Synchronization Schemes for Internet of Things and Edge Intelligence Applications

Richard Olaniyan

School of Computer Science
McGill University
Montreal, Canada

September 2021

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Dedication

Dedicated to my parents; Mr Olawole Olaniyan and Mrs Comfort Olaniyan, my siblings; Helen, Olamide and Olatomi and my better half; Ololade, without whose support and encouragement none of this would have been possible. Also, to the entire McGill research community, who with their hard work, make this world a better place!

Acknowledgements

This research was sponsored in part by a Nigerian Presidential Special Scholarship Scheme for Innovation and Development (PRESSID)/Petroleum Technology Development Fund (PTDF), a Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant, a Grad Excellence Award in Computer Science, McGill University and an Ericsson/MITACS Accelerate grant. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the sponsors.

I would like to express my heartfelt gratitude to my supervisor Prof. Muthucumaru Maheswaran for his patience, motivation, enthusiasm, and profound knowledge. His continuous guidance and feedback helped me to improve the quality of my research and thesis writing. I would also like to thank my committee members Jörg Kienzle and Clark Verbrugge for evaluating my thesis and providing their invaluable comments and feedback.

I thank my fellow lab mates in *Advanced Network Research Lab*: Olamilekan Fadahunsi, Richboy Echomgbe, Ben Yu, Jianhua Li, Xiru Xu and Tianzi Li for their insightful inputs and stimulating discussions, and for all the fun we have had together. I am thankful to the *School of Computer Science, McGill University* for all facilities and resources that they provided to make my learning and research effective and interesting.

Last but not the least, I would like to thank my family for supporting me through the thicks and thins of the past years.

Abstract

Devices controlled by cloud or edge resident coordinators are becoming an important trend for creating Internet of Things (IoT) systems and smart systems. The cloud provides a global perspective, while the edge provides low latency and localized service to the devices. Coordinating these devices to work collectively to solve problems with strict timing requirements in the presence of disconnections is a challenge. An important coordination task is to have devices perform the same action at the same point in time. With the large number of devices and data being generated by millions of edge devices, there is a need for a synchronization scheme to orchestrate the actions of multiple devices such that they can line up their start times for tasks that require strict coordination. Clock synchronization is necessary but not sufficient for such a system. A synchronization scheme for an edge-based IoT system needs to handle issues such as network disconnections, faults, failures, and mobility, all of which are attributes expected from an edge-based system.

With the recent intersection of edge computing and artificial intelligence (AI) applications, a new Edge AI paradigm has sufficed. Real-time AI applications mapped on edge computing need to perform data capture, data processing/intelligence extraction and device actuation within some tolerated time bounds. Synchronization across devices is an important problem that needs to be solved at the different stages of an AI application. Synchronized data capture reduces the amount of time required in preprocessing data (data aggregation, data cleaning, missing data handling, etc.). In the data processing phase, synchronization is key in ensuring convergence, accuracy and speed of the distributed training process across multiple edge devices. The actuation phase in some cases requires some actions to be performed at the same time at different devices, thus, the need for synchronization.

To solve the problem of synchronization in edge-based IoT, we propose three task-based and two redundancy-based algorithms. We present a tree-hierarchical architecture with controllers at different levels and devices at the leaf for synchronizing the operations of large collections of Internet of Things (IoT) such as drones, Internet of Vehicles, etc. Tasks are differentiated into three categories - synchronous (remote call with strict timing requirements), asynchronous (remote call with relaxed timing requirement) and local (self-call of a device on itself with relaxed timing requirement) depending on the execution mode and requirements. We evaluate the performance of the algorithms using trace-driven simulations and compare them to existing solutions. We identify the specific IoT application domains and runtime conditions in which each algorithm adapts best.

We further propose a fast edge-based synchronization scheme that can time align the execution of input-output tasks as well as compute tasks. The primary idea of the fast synchronizer is to cluster devices into groups that are highly synchronized in their task execution and statically determine synchronization points using a game-theoretic solver. The cluster of devices uses a late notification protocol to select the best point among the pre-computed synchronization points to reach a time aligned task execution as quickly as possible. We evaluate the performance of our synchronization scheme using trace-driven simulations as well as an implementation in Ray - a Python framework for programming distributed applications, and we compare the performance with existing distributed synchronization schemes for real-time AI application tasks. We show that our fast synchronizer delivers significant performance improvements over existing solutions.

Résumé

Les appareils contrôlés par les coordinateurs résidents du cloud ou de périphérie deviennent une tendance importante pour la création de systèmes Internet des objets (IoT) et de systèmes intelligents. Le cloud offre une perspective globale tandis que la périphérie fournit une faible latence et un service localisé aux appareils. Coordonner ces dispositifs pour qu'ils fonctionnent collectivement pour résoudre des problèmes avec des exigences de synchronisation strictes en présence de déconnexions est un défi. Une tâche de coordination importante consiste à demander aux appareils d'effectuer la même action au même moment. Avec le grand nombre d'appareils et de données générés par des millions d'appareils de périphérie, il est nécessaire de disposer d'un schéma de synchronisation pour orchestrer les actions de plusieurs appareils afin qu'ils puissent aligner leurs heures de début pour les tâches qui nécessitent une coordination stricte. La synchronisation d'horloge est nécessaire, mais pas suffisante pour un tel système. Un schéma de synchronisation pour un système IoT basé sur la périphérie doit gérer des problèmes tels que les déconnexions réseau, les pannes, les pannes et la mobilité, qui sont tous des attributs attendus d'un système basé sur la périphérie.

Avec l'intersection récente des applications de Edge computing et d'intelligence artificielle (IA), un nouveau paradigme Edge AI a suffi. Les applications d'IA en temps réel mappées à l'informatique de pointe doivent effectuer la capture de données, le traitement des données / l'extraction de l'intelligence et l'activation de l'appareil dans des limites de temps données. La synchronisation entre les appareils est un problème important qui doit être résolu aux différentes étapes d'une application d'IA. La capture de données synchronisée réduit le temps nécessaire au prétraitement des données (agrégation de données, nettoyage des données, traitement des données manquantes, etc.). Dans la phase de traitement des données, la synchronisation est essentielle pour garantir la convergence, la précision et la vitesse du processus de formation distribué sur plusieurs périphériques de périphérie. La phase d'actionnement dans certains cas nécessite que certaines actions soient effectuées en même temps sur différents appareils, d'où la nécessité d'une synchronisation.

Pour résoudre le problème de la synchronisation dans l'IoT basé sur la périphérie, nous proposons trois algorithmes basés sur des tâches et deux algorithmes basés sur la redondance. Nous présentons une architecture arborescente avec des contrôleurs à différents niveaux et des dispositifs à la feuille pour synchroniser les opérations de grandes collections d'Internet des objets (IoT) tels que les drones, l'Internet des véhicules, etc. Les tâches sont différenciées en trois catégories - synchrones (appel à distance avec exigence de synchronisation stricte), asynchrone (appel à distance avec exigence de synchronisation assouplie) et local (appel automatique d'un appareil sur lui-même avec exigence de synchronisation assouplie) en fonction du mode d'exécution et des exigences. Nous évaluons les performances des algorithmes à l'aide de simulations basées sur les traces et les comparons aux solutions existantes. Nous identifions les domaines d'application IoT spécifiques et les conditions d'exécution dans lesquels chaque algorithme s'adapte le mieux.

