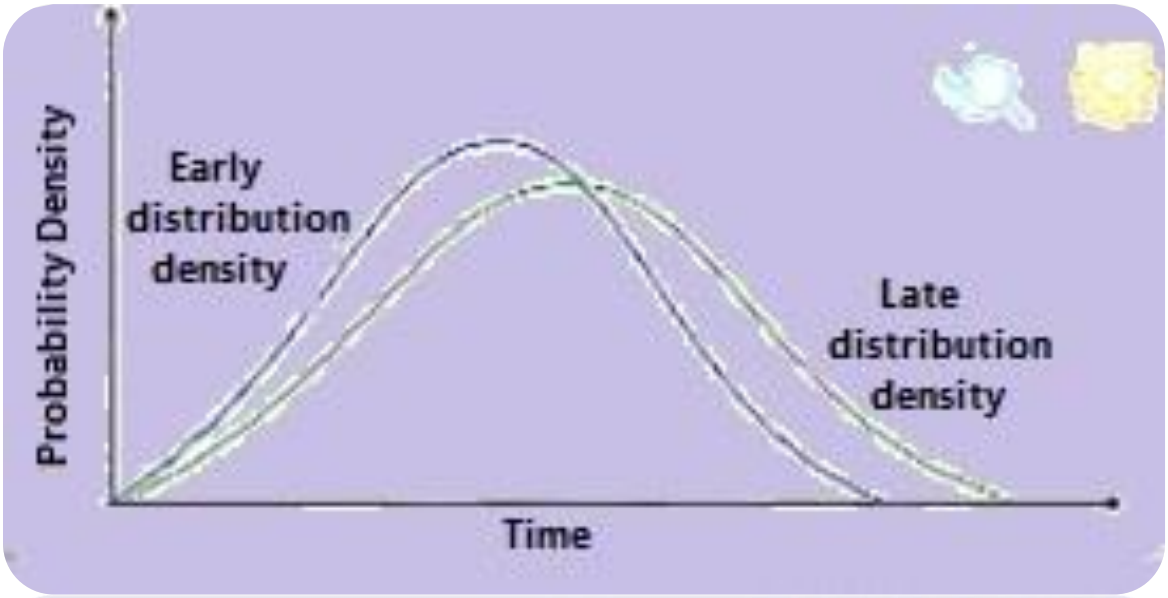

Figure 4.5 Showing Distributions (late and early), model mixture of cluster

Moreover, late notification protocol uses there, a late notification protocol is a type of protocol used in distributed systems to handle situations where a node receives a message or an update after the event has occurred. The protocol is designed to ensure that the node receives the update and can handle it appropriately, even though it may have missed the original event. The late notification protocol typically involves the following steps: Time stamping: All messages and updates are time stamped when they are generated. This ensures that each node has an accurate record of when events occurred. Broadcasting: When a node generates a message or update, it broadcasts it to all other nodes in the system. This ensures that all nodes receive the message or update at roughly the same time. Buffering: An update or message is received by a node, to find out if it missed the event, it examines the timestamp. If the node has missed the event, it stores the message or update in a buffer until it can be processed. Late Notification: Once the node is ready to process the buffered messages or updates, it notifies other nodes of its readiness to receive late notifications. Other nodes can then send any messages or updates that the node missed. Processing: The node processes the buffered messages or updates and any late notifications it receives. The late notification protocol is useful in situations where nodes may be temporarily disconnected from the network, or when messages or updates are generated in rapid succession, making it difficult for all nodes to receive them in real-time. By time stamping and buffering messages and updates, the late notification protocol ensures that all nodes eventually receive the updates and can process them correctly. The function of LNP in our work is also defined as:-