Nous proposons en outre un schéma de synchronisation rapide basé sur les bords qui peut aligner dans le temps l'exécution des tâches d'entrée-sortie ainsi que des tâches de calcul. L'idée principale du synchroniseur rapide est de regrouper les périphériques en groupes hautement synchronisés dans leurs exécutions de tâches et de déterminer statiquement quelques points de synchronisation à l'aide d'un solveur théorique des jeux. Le cluster d'appareils utilise un protocole de notification tardive pour sélectionner le meilleur point parmi les points de synchronisation précalculés afin d'atteindre une exécution de tâche alignée dans le temps aussi rapidement que possible. Nous évaluons les performances de notre schéma de synchronisation à l'aide de simulations pilotées par trace ainsi que d'une implémentation dans Ray - un framework Python pour la programmation d'applications distribuées, et nous comparons les performances avec les schémas de synchronisation distribués existants pour les tâches d'application d'IA en temps réel. Nous montrons que notre synchroniseur rapide offre des améliorations de performances significatives par rapport aux solutions existantes.

Contents

List of Figures

List of Tables

List of Publications and Patent

Richard Olaniyan, Olamilekan Fadahunsi, Muthucumaru Maheswaran, and Mohamed Faten Zhani. Opportunistic edge computing: concepts, opportunities and research challenges. *Future Generation Computer Systems*, 89:633–645, 2018.

Richard Olaniyan and Muthucumaru Maheswaran. Synchronous scheduling algorithms for edge coordinated internet of things. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2018.

Richard Olaniyan and Muthucumaru Maheswaran. Multipoint synchronization for fog-controlled internet of things. *IEEE Internet of Things Journal*, 6(6):9656–9667, 2019.

Richard Olaniyan and Muthucumaru Maheswaran. A fast edge-based synchronizer for tasks in real-time artificial intelligence applications. *IEEE Internet of Things Journal*, 2021.

Richard Olaniyan, Muthucumaru Maheswaran, Emmanuel Thepie Fapi, Manoj Kopparambil Nambiar, and Bassant Selim. Edge device, edge server and synchronization thereof for improving distributed training of an artificial intelligence (ai) model in an ai system. (Application No. 63/151335) United States Patent and Trademark Office, 2021.

Chapter 1

Introduction

1.1 Overview

Tightening environmental regulations and the need for meeting the demands of increasing population are driving the need for smarter factories, intelligent transport systems, smart buildings, and smart cities. The Industrial Internet of Things (IIoT) [1, 2, 3] and many academic research projects (e.g., TerraSwarm [4, 5]) have proposed paradigms for achieving this objective by connecting the machines within these systems to fog-based controllers.

Internet of Things (IoT) [6, 7, 8, 9, 10] come in many shapes and sizes. Because each IoT component will be handling a part of a problem domain, cooperative computing among the IoT components is essential for solving many interesting problems [11]. For example, the use of drones in activities such as rescue missions after natural disasters is increasing with recent advancements in drone technologies [12, 13, 14]. A synchronization mechanism is required to ensure that the autonomous drones capture a specified part of the search area at the same time. If the drones do not follow strict timing alignments, it will lead to a poor reconstruction of the search area. As another example, take an autonomous car, the car could form a swarm with other cars and roadside units.

To drive successfully, the car needs assurance that data is collected within a short time interval by the sensing and processing components of the swarm. The car thus needs support from the swarm to ensure that its tasks are run simultaneously across the different constituents.

Cooperating IoT [15, 16] need to run tasks of the application in a coordinated manner. One such coordination is running the tasks at the same time. For instance, drones lifting a workload need to exert the force at the same time so that the effort adds up. To run the tasks at the same time, there is a need for synchronization primitives that would launch the tasks with the start times lined up at all IoT devices. Earlier works on synchronization in IoT have focused on time/clock synchronization [17, 18, 19] where IoT devices have a common notion of time. That is, IoT devices maintain the same clock with minimal skew. Just having synchronized clocks is not sufficient because even with an earlier agreed start time, the IoT devices could fail, disconnect from the swarm [5], or be busy with a prior task. With tighter synchronization schemes, fine–grained tasks can be run across the cooperating IoT while guaranteeing the desired quality of synchronization (QoS_{ync}).

With the recent advances in fog computing [20, 21, 22], edge computing [23, 24, 25, 26] and 5G technologies [27, 28, 29] it is becoming increasingly possible to run data-intensive applications closer to the devices that generate data with the lowest possible latency (e.g., real–time video surveillance or tracking). IoT leverages fog computing by having IoT controllers resident at the edge of the network thus augmenting the computing capabilities of devices [30]. The services provided by IoT controllers can be quickly accessed by IoT devices such as mobile phones or vehicular nodes as they move around in different physical vicinities. Fog–controlled IoT is thus enabling the creation of a coalition of resources among fog-resident IoT controllers and IoT devices (swarm of machines [5]) that have larger computing capabilities than any of the individual constituents. There is therefore a need for new synchronization schemes and programming approaches to enable cooperation and coordination among the constituents of the swarm while reducing operating costs and energy consumption through efficient scheduling schemes.

In the first part of this thesis, we focus on exploring the synchronization space in fog–controlled IoT. We provide a definition and taxonomy for synchronization in IoT. We propose and evaluate several synchronization scheduling algorithms for strict simultaneous execution across a bunch of cooperating IoT nodes. The scheduling algorithms need to ensure that the required quorum of nodes is available before executing a synchronous task. The nodes do not start other tasks while waiting for the synchronization condition.

Edge computing likewise has opened new grounds for computations to be done at edge devices rather than shipping all computations to the cloud. With the increasing number of smart devices and sensors, a vast amount of data will continue to be generated. Gathering intelligence at the edge is a big challenge because convolutional neural networks (CNN) [31, 32] and deep neural networks (DNN) [33, 34] require thousands (or sometimes millions or billions) of interconnected units and parameters to train models [35, 36]. In order to extract real time intelligence from the huge volume of data generated, processing needs to be done at the edge close to where the data is generated [37, 38, 39]. However, most edge devices lack the computing capacity to train and run deep learning (DL) models in a timely manner [36].

A typical AI application can be broken down into three phases – sensing, processing, and actuating. Sensing involves data acquisition, preprocessing, and data cleansing. Processing involves training AI models and gathering intelligence from captured data. Actuating involves using the intelligence gathered to take further actions [40]. Artificial Intelligence (AI) applications are increasingly thriving with the advancements in deep learning technologies and processing technologies, leading to the large demand on processing and optimization [41, 42]. This demand has necessitated the need for integrating edge computing and AI [38]. AI on edge aims at extracting intelligence from distributed edge data generated by edge devices with the goal of improving overall performance [35].

Synchronization is a key issue both in the sensing phase: data capture and processing phase: training AI models. The quality of data captured can be greatly improved through synchronization while maintaining a high capture rate. Take for example, an edge-assisted autonomous driving system, the edge is heavily involved in data capture, aggregation, and processing with the goal of improving driving efficiency while ensuring safety. An autonomous driving system consists of several sensors and thus, data from multiple sensors must be combined [43]. Due to the real time and tight latency constraints required for seamless autonomous driving, intelligence must be gathered from acquired data in real time. Thus, AI model training and intelligence extracted from the data must be in a timely and coordinated manner to ensure safe and efficient driving.

Having data from multiple sources that are time aligned reduces the extra time spent in data aggregating and cleaning. Due to the vast number of devices that will be capturing and generating data, synchronizing the data capture process makes it easier to aggregate data from different sources and devices. It is also easier to identify and fill gaps in data, as well as maintaining appropriate time scales [44]. The success and accuracy of distributed training on the edge strictly depends on the synchronization scheme adopted in aggregating shared parameters or local updates. The parameter server framework [45, 46, 47, 48, 49] was proposed as a way of aggregating updates from distributed devices.

To time align the task executions, we need to delay the fast tasks, which can result in an overall throughput reduction. Instead of simply making the synchronization point fuzzy, we introduce a task scheduling method that creates alternative synchronization points by considering global tasks with synchronization requirements along with local tasks that need no synchronization. Many distributed AI/ML problems like federated learning at the edge can benefit from such an efficient synchronization scheme.