In our system, clustering the workers has the primary benefit of lowering the message overhead needed to achieve synchronizations. Therefore, it is assumed that workers in a cluster would maintain synchronizations and use the same synchronizations choices. Communications is necessary for the late plans, which entails sending late alerts to let neighboring clusters know when something is late. We create a late alert system that requires three

messages to be received by workers in a specific cluster in order to limit the amount of messages needed to label a cluster as late. According to the cluster's predicted finish time for the present task, after 50% of the execution is complete, we presume that workers can predict when they will be late. Workers in other cluster receive a late notification from main employee in It will take a while for the cluster to realize it. We expect to receive it if half the employees are late because the likelihood of receiving a second late notification after the first is 2=N for each late person. The likelihood of getting a third late notification for each tardy employee equals (1=N). Therefore, we expect to get a total of 3 late notices if every worker in a cluster is running late. In the event that network partitioning causes the loss of late notification broadcast messages, a cluster may become trapped in the at sync option. Network segmentation may result in partial or total isolation. While a worker who is entirely isolated is completely separated by other system employees, an isolated worker only loses connectivity for one or a few iterations. Therefore, incorporate prior to confirm security in the case of future late notifications and brief isolation. So, the initial notice of lateness will be included in the second inadequate warning. Employees in an alternative cluster will receive both of the late notices that are offered in the subsequent email if the first notice of lateness is lost as a result of network partitioning. When a worker is entirely disconnected, it will synchronize on its own until it is reconnected.

Moreover, in order to achieve minimal exchange messages which could be light weight and scalable hence, understanding the proposed synchronizations approach requires a description of a typical sync is used in a two-way message exchange between two nodes. The 2-way message exchange between the two nodes serves as the main structural element of the synchronizations process. This leads us to believe that within the limited span of one communication interchange, between two nodes, the clock drifting is constant. Additionally, the proliferation delay is thought to be constant in both directions. In Fig. 4.5, pay attention to the two-way communication flow between nodes A and B. By delivering a synchronizations message at time t1 according to the the node's local time, Node A starts synchronizations. This message contains information about a as well as the value of t 1. At time t 2, which may be computed as, B gets this message, below is the written statement of the same.

$$t_2 = t_1 + \Delta + d$$

Where the relative clock drift between the nodes is and d is the propagation delay of the pulse. At time t3, B responds with an acknowledgement that includes information about himself, the times t1, t2, and t3, as well as the response. After that, node A may synchronize with node B by calculating the propagation delay and clock drift as illustrated below in the written form statements (2) & (3) .

$$\Delta = ((t_2 - t_1) - (t_4 - t_3))/2 \quad \textbf{(2)}$$

$$d = ((t_2 - t_1) + (t_4 - t_3))/2 \quad \textbf{(3)}$$

The synchronization the phases are initiated through the syn beginning signal sent by the root node. Level 1 nodes start a two-way message exchange with the root node after receiving this message. It reduce collisions on wireless channel, each node waits for a different amount of time before the data exchange starts. They synchronize their clocks with the main the nodes after receiving an acknowledgment from it. After awaiting an arbitrary amount of time to make sure that level 1 nodes have finished their synchronizations, level 2 nodes start a two-way exchange of messages with a level 1 node they observed speaking to the root. Finally, the root node and every the additional nodes stay in sync as a result of this approach.

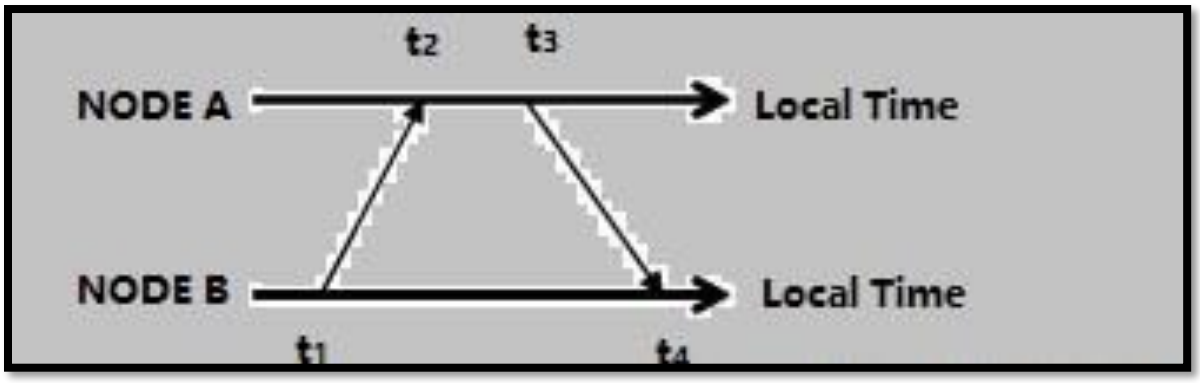

Figure 4.6 Showing two way communication in nodes pair.

A network architecture built on a tree structure must be developed before the sensors can be brought into sync. Each sensor node employs the suggested Algorithm 1 to effectively flood the network and create a logical hierarchical structure from a designated source point. Each sensor must first be configured to accept flood packets (fd_pckt) in order to broadcast the fd_pckt along with the node identity and level. A node must first identify its parent node as the source of the broadcast before setting the current recipient node's level to a number that is one higher than that parent node.

```
Begin
    Accept (fd_pckts)
    Initialize : no_reciever - 0;
    Node_Level(Root)-0;
    If (current_reciever - - root)
         Broadcast (fd_pckts)
     Else if (current_reciever != root)
         Begin
              Accept (fd_pckts);
              Parent(curent_reciever) - Source(broadcast_msg);
              Node_Level(curent_reciever)-Node_Level(Parent)+1;
              Broadcast (ack_pckt, node_id);

               Ignore (fd_pckts);
         End
           Else if (current_node receives ack_pckt)
                no_ receiver++;
End
```

Figure 4.7A Showing straightforward steps of hierarchy level

As illustrated HTS in Fig. 4.6A consists of three straightforward steps that are carried out at every level of the hierarchy. Furthermore, the figures from 4.6 B to 4.6 E much clarify the positions of the nodes and figure 4.6 B on the control channel, the Reference Network (RN) first sends out a beacon. (See Fig. 4.6 B). The specified clock channels will be moved to by one of the reference node's designated

offspring, who will then react there (Fig. 4.6 C). To synchronize the initial wave of nodes that are under it surrounding the reference node, the RN will compute the clock offset and broadcast it to all offspring nodes (Fig. 4.6 D). Further away from the reference node in the hierarchy, this procedure can be repeated (Fig. 4.6 E). The following is a more thorough explanation of the HTS scheme:

Step 1: RN initiates the synchronization by broadcasting the syn_begin message with time t1 using the control channel and then jumps to the clock channel. All concerned nodes record the received time of the message announcement. RN randomly specifies one of its children, e.g. SN2, in the announcement. The node SN2 jumps to the specified clock channel.

Step 2: At time t3, SN2 replies to the RN with its received times t2 and t3.

Step 3.1: RN now contains all time stamps from t1 to t4. It calculates clock drift Δ and propagation delay d, as per equation (2) and (3), and calculate t2 = t1 + Δ + d, and then broadcasts it on the control channel.

Step 3.2: All involved neighbor nodes, (SN2, SN3, SN4 and SN5) compare the time t2 with their received timestamp t2′.

i.e. SN3 calculates the offset d′ as:

$$d' = t2 - t2'$$

Finally, the time on SN3 is corrected as:

$$T = t + d + d'$$

Where t is the local clock reading.

Step 4: SN2, SN3, SN4 and SN5 initiate the syn_begin to their downstream nodes.