In the later part of this thesis, we focus on developing fast synchronization schemes for distributed data acquisition (sensing phase of an AI application) and distributed training on edge devices (processing phase of an AI application). We aim to minimize the number of messages and the communication overhead incurred in reaching synchronization among distributed edge devices. For distributed training, we aim to reduce training time (faster convergence) while maintaining high model accuracy.

1.2 Basic System Model

We employ a tree–based controller–worker computing model for interconnecting the components, where the cloud–level provides a global (bird–eye) perspective for controlling the components and processing global analytics. The fog–level components provide low latency service access and localization and are located between the cloud level and the device level. Tight clock synchronization is assumed across all levels in the tree by leveraging the hierarchical architecture and the recent advancements in clock synchronization schemes.

The controller–worker architectural model is inspired by models such as multi–robot systems [50, 51], Single Program Multiple Data (SPDM) [52] where tasks are split into parts and simultaneously scheduled on multiple processors with different input data to achieve parallelism and Software Defined Networking (SDN) [53, 54, 55] where programmable switches running the same application are coordinated by a central controller and fog computing.

1.2.1 Definition of Synchronization

First, we give a definition for synchronization in the context of this thesis. We define synchronization as the coordination of a set of nodes to harmonize on the execution of a task by time aligning the start of the execution. Thus, for a number of nodes to be considered *synchronized*, they need to have started the execution of a (synchronous) task at the same point in time.

1.2.2 Node Model

The three–level hierarchical model consisting of the cloud, fog, and device levels is shown in Fig. 1.1. Controllers can be resident at any of the three levels in the tree. Workers are at the leaves of the tree and only in the device level. This architecture is suitable for achieving synchronization in IoT because it permits worker nodes to join, leave, or move around, thus making changing membership from one controller to the other possible. The device–level and fog–level controllers provide localized and low latency services to the workers, which is necessary for reaching synchronization in IoT systems.

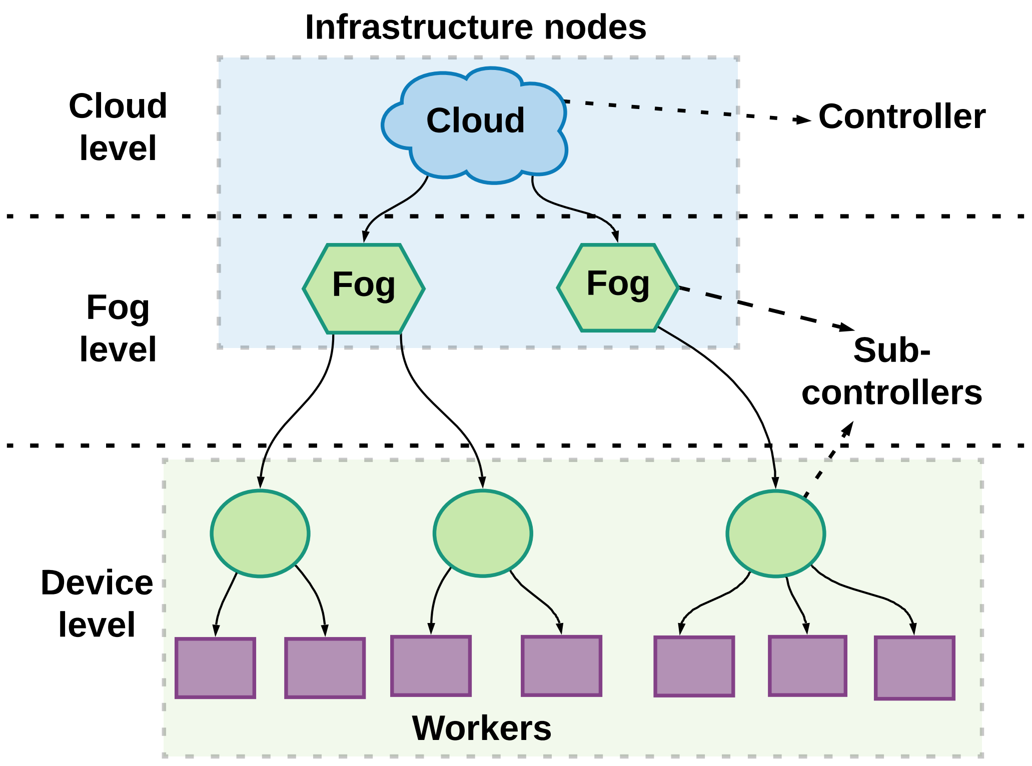


Fig. 1.1 Basic multi–level node architecture with cloud, fog and device levels.

Disconnection can occur between the workers and controllers, which can be due to mobility, network disruption or node failure. Disconnected workers can rejoin the system, or new workers join the system by connecting to a fog. Worker nodes are isolated from each other and thus, no direct link exists between them. Workers only communicate with the controller to which they are connected. Communication between a worker and controller is bidirectional, that is, data can flow both ways. The scope of the task being run determines what level of controller the workers communicate with. A global scope will require communication to exist between the main controller (cloud–level controller) and workers, either directly or through sub–controllers (fog and device level controllers). A nonglobal scope involves workers communicating with the fog–level controller, while a much more localized scope involves workers communicating and working with only the device–level controller.

Assumptions

1. We assume heterogeneous workers with different but similar computational and processing capabilities, and as such will have different execution times for the same task with some variations.
2. Workers follow a single–threaded execution model inspired by the JAMScript IoT programming language and runtime [56].
3. All the workers in the system are running the same application.
4. Workers can have faults and failures, but we expect faulty workers to get repaired after some time.

1.2.3 Application Model

An application written for the model consists of local and remote function calls. Local function calls are invoked at a single node in which they are triggered. The remote function calls invoked by the controllers on sub–controllers and workers are called downcalls. For instance, the root controller can downcall to the fog–level or device–level controller. Downcalls from a node is usually to all the nodes under it (one-to-many call). Calls by the workers to controllers at any level (i.e., one of the cloud–level, fog–level, or device–level controllers) are called upcalls. Upcalls are one–to–one calls (unicast).

The upcalls and downcalls can either be synchronous or asynchronous as shown in Fig. 1.2. A synchronous call blocks the calling node until the remote execution

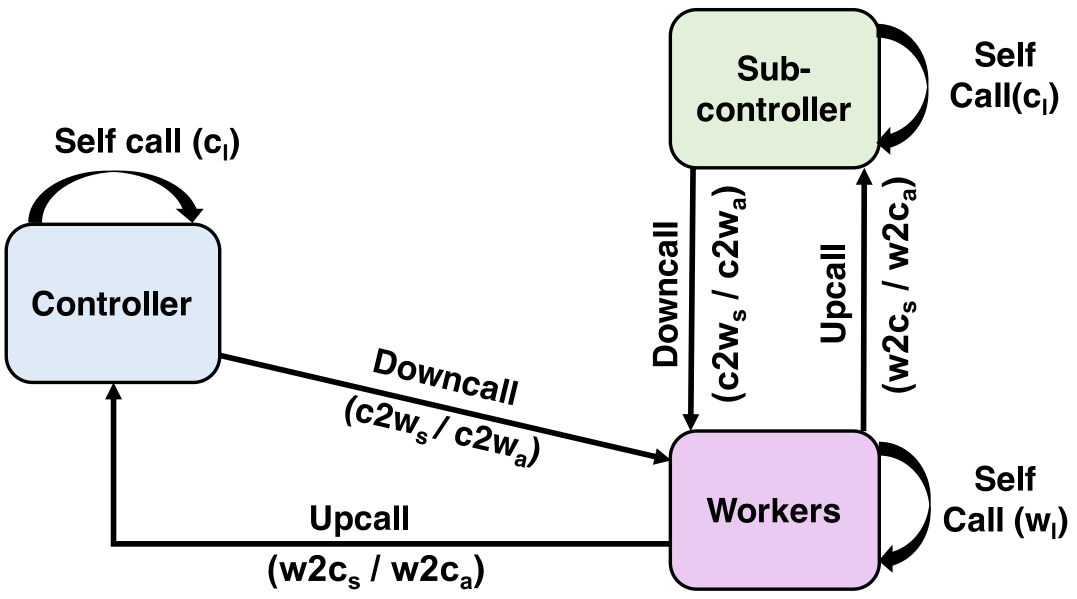

Self call (c_l)
Controller
Sub-controller
Self Call(c_l)
Downcall ($c2w_s$ / $c2w_a$)
Upcall ($w2c_s$ / $w2c_a$)
Downcall ($c2w_s$ / $c2w_a$)
Upcall ($w2c_s$ / $w2c_a$)
Workers
Self Call (w_l)

Fig. 1.2 Task model showing synchronous (with subscript s), asynchronous (with subscript a) and local (with subscript l) calls between controllers and workers.

completes, while an asynchronous call is nonblocking. Because the upward and downward calls can involve multiple workers, the synchronous execution of the calls is not simple. In particular, we need to define the conditions under which the group would provide a valid execution for a synchronous call.

When a controller makes a synchronous downcall, workers individually sign up for executing the function. Once the controller receives commitments from enough workers to run the function, the controller will proceed with the running of the function on workers. The requirement is to start the execution precisely (assuming the underlying clocks are synchronized) at the same time across all workers. This is a hard problem because even the signed–up workers can become available to execute the function at different times. Therefore, the function execution needs to be scheduled across the workers such that the start–time skew and idle times at the workers are simultaneously minimized.

We classify the tasks in our model based on function calls into the following categories.

1. **Controller–to–worker asynchronous call**: This is a call from a controller to its worker nodes to run a task without having strict timing conditions, that is, the controller does not wait during this period. We denote this task as $c2w_a$.

2. **Controller–to–worker synchronous call**: The controller sends a command to all its worker nodes to start executing a task at the same point in time. The controller waits until all worker nodes have finished executing the task and returned an output. We denote this task as $c2w_s$.

3. **Worker–to–controller asynchronous call**: This is a call from a worker node to one of its controllers to run a task. The worker does not wait during this period and continues with its own execution plan. This task is denoted as $w2c_a$

4. **Worker–to–controller synchronous call**: The worker node waits for an acknowledgement from the controller that it has finished executing the task associated with the call. We denote the task as $w2c_s$.

5. **Self calls**: This is a call from a node to itself. It could be either a worker node calling itself or a controller calling itself. A self triggered worker task is called a local worker task and is denoted as w_l, while a self triggered controller task is denoted as c_l.

1.3 Thesis Contributions

The four major contributions of this thesis are as follows:

1. We study synchronization in complex edge distributed systems where tasks can have different timing requirements. We develop a taxonomy that describes different synchronization strategies.

2. We develop new synchronization schemes that can handle various task types (synchronous, asynchronous and local) and task graphs

configurations (sporadic, periodic, dense and light). We use the well known publish-subscribe messaging protocol to reduce communication overhead in reaching synchronization. This thesis is the first study to optimize synchronization tasks with local tasks.

3. We further incorporate fault tolerance into the synchronization schemes using time and component redundancies and showed how it works. The schemes developed provide capabilities such as handling not only synchronous tasks but also local tasks among workers, as well as ahead-of-time computations to minimize synchronization overhead, thus making them lightweight.

4. We develop a new and fast synchronization scheme for distributed training in an edge environment. We achieve faster synchronization compared to existing schemes by pushing computational intensive actions to static time and using game theory to make optimal synchronization decisions, thus making runtime activities as fast as possible. We minimize the controller's involvement and message overhead (which could be a major bottleneck) using clustering and silent notifications.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we provide some background on edge computing, fog computing, clock synchronization approaches, synchronization in distributed systems, nature–inspired synchronization and synchronization in real–time AI applications. We provide an overview of existing literature and works on synchronization in distributed systems and IoT systems, as well as synchronization in distributed training in AI applications. In Chapter 3, we develop a taxonomy for synchronization in IoT. Motivating use cases and application scenarios as well as challenges in synchronization in IoT systems and AI applications are likewise given in Chapter 3. Chapter 4 introduces three novel task–based synchronization algorithms for fog–controlled IoT. The evaluation of the proposed algorithms through extensive simulations and detailed analysis of the results are likewise given. In Chapter 5 we introduce two new redundancy-based synchronization algorithms for fog–controlled IoT. We evaluate the performance of our algorithms through extensive simulations and comparing with existing synchronization schemes. Chapter 6 presents a game–theoretic fast synchronization scheme for distributed training in edge–based AI application tasks. The scheme is evaluated using simulations and implementation and compared to existing schemes. A summary of the thesis and possible future extensions of this research is given in Chapter 7.

Chapter 2

Background and Related Works

2.1 Edge and Fog Computing

2.1.1 Edge Computing

Edge computing [23, 24, 25, 26] is a computing paradigm that enhances the computational, storage and management capabilities of IoT systems by processing data generated by IoT devices at the edge of the network. Edge computing looks to minimize the shortcomings of cloud computing by reducing data transfer over the network and processing data closer to where the data was generated. The edge is defined as any computing or network resources that is along the path between data sources and the fog layer, for example, smartphones, gateways, etc, as shown in Fig. 2.1. With more demands being placed at the edge of the network, there is a need to provide some performance guarantees that is usually provided by the cloud such as security, privacy and reliability.

The two common implementations of edge computing are the hierarchical [57, 58] and the software defined models [59]. The hierarchical model is suitable for edge computing because edge servers can be deployed at different distances from the end devices. The functions of the edge servers are thus