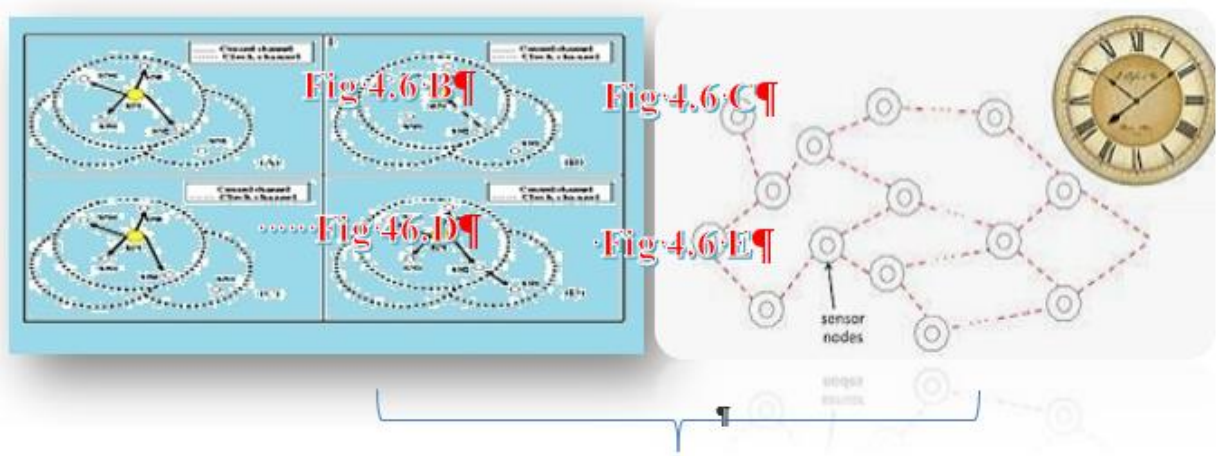

Fig 4.7F Process Sync. Sensor node

Joint Figures 4.7B to 4.6E shown in Blue Leave
4.7 B, Broadcast of Reference nodes,
4.7 C, reply of neighbor
4.7 D, synchronized all neighbor
4.7 E, repetition at lower layers

When a sensor node starts the synchronizations process, we presume that it is aware of its neighbors. The notification in Step 1 specifies the responsibility from the node. Only this one node has access to the time channels designated by the RN. They are currently awaiting the second update from the RN the other nodes can conduct regular data transmission without being bothered by the synchronizations dialogue between RN and SN2, further the figure 4.6 F defines much more regarding the synchronization of sensor node. Let's come to again on path, when the syn begin message is sent, the RN starts a timer. The RN returns of standard controls of channel once the clock ends in the event that The communication is misplaced on the way to SN2, ending the waiting period. A multi-level hierarchy is dynamically built as the synchronizations ripple propagates from the reference node to the remainder of the network. Depending on how many hops there are between a node and the reference node, or the central reference point, that node is assigned a level. The synchronizations is repeated by the nodes phases from root to leaves at each level as described above, starting from the reference nodes.

The network assesses the total number of synchronizations devices in order to reduce overall synchronizations energy usage of synchronization-related message exchanges that took place during the most recent synchronizations period. It then adapts the duration of the subsequent synchronizations period. Such as adopting the sensor initiated (SI) mode rather than the always on (AO) mode is a superior alternative to conserve network resources when network traffic is infrequent and synchronizations latency is not a serious issue. In some situations, the sensor clocks may also be permitted to desynchronize if no sensing events take place. Counting the necessary time message exchanges (beacons) for each pair-wise synchronizations is a crucial issue. We take into account a variety of variables to determine network characteristics, include the pair-wise synchronizations parameters N, synchronizations method, sync interval, and numbers of beacon in order it handle these design issues. In fact, it strives for effective synchronizations through the use of network resources (message exchanges). A crucial factor in determining the pair-wise synchronization's accuracy and power efficiency is the amount of timing messages (beacons) per pair. Assume that the two nodes' clock timing discrepancy is modelled as follows: When t is used as the reference time, the formula is: = o + st, here o and s, respectively, stand for the clock offset and skew errors. When two nodes exchange i messages, clock offset and skew estimation errors take place. These mistakes are identified as o,i and s,i, respectively. In general, it is challenging to pinpoint a particular mathematical model for clock skew or offset issues.