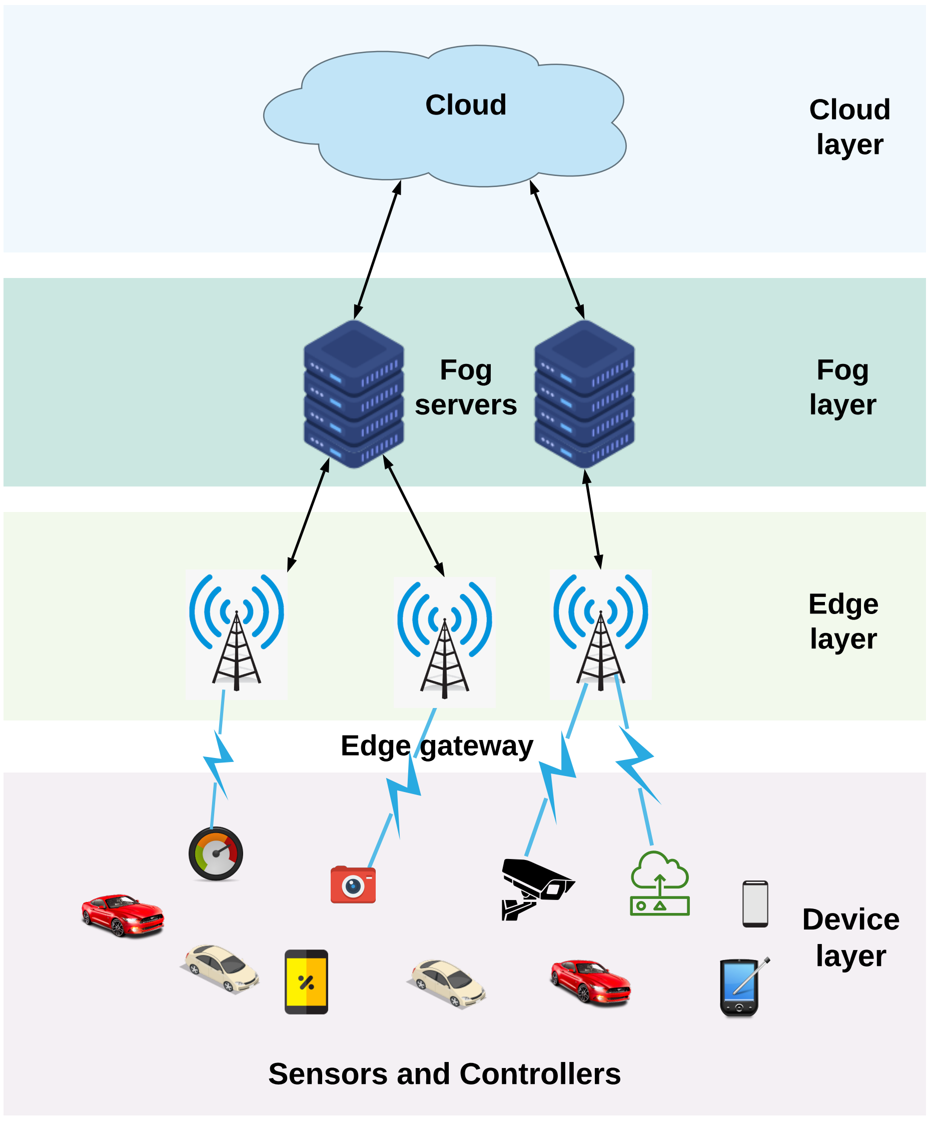


Fig. 2.1 Architecture showing edge and fog computing.

defined based on their distances from the end devices. The software defined models leverage Software Defined Networking (SDN) to manage the complexities of large–scale edge computing systems. Edge computing has been integrated with other domains such as mobile computing, vehicular computing and artificial intelligence, leading to mobile edge computing (MEC) [60, 61, 62], vehicular edge computing (VEC) [63, 64, 65] and AI on the edge (Edge AI) [36, 66, 67, 68] respectively.

Mobile edge computing is an implementation of edge computing where the radio access network (RAN) [69, 70] is the core of the edge [71]. Edge servers in MEC are usually co–located with RAN controllers in a (micro) base station. MEC servers not only offer computational resources to end devices, but also offer real time information on the network status. Vehicular edge computing is a way of integrating vehicular networks into MEC. The aim of VEC is to move computing, communication, and storage resources closer to vehicular units. Vehicular units by themselves are equipped with these resources, however, roadside units (RSUs) which serve as edge servers are placed close to vehicles for acquiring, processing and storing data from various vehicles [64].

Edge AI refers to systems that use machine learning algorithms to process data by devices generated closer to the edge of the network, thus, closer to the data source. Real time decisions can be made since there is no data offload involved. Edge AI pushes processing and data closest to the point of interaction of users. Using AI, edge computing applications can provide applications that have very minimal latency, improved user experience, additional intelligence and eliminates privacy issues.

2.1.2 Fog Computing

Fog Computing [20, 21, 22] is a distributed paradigm that advocates moving computing closer to end devices by operating at the local area network. Fog computing is an extension of cloud computing from the core network to the edge network as shown in Fig. 2.1. Fog computing has to work in conjunction with the cloud and 5G networks [27, 28, 29] to provide the computing power to the applications. Some characteristics of fog computing are mobility, wireless access, low latency, location awareness, large number of end devices, real–time interaction, inter–operability and heterogeneity. These characteristics make fog computing ideal for IoT applications and services. Some example cases where fogs have been deployed as a platform for IoT are connected vehicles [72], smart grids [73] and smart cities [74]. Smart grids and smart cities are also good examples of where the richness of the fog and its relationship with the cloud is used to deploy IoT applications. Fog computing improves reliability by making applications immune to localized link disruptions. Fog computing likewise improves privacy and security because of the local aspect it provides.

Traditional fog computing architectures follow a three–tiered structure, with the fog in between the cloud and devices. With fog computing, computation, storage, networking, decision–making, and data management need not only occur in the cloud, but also occur along the IoT–to–Cloud path as data traverse to the cloud (preferably close to the IoT devices). The horizontal platform in fog computing allows computing functions to be distributed between different platforms. In addition to facilitating a horizontal architecture, fog computing provides a flexible platform to meet the data-driven needs of operators and users. One of the key challenges in fog computing is programming. With fog computing, we need to deal with clouds and edge devices as well. That is, the programmer needs to identify the processing tasks for the global scope of the cloud computing, local scope of fog computing, and device scope of edge computing. Other potential issues that need to be resolved before having a fully efficient fog computing system include fog networking, quality of service (connectivity, reliability, capacity, and delay), computation offloading, programming model, resource management, privacy and security [75].

2.2 Clock Synchronization

In IoT, clock synchronization [76, 77, 78, 79] has been a major focal point where the main goal is to make IoT devices have the same notion of time, either real or logical. The following quality metrics are the most important in clock synchronization [80]: (i) clock skew – difference between system clocks and external reference clock, (ii) clock offset – difference between any two system clocks, and (iii) integration time – the time to synchronize a non–synchronized system clock. Achieving clock synchronization in IoT introduces several challenges because of varying environmental conditions, the desired low cost of

IoT devices/sensors, unstable network connections, limited bandwidth and constrained capabilities (computation, communication and processing) of IoT devices [81, 82, 83].

Several protocols have been proposed for clock synchronization in IoT and wireless sensor networks [83]. Centralized (master–slave) solutions [84, 85] consist of slave nodes synchronizing their clocks to using the master's clock as a reference while distributed (peer–to–peer) solutions [86, 87] have nodes that communicate with one another to synchronize their clocks. Internal synchronization clock protocols aim to minimize the offset among clocks in the system while external synchronization involves synchronizing the clocks of nodes in the system to an external real–world clock using a standard source such as Universal Time (UTC) [88]. Probabilistic synchronization protocols provide only some probabilistic guarantee on the tolerable maximum clock offset among the clocks in the system, while deterministic synchronization protocols provide guarantee on the maximum clock offset with certainty. A blockchain–based clock synchronization was proposed in [89] for IoT where a consensus algorithm is used to guarantee the real–time and security needs of IoT systems.

A clock synchronization algorithm was proposed using the time–triggered architecture for TTEthernet, an extension of the standard Ethernet [81]. The TTEhernet network has switches and end systems that are connected with redundant channels to achieve fault tolerance. The three components of the synchronization algorithm are synchronization masters, compression masters and synchronization clients. The synchronization masters are responsible for sending initial clock information state to compression masters, the compression masters perform a convergence function on the information and send the new data back to the synchronization masters, the synchronization clients get the clock synchronization information from the synchronization masters and ensure that only one data frame from a single synchronization master is used in each cycle.

A flooding time synchronization protocol was proposed in [85] to achieve micro–second time synchronization range with scalability up to hundreds of nodes. The algorithm synchronizes the local clocks of participating nodes using a single radio message that is timestamped at both the sender and the receiver. A sender can synchronize with multiple receivers in this protocol. In [90], an improved flooding time synchronization protocol was proposed, where nodes do not need to send the *id* and *sequence number* of the synchronization message. This is because the time synchronization message is the same as the one sent previously.

2.3 Synchronization in Parallel and Distributed Systems

In gang scheduling [91, 92, 93, 94] and coscheduling [95, 96, 97] where tasks of a parallel job are expected to be scheduled and executed at the same time, synchronization is achieved by using busy waiting. In such systems, if tasks that are dependent on each other are not scheduled and executed at the same time, it can lead to starvation or deadlock. Thus, to achieve task synchronization, tasks with dependents must wait for processors to be available until all the dependent tasks can be scheduled at the same. Tasks have preemptive executions and thus, a task can be suspended after it has started executing and continue execution at a later point in time.

Coscheduling involves scheduling related processes or tasks to run in parallel. There are frameworks consisting of dependent tasks or tasks that need to communicate with one other for successful completion, and thus, must be scheduled together [97]. There are distributed applications that require that sub–tasks should be coordinated and synchronized, thereby bringing about the need for a scheduling mechanism to guarantee this synchronization. Coscheduling solves this problem by ensuring that communicating sub–tasks are available for interaction when needed. For example, consider an application where a running task needs to send a message to a task that has not been scheduled, the task will keep waiting for a reply that is not forthcoming, which will cause blocking in the execution of the application. There are two variations of coscheduling namely implicit coscheduling and explicit coscheduling [95]. In explicit coscheduling, all dependent tasks are scheduled at the same time, that is, it is either all the tasks are scheduled together or none of the tasks is scheduled.

A global scheduling mechanism is employed in explicit coscheduling. In implicit coscheduling, there is no strict enforcement of the rule that all tasks must be scheduled together, rather, tasks can be scheduled independently using local scheduling, but scheduling decisions are made in cooperation.

Gang scheduling is a stricter form of coscheduling where dependent tasks are scheduled simultaneously. The tasks are usually from the same job or framework. Gang scheduling ensures that tasks can communicate with one another at any point in time and are thus scheduled concurrently. The main challenge of gang scheduling is how to achieve high cluster utilization, this is because all the tasks in a gang must wait until enough machines are available to run all the tasks at once. In [91] they identified the optimal performance conditions and efficient mean response time of jobs while ensuring fairness to different categories of gangs (small and large gangs). Resource sharing among running tasks also incurs some overhead in gang scheduling [98]. To solve the inherent problem of the neglect of memory considerations in gang scheduling, a gang scheduling algorithm with memory considerations was proposed in [92].

The Paxos [99] algorithm was proposed for achieving fault tolerance in distributed systems. It uses a consensus algorithm to achieve a single and unified update. Paxos employs a coordination model to achieve the goal of having a single value accepted by all servers. The algorithm chooses a leader that serves as the coordinator of all activities. It is assumed that there will be a leader at every point in time. A more defined specification of the Paxos algorithm was proposed in [100]. It was intended for helping system builders to realistically implement the Paxos algorithm in their systems. The Paxos protocol [101, 102, 103] ensures synchronization during a leader election by doubling the progress timer to create more time for the completion of the leader election and to allow newly elected leaders to have more time to complete global ordering. Synchronization is likewise achieved by ensuring that servers only move to a higher view (having a more up–to–date log) on suspecting a failed leader and that the progress timer of a server is not already set.

In parallel computing models such as the bulk synchronous parallel (BSP) model [104] where the execution model is broken into computation and communication super steps, synchronization is achieved by using barriers. Barrier synchronization involves processes stopping at the barrier until all other processes reach the barrier [105, 106]. Thus, faster processes must wait for slower processes, with the bottleneck being the slowest process. The BSP model [104] consists of a sequence of supersteps. Computation and communication are separated into different supersteps to decrease the burden on the programming of synchronized parallel algorithms. The computation superstep consists of a few small operations, while a communication superstep consists of transferring a data word from one processor to the other. The communication superstep is defined in terms of *h–relations*, where h is the maximum number of messages (words) sent or received in a superstep. The communication time is determined by h and g, where g is the time required to send a single word of data (usually measured in terms of number of CPU operations). The synchronization cost L contains fixed overhead costs such as data sending start–up costs and costs of global checking to determine whether a superstep has been completed by all components. Therefore, the cost of the communication superstep is given by:

$$T(h) = hg + L \tag{2.1}$$

The cost of the computation superstep is defined as:

$$T(c) = c + L \tag{2.2}$$

where c is the maximum number of operations of a processor in a superstep. The BSP model was further extended in [107] by introducing *memory/cache size* as a new parameter and also using a hierarchical structure with an arbitrary number of levels.

Time slotting has been adopted as a way of achieving synchronization in wireless sensor networks [108, 109]. Dedicated synchronization time slots are chosen, and devices attempt to perform synchronization tasks only in the dedicated time slots. Time slotting suffers from straggling workers, as with barrier synchronization. Slow workers will miss the dedicated synchronization slot, which consequently results in the reduced synchronization quality or synchronization failing.

A preemptive scheduling algorithm for synchronized task groups was proposed in [110] where tasks are partitioned based on communication, synchronization and mutex. Synchronization tasks having the same priority are expected to arrive at the synchronization point (SP) at the same time and cannot be pre–empted. The synchronization task with the maximum execution time before the SP is scheduled to run first, while that with the maximum execution time after the SP is scheduled to run immediately after the SP. In [111], a preemptive synchronized scheduling policy was proposed for synchronization tasks that share the same mutual exclusion resources under homogeneous processors. Tasks are allowed to share resources based on a mutual exclusion policy. Tasks that share mutually exclusive logical resources are grouped into the same component and have a local scheduler. Whenever a global scheduler decides on a component to be executed, the local scheduler decides which task gets the resource access.

The synchronization–aware algorithm proposed in [112] bundles tasks by size or mutex sharing and attempts to schedule them onto a processor. Bundles that do not directly fit into a processor are put in a separate queue and sorted based on their cost (the penalty of transforming a local mutex into a global mutex) in increasing order. The bundle with the smallest cost is broken down into pieces, with the largest piece determined by the size of the largest possible space in the processors. If the process is not successful, a new processor is added, and the partitioning process is repeated.

In real–time computing systems, synchronization is handled using time–triggered controls where all synchronous activities are executed at some predefined points in time [113, 114]. Synchronized clocks are used to achieve synchronization in the systems by making each node have similar internal clocks. Time synchronization schemes [17, 18, 115] have been developed for IoT to allow devices have a common notion of time. In [17], a visible light produced by light–emitting diodes (LEDs) is used by devices within that vicinity to synchronize. Synchronization is achieved by allowing several LEDs to send out binary signals at the same time and at predefined intervals, and devices synchronize when there is a phase transition. In [18], a time synchronization protocol was proposed to mitigate the effect of temperature change on hardware clocks in IoT networks using time–slotted channel hopping.

Synchronization schemes in distributed systems cannot be directly applied to IoT due to the following factors. (i) Node connectivity could be highly unstable in IoT due to mobility and disconnection, unlike distributed systems where stable connection exists among nodes. Synchronization schemes in distributed systems are developed based on this assumption and are thus not suitable for highly dynamic systems, (ii) the network topology rapidly changes due to nodes joining and leaving in IoT, unlike distributed systems, (iii) there is an interaction with physical things that have real–time window constraints in IoT unlike traditional distributed systems which do not necessarily affect real world systems, and (iv) nodes in IoT could be highly heterogeneous e.g., sensors, mobile phones, cars, etc., unlike distributed systems where nodes have similar characteristics.

2.4 Nature–Inspired Synchronization

A common and naturally occurring synchronization is one noticed in fireflies [116, 117, 118, 119, 120]. Male fireflies randomly emit flashes at night and over a period, the flashes get synchronized. Analysis of firefly synchronization has been carried out and lots of models have been developed over the years. Pulse coupled oscillators (PCOs) [121, 122] have been used to study the synchronization behavior in fireflies. PCOs refer to systems with interacting (through pulses received from neighbors) oscillators that oscillate periodically in time. Synchronization of PCO involves making the individual oscillators emit pulses at the same time. Synchronization in fireflies is eventual, meaning that the system goes through a process of communication among components and phase adjustments before finally reaching synchronization. Synchronization in PCOs is achieved by adjusting the phase of oscillators (using a phase modification function) upon receiving a flash message from another oscillator.