The TSRT protocol takes a benefit of using wireless technology broadcast capabilities to establish a single point of consensus among all nearby peers on the time of arrival of the broadcast message. This shared reference point may be utilized to accomplish synchronization in Step 4 , meaning that t2 at node SN2 and at node SN3, t2' happened simultaneously. (RN)'s first broadcast to When the RN synchronizes with SN2, The other nearby nodes may hear SN2 and its previous update alerting SN2 of its offset d2. If the RN includes the time t2 in the update sent to SN2 (redundant for SN2), all neighbors can synchronize. It would make sense for SN3 to obtain its offset from SN2's local clock from overhearing t2, and its offset from SN2's local clock to the RN reference clock from overhearing d2, as t2 and t2' occurred concurrently. Since there is only one little overhead transaction every hop between a parent node and all of its offspring, TSRT is incredibly scalable and lightweight. As a result, just three communications are needed for SN3 and all RN children to calculate their own offsets from the RN reference clock: two control broadcasts and one clock channel reply. Instead of synchronization taking place among all of a core node's neighbors, pair verification occurs between two neighbors in RBS.

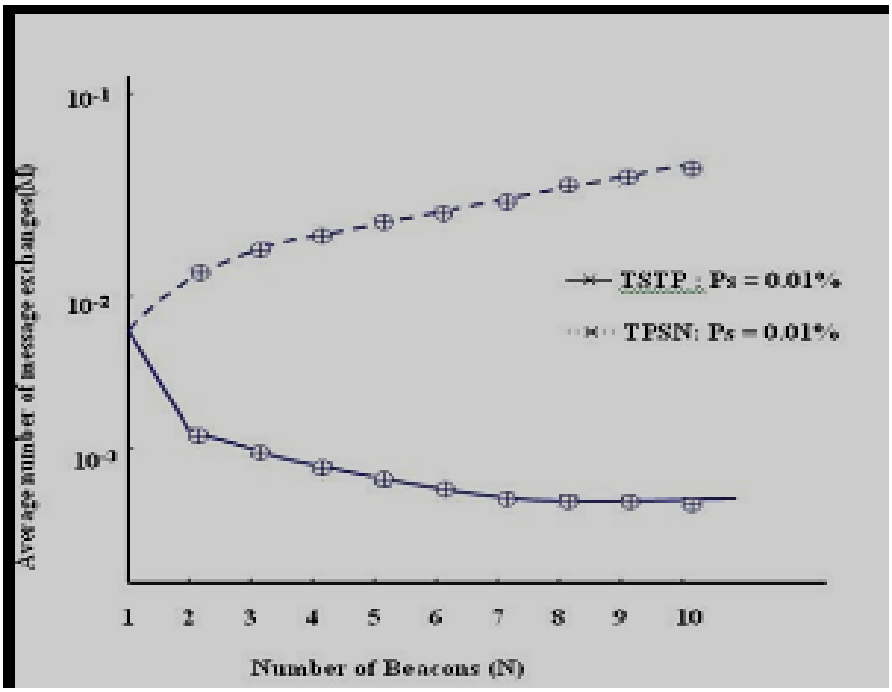

Figure 4.8 Showing No. of Avg. Msgs (M), PS=0.01, %

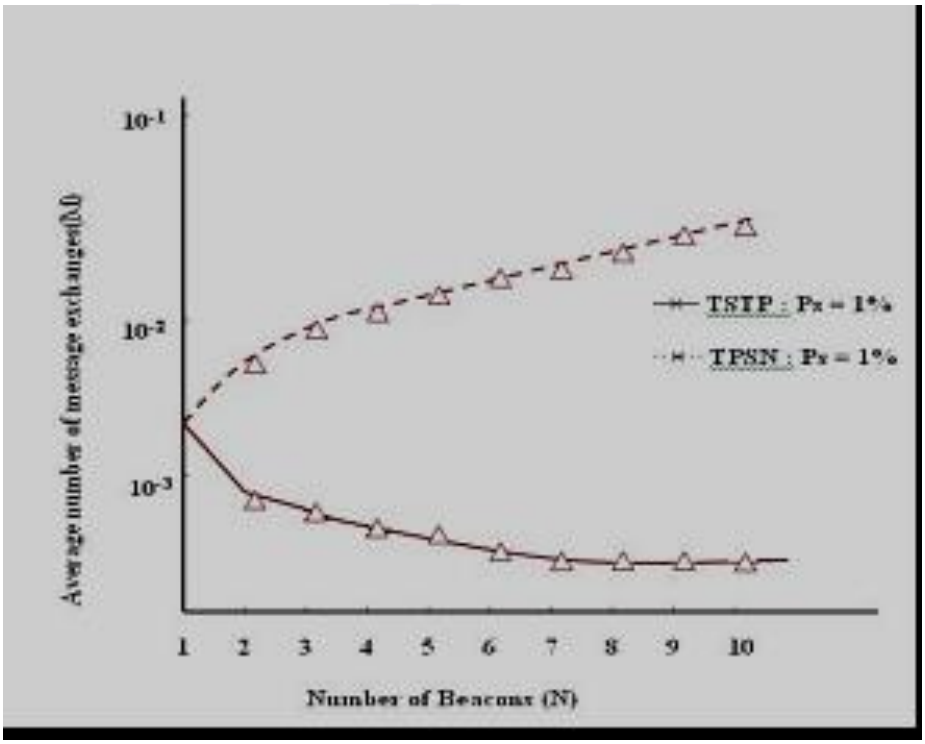

Figure 4.9 Showing No. of Avg. Msgs (M), PS=1.0, %

When several beacon broadcasts are necessary, it can be shown that TSRT requires fewer timing messages than TPSN. Additionally, as N rises, the average number of timing messages required by TPSN and TSRT varies increasingly, making TSTP much more efficient than TPSN. Additionally, it tends to be seen

that a humble number of guides is satisfactory to bring down M for TSRT. To adhere to a stricter network-wide protocol wide error probability Ps restriction, more beacons are required, as anticipated. Since the synchronizations time is related to the number of N, a smaller N leads to better latency performance, which is highly preferred in practice. However, it might not be the most energy-efficient option.

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

This research focuses on synchronization strategies for Edge and IoT intelligence applications, proposing two dynamic synchronization systems using time-based and component-based redundancies improving fault tolerance. Task-focused and redundancy-based synchronization techniques are evaluated against existing approaches.

Applications requiring device-level redundancy unable to wait for task execution should use component-based redundancies. Applications allowing repeated task execution benefit from time-based redundancy. Keeping controller informed about task completion proves crucial—publish-subscribe update methods reduce controller interaction, enabling more efficient asynchronous task completion.

Results show increasing component-based redundancy reduces runtime and unsuccessful synchronization percentage. Worker task completion time predictability significantly impacts runtime and synchronization task failure. Recommended redundancy-based algorithms perform faster with fewer failures compared to barrier and time-slotted synchronization.

Simulation study evaluates rapid synchronization method benefits, demonstrating efficient operation as worker heterogeneity and density increase. The suggested approach outperforms or matches SSP and BSP approaches considering different staleness levels. Quick synchronization method assessed within Ray Python framework by comparison with ASP, SSP, and BSP principles in various clustered configurations.

Wireless Sensor Network advantages in tracking item movement and ambient characteristics are discussed, emphasizing synchronization necessity for optimal results. The TSRT synchronization method delivers deterministic synchronization with few pair-wise message exchanges, exceptionally well-suited for WSNs requiring high-accuracy synchronization with constrained processing and bandwidth.

The proposed method minimizes power consumption switching between RBS and TPSN, enabling distributed sensor synchronization within milliseconds. Theoretical performance demonstrates advantages over existing procedures. Due to fixed broadcast messages within single broadcast domain, the synchronization technique proves more lightweight than RBS and TPSN.

6.2 Recommendations

Future work priorities in fog-controlled IoT include handling device mobility across fogs. Vehicular clouds present vehicle movement entering and exiting various fog zones. Synchronization schedulers must manage vehicle-to-fog relationships to reduce mobility-induced synchronization task failures.

Prediction models forecasting node availability in dynamically changing networks will make quorum checks unnecessary. Expanding micro-batching concepts to include devices as well as tasks achieves more localization.

While synchronization techniques in game models have been widely studied, this work develops WSN models investigating rapid synchronization strategies considering current methodologies like game models. Machine learning techniques can anticipate task completion times for improved synchronization planning.