Firefly inspired synchronization schemes have been proposed for wireless and sensor networks in IoT [121, 122, 123, 124, 125]. The IoT sensors act as phase coupled oscillators (PCO) that attempt to synchronize their phases based on some phase adjustment function. The resultant synchronization is pairwise between the oscillators that sent and received the flash messages. Well studied models for achieving synchronization in PCOs include the Kuramoto model [118], Mirollo–Strogatz [117] and Ermentrout model [119].

The Kuramoto model [118] consists of a population of n nearly identical and coupled phase oscillators having frequencies ω_i distributed with a given probability density. The model assumes global coupling such that each oscillator is affected by every other oscillator. The oscillators naturally run at their natural frequency. The coupling among the oscillators tends to synchronize the oscillators with one another. The phase dynamics of the oscillators is governed by the function:

$$\phi_i = \omega_i + \sum_{j=1}^{n} H_{ij} \sin(\theta_j - \theta_i), \quad i = 1, ..., n$$

The oscillators run incoherently when the coupling H_{ij} is weak, however, beyond a threshold, synchronization emerges. Kuramoto was able to prove that there will be a phase transition to synchronization (convergence) and show that a direct equation exists that gives the necessary coupling strength needed for synchronization.

The Mirollo–Strogatz model [117] uses a simple communication and node model. Nodes observe flashes from their neighbors, but do not care which neighbor they received a flash from. The only state kept by a node is its internal clock. The M&S model introduced non–linearity into the simple phase–advance model by introducing a voltage variable. This allows for easier adjustment of the sensitivity of the phase, depending on the actual phase upon receiving a flash message. The rate of adjustment is determined by a firing function $f(t)$ and the pulse strength ϵ, which is a small constant < 1. Suppose a node receives a flash message from a neighbor at time $t^{'}$, it instantaneously jumps to a new interval time $t^{''}$, where

$$t^{''} = f^{-1}(f(t^{'}) + \epsilon)$$

If $t^{''} > \omega$, the node immediately fires and resets its internal time to 0. Mirollo and Strogatz proved that if the function f is concave down, monotonically increasing, and smooth, the set of nodes (PCOs) will always converge regardless of the number of nodes or their starting points.

The Ermentrout model [119] allows PCOs to synchronize at a nearly zero phase difference. This model updates the frequency of an oscillator upon receiving a flash message from a peer, rather than updating the phase as in other models. The model has a *phase response curve* PRC that pushes the frequency high or low depending on the point the flash message was received. Ermentrout proved that the system converges to synchronization with very small difference in the phases of the oscillators.

$$\omega^{+} = \omega + \epsilon(\omega_n - \omega) + \begin{cases} f^{+}(\phi)(\Omega_l - \omega) & \text{if } \phi < \frac{1}{2} \\ f^{-}(\phi)(\Omega_u - \omega) & \text{if } \phi > \frac{1}{2} \end{cases} \tag{2.3}$$

with

$$f^{+}(\phi) = \max(\frac{\sin(2\pi\phi)}{2\pi}, 0) \quad f^{-}(\phi) = -\min(\frac{\sin(2\pi\phi)}{2\pi}, 0) \tag{2.4}$$

where ω_n is the natural frequency of the oscillator, Ω_l and Ω_u are the lower and upper bounds on the frequencies of oscillators. The functions $f^{+}(\phi)$ and $f^{-}(\phi)$ are non–negative periodic functions in the interval $(0 < \phi < 1)$ that slows down or increases the frequency of a node. Function $f^{+}(\phi)$ is positive when $\phi < \frac{1}{2}$ while $f^{-}(\phi)$ is positive when $\phi > \frac{1}{2}$.

A synchronization scheme combining heartbeat synchronization and a firefly inspired model for overlay networks was proposed in [123]. The proposed protocol has two main parts, nodes selecting their peer list and processing a flash message received to achieve synchronization. A game theoretic approach for synchronizing pulse coupled oscillators was proposed in [126]. The model developed is an extension of the well–known Kuramoto model for synchronizing systems of oscillators. The game is noncooperative, with the oscillators in the system competing against one another. An "oblivious solution" was developed where individual oscillators do not have access to the full system state. The oscillators make decisions strictly based on local states and a consistent average value.

An emergent broadcast slot synchronization scheme inspired by firefly algorithms was proposed in [122] for Internet of Things (IoT). Each node maintains a time window that it can be awake during a steady state. Nodes in the network go through three states to achieve synchronization. The first state is the initialization state, where nodes start their random timers and identify their neighbors. The nodes then transition to the synchronization state, here, the nodes coordinate their synchronization error tolerance window with their neighbors using a Pulse Coupled Oscillator (PCO) model with a phase advancement function. A node becomes synchronized if its synchronicity is greater than the synchronization threshold. Thereafter, the node moves into the steady duty cycle state. In this phase, nodes only wake up during their synchronization error tolerance window to exchange messages.

A PCO based synchronization scheme was proposed in [127] for the synchronization of wireless sensor networks with a high degree of scalability. Nodes have an internal clock that increase with a constant speed from 0 to a specified period. Whenever a node crosses the specified period, its linearly increasing phase is 1, it emits a message and resets it phase to 0. Nodes that receive a message from another node update their internal clock and phase according to an updating function. A refractory period is included immediately after firing to introduce stability into the scheme. A synchronization scheme for oscillator networks with stochastic behavior was developed in [128]. When an oscillator's phase reaches or passes 1, it resets its phase to 0 and then probabilistically fires. Oscillators receive a pulse from other nodes after a stochastic delay and update their phase according to a phase function. Synchronization is reached when oscillators align their phases to form an invariant subset of the state space and all oscillators within the subspace have their phases synchronized.

2.5 Synchronization in Real–Time Artificial Intelligence Applications

Synchronization is an important problem in iterative algorithms such as distributed real–time ML, where the convergence rate and the per iteration times are affected by it. The BSP model is one of the earliest ideas. While BSP [104, 107, 129] guarantees total participation in synchronization, it performs poorly in heterogeneous systems where devices can perform the same computations in different amounts of time. This causes BSP to suffer a lot from straggling devices. ASP [130, 131] is the other extreme, where workers do not have to wait for one another.

The asynchronous parallel model (ASP) [130, 131] is a distributed training method where training nodes propagate their updates to the parameter server as soon as they are available without considering the other nodes. ASP is the slackest form of synchronization, where worker nodes do not have to wait for one another. Asynchronicity mitigates the effects of straggling workers as well as reduces the extra communication overhead in synchronizing. However, due to the staleness in updates and gradients, ASP takes much more iterations to reach convergence.

In [130], a parallel and decentralized ASP model was proposed for stochastic gradient descent (SGD) [132, 133, 134, 135]. They sought to develop a communication–efficient asynchronous algorithm for distributed training in a heterogeneous environment while guaranteeing convergence. Each worker in the system maintains a local model in its memory and undergoes the following steps (i) sample a mini batch of data, (ii) compute the stochastic gradient, (iii) update local model, and (iv) select random neighbor, average local model, and set the local model to averaged model. The workers run the steps in isolation without any global synchronization. They proved theoretically and experimentally that the algorithm converges and has performance as good as its synchronous counterpart. The asynchronous parallel stochastic gradient descent algorithm proposed in [131] uses the one–sided asynchronous communication paradigm to achieve fast convergence with linear scalability and some guaranteed accuracy. The one–sided asynchronous communication paradigm uses early communication of data to workers.