Real-world testing will gauge proposed job synchronization solution performance, examining actual network conditions, node connections, communication lags, mobility patterns, faults, and failures. Game-theoretic synchronization technique will expand to accommodate more than two clusters, producing finely tuned clusters with higher close synchronization probability.

Full synchronization method integration into artificial intelligence application frameworks and operations programming languages will enable real-world testing and efficacy confirmation.

REFERENCES

Ameya, 2020, Application of synchronized clocks in Distributed Systems ,https://medium.com/@ameya_s/application-of-synchronized-clocks-in-istributed-systems-9d630ba1cb56

Amiri, M. M., & Gündüz, D. (2020) Machine learning at the wireless edge: Distributed stochastic gradient descent over-the-air. *IEEE Transactions on Signal Processing*, *68*, 2155-2169.

Bogdan-Martin, D. (2019) Measuring digital development: Facts and figures 2019. In *Technical report*. International Telecommunications Union (ITU).

Dalwadi, N. N., & Padole, C. M. (2017) Comparative study of clock synchronization algorithms in distributed systems. *Advances in Computational Sciences and Technology*, *10*(6), 1941-1952.

Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S., & Zomaya, A. Y. (2020) Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, *7*(8), 7457-7469.

Fan, K., Sun, S., Yan, Z., Pan, Q., Li, H., & Yang, Y. (2019) A blockchain-based clock synchronization scheme in IoT. *Future Generation Computer Systems*, *101*, 524-533.

Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., & Vahdat, A. (2018) Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (pp. 81-94).

Harris, T., Maas, M., & Marathe, V. J. (2017, April) Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems* (pp. 1-14).

C. Ren, K. He, X. Zhang , and J. Sun, (2021) "Synchronization Mechanisms for Distributed Edge Intelligence," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Jararweh, Y., Doulat, A., AlQudah, O., Ahmed, E., Al-Ayyoub, M., & Benkhelifa, E. (2016, May) The future of mobile cloud computing: integrating cloudlets and mobile edge computing. In *2016 23rd International conference on telecommunications (ICT)* (pp. 1-5). IEEE.

Jia, P., Wang, X., & Zheng, K. (2019) Distributed clock synchronization based on intelligent clustering in local area industrial IoT systems. *IEEE Transactions on Industrial Informatics*, *16*(6), 3697-3707.

J. Wang, J. Zhang, H. Tu, Y. Ren, J. Wan, L. and Zhou, M. Li, (2018) "An adaptive synchronous parallel strategy for distributed machine learning," *IEEE Access*, vol. 6, pp. 19222–19230.

Keiji Ozaki, Kenichi Watanabe, (2018) A Fault-Tolerant Model of Wireless Sensor-Actuator Network.

Krishnamurthi, R., Kumar, A., Gopinathan, D., Nayyar, A., & Qureshi, B. (2020) An overview of IoT sensor data processing, fusion, and analysis techniques. *Sensors*, *20*(21), 6076.L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805.

Lian, X., Zhang, W., Zhang, C., & Liu, J. (2018, July) Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning* (pp. 3043-3052). PMLR.

Lovén, L., Leppänen, T., Peltonen, E., Partala, J., Harjula, E., Porambage, P., ... & Riekki, J. (2019) EdgeAI: A vision for distributed, edge-native artificial intelligence in future 6G networks. *The 1st 6G wireless summit*, 1-2.

Lueth, K. L. (2020) Iot 2019 in review: The 10 most relevant iot developments of the year. *IoT Analytics, January*.

Mani, S. K., Durairajan, R., Barford, P., & Sommers, J. (2018) A system for clock synchronization in an internet of things. *arXiv preprint arXiv:1806.02474*.

Majid, M., Habib, S., Javed, A. R., Rizwan, M., Srivastava, G., Gadekallu, T. R., & Lin, J. C. W. (2022) Applications of wireless sensor networks and internet of things frameworks in the industry revolution 4.0: A systematic literature review. *Sensors*, *22*(6), 2087.

Noh, K. L., & Serpedin, E. (2007, February) Adaptive multi-hop timings synchronization for wireless sensor networks. In *2007 9th International Symposium on Signal Processing and Its Applications* (pp. 1-6). IEEE.

Olaniyan, R., & Maheswaran, M. (2019) Multipoint synchronization for fog-controlled Internet of Things. *IEEE Internet of Things Journal*, *6*(6), 9656-9667.

Papazachos, Z. C., & Karatza, H. D. (2011) Gang scheduling in multi-core clusters implementing migrations. *Future Generation Computer Systems*, *27*(8), 1153-1165.

P. Jia, X. Wang, and K. Zheng, (2019) "Distributed clock synchronization based on intelligent clustering in local area industrial iot systems," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 3697–3707.

Perifanis, N. A., & Kitsios, F. (2022) Edge and Fog Computing Business Value Streams through IoT Solutions: A Literature Review for Strategic Implementation. *Information*, *13*(9), 427.

Rahamatkar, S., & Agarwal, D. (2011) A reference based, tree structured time synchronization approach and its analysis in WSN. *arXiv preprint arXiv:1103.4905*.

Ray, P. P. (2018) A survey on Internet of Things architectures. *Journal of King Saud University-Computer and Information Sciences*, *30*(3), 291-319.

Siddique, K., Akhtar, Z., Yoon, E. J., Jeong, Y. S., Dasgupta, D., & Kim, Y. (2016). Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications. *IEEE Access*, *4*, 8879-8887.

S. Wang, S. Raza, M. Ahmed, and M. R. Anwar, (2019) "A survey on vehicular edge computing: architecture, applications, technical issues, and future directions," *Wireless Communications and Mobile Computing*, vol. 2019.

Steiner, W., Bonomi, F., & Kopetz, H. (2014, March) Towards synchronous deterministic channels for the internet of things. In *2014 IEEE world forum on Internet of Things (WF-IoT)* (pp. 433-436). IEEE.

S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, (2019) "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716.

Sun, X., Su, Y., Huang, Y., Tan, J., Yi, J., Hu, T., & Zhu, L. (2020) Edge computing-based ERBS time synchronization algorithm in WSNs. *Wireless Communications and Mobile Computing*, *2020*, 1-11.

Sun, Y., Jiang, Q., & Zhang, K. (2012, September) A clustering scheme for reachback firefly synchronicity in wireless sensor networks. In *2012 3rd IEEE International Conference on Network Infrastructure and Digital Content* (pp. 27-31). IEEE.

E. Serpedin, Y.-C. Wu and Q. Chaudhari (2011) "Clock synchronization of wireless sensor networks," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 124– 138.

Wang, T., Zhou, J., Liu, A., Bhuiyan, M. Z. A., Wang, G., & Jia, W. (2018) Fog-based computing and storage offloading for data synchronization in IoT. *IEEE Internet of Things Journal*, *6*(3), 4272-4282.

Wang, Y., Núñez, F., & Doyle, F. J. (2013) Statistical analysis of the pulse-coupled synchronization strategy for wireless sensor networks. *IEEE transactions on signal processing*, *61*(21), 5193-5204.

W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, (2016) "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646.

Xie, K., Cai, Q., & Fu, M. (2018) A fast clock synchronization algorithm for wireless sensor networks. *Automatica*, *92*, 133-142.

X. Lian, W. Zhang, C. Zhang, and J. Liu, (2018) "Asynchronous decentralized parallel stochastic gradient descent," in International Conference on Machine Learning, pp. 3043–3052, PMLR.

Yadav, P., McCann, J. A., & Pereira, T. (2017) Self-synchronization in duty-cycled internet of things (iot) applications. to appear in IEEE Internet of Things Journal. *arXiv preprint arXiv:1707.09315*.

Y. Sun, Q. Jiang, and K. Zhang, (2012) "A clustering scheme for reachback firefly synchronicity in wireless sensor networks," *in Network Infrastructure and Digital Content (IC-NIDC), 2012 3rd IEEE International Conference on*, pp. 27–31, IEEE.

Z. C. Papazachos and H. D. Karatza, (2011) "Gang scheduling in multi-core clusters implementing migrations," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1153–1165.