The BSP model was originally developed for parallel systems such as Hadoop [136], MapReduce [137], Pregel [138] and Spark [139]. In BSP distributed training models, the next training iteration only starts after all workers have completed the previous iteration as shown in Fig. 2.2. BSP is a more perfect match for a homogeneous distributed worker setup where workers are expected to have relatively the same run time per iteration.

An improved BSP algorithm was proposed in [140]. The elastic BSP model relaxes the strict synchronization in BSP to allow for faster and better convergence. It differs from SSP by considering the processing capabilities of workers and uses this information to schedule future synchronization points. Fixing the synchronization barrier varies during training, depending on the runtime environment. The model assumes a stable environment and predicts the future iteration times of workers based on the most recent iterations. A greedy look–ahead algorithm is then used to determine the next synchronization point based on the predicted future iteration times of workers. The algorithm looks to minimize the overall waiting times among the workers.

SSP [141, 142] takes a nuanced approach to solve the straggler problem by relaxing BSP's strict barrier condition as shown in Fig. 2.2. SSP allows devices to proceed to the next iteration if the gap between the fastest and slowest device is within a bound – called staleness value. Although, SSP leads to faster

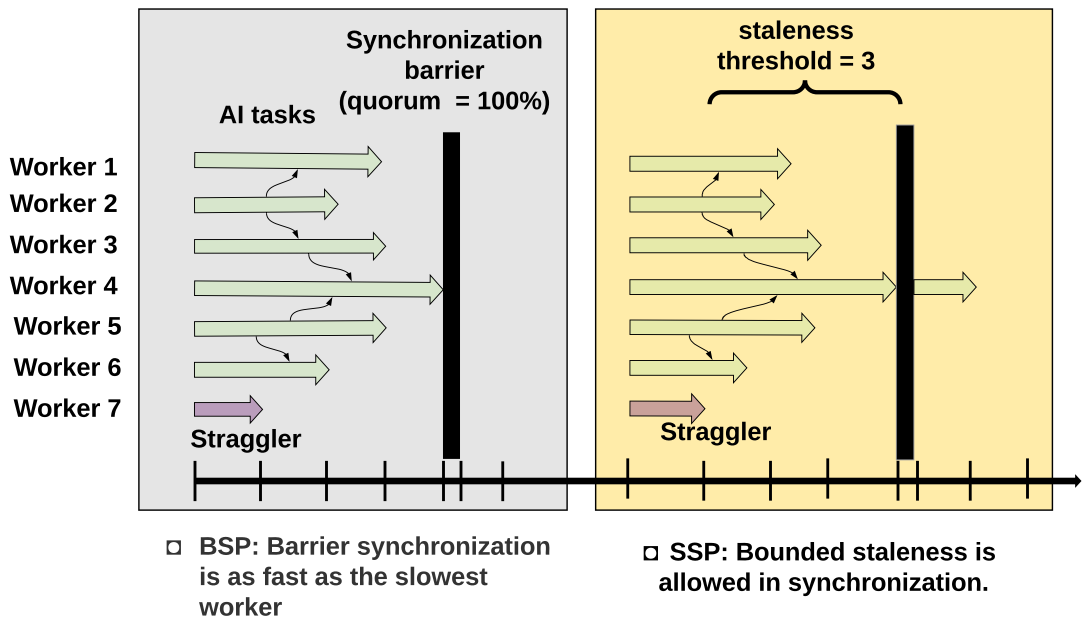

Synchronization barrier (quorum = 100%)
AI tasks
Worker 1
Worker 2
Worker 3
Worker 4
Worker 5
Worker 6
Worker 7
Straggler
staleness threshold = 3
Straggler
BSP: Barrier synchronization is as fast as the slowest worker
SSP: Bounded staleness is allowed in synchronization.

Fig. 2.2 Example showing the workings of bulk synchronous parallel (BSP) and stale synchronous parallel (SSP) models.

execution and less wait time, the quality of synchronization is reduced compared to BSP by allowing asynchrony. An SSP implementation enforces at least the following requirements – bounded clock difference, timestamped updates, model state guarantees, and read local writes.

A stale synchronous parallel (SSP) parameter server model for distributed machine learning was proposed in [141]. Each worker maintains an internal clock with which it updates shared parameters. Updates from each worker are committed at the end of their internal clock. A worker might not immediately see updates from other workers due to the different internal clocks operated by the workers. The algorithm maintains a user–defined bounded staleness, which is the largest tolerable gap between the fastest worker and the slowest worker. The staleness is measured in terms of clock (iterations). The fastest worker is forced to wait for the slowest worker to catch up whenever the gap between them is more than the staleness threshold. They showed in [141] that their algorithm outperforms both BSP and ASP algorithms. An eager SSP implementation was proposed in [143] where updates from workers are eagerly propagated to the parameter server. They showed that faster convergence can be achieved by reducing the average staleness.

DSSP [144] was proposed as an improvement on the SSP model for deep neural network training. In DSSP, rather than having a static staleness value, a value is dynamically selected from a range of values based on real–time processing speeds at runtime. Thus, the staleness value is continuously updated among workers as the training process continues, with the goal of minimizing waiting times or wasted work cycles. A synchronization controller is used to monitor the progress of worker nodes and for making projections about future execution progress. The goal of the dynamic staleness is to minimize the waiting (idle) time on workers. The synchronization controller always finds the slowest worker and after each update from a worker, it checks that the worker is no more than the staleness threshold (iterations) away from the slowest worker. A similar approach to DSSP was proposed in [145] where a performance monitoring model is used to adjust the synchronization delay threshold.

Chapter 3

Taxonomy and Motivation

In this chapter, we provide a taxonomy of synchronization in IoT by exploring a very wide range of operating conditions in edge IoT systems. We motivate the need for synchronization in IoT and distributed AI applications, as well as provide application scenarios and use cases.

3.1 Taxonomy of Synchronization in IoT

We classify synchronization in IoT as shown in Figure 3.1 based on the number of controllers in the system, number of controllers at each synchronization point, synchronization point time alignment, worker participation, worker node types, worker mobility, quality of synchronization and quorum requirement. The classifications are briefly explained as follows.

1. **Total number of controllers**: The number of controllers in an IoT system could be single or many, depending on the scope of the system. In the case of a single controller, all workers must be connected to the controller to be part of the system. In a multi–controller system, workers connect to the controller that is closest to them or assigned to their physical vicinity. Workers can

change controller membership as they move around.

2. **Number of controllers at synchronization point**: Synchronization could be at a global scale or local scale. In a global synchronization, all the controllers in the system must ensure that the devices under them synchronize their activities on the given sync task. In a local synchronization, only the subset of controllers that are affected by the synchronization are involved in the synchronization process.

3. **Synchronization point time alignment**: All the controllers participating in the synchronization can either orchestrate the devices under them to start the execution of the sync task at the same time or at different times. Thus, a single–controller system can only permit one aligned sync point, while a multi–controller system can permit both an aligned and a non–aligned sync point.

4. **Worker participation**: Some synchronization tasks require either all or some specified number of the workers connected to a controller to partake in the synchronization process. The participation of workers is dependent on their availability to run the sync task and the requirements of the sync task itself.

5. **Node types**: Synchronization in IoT is affected by node types. Nodes could be homogeneous, having similar attributes or heterogeneous, having different attributes.

6. **Node mobility**: Mobility is an important factor affecting synchronization in IoT. Nodes could be mobile, such as hand–held devices, cars, etc., or fixed, such as nodes placed on lamp posts.

7. **Quorum requirement**: The conditions at which synchronization can occur are either having at least the required fraction of nodes available to run the synchronous task (*ratio–based quorum*) or having the desired representation per group of clustered nodes (*cluster–based quorum*).

8. **Quality of Synchronization**: In atomic synchronization, a sync task is executed or fails depending on the quorum success or failure. In eventual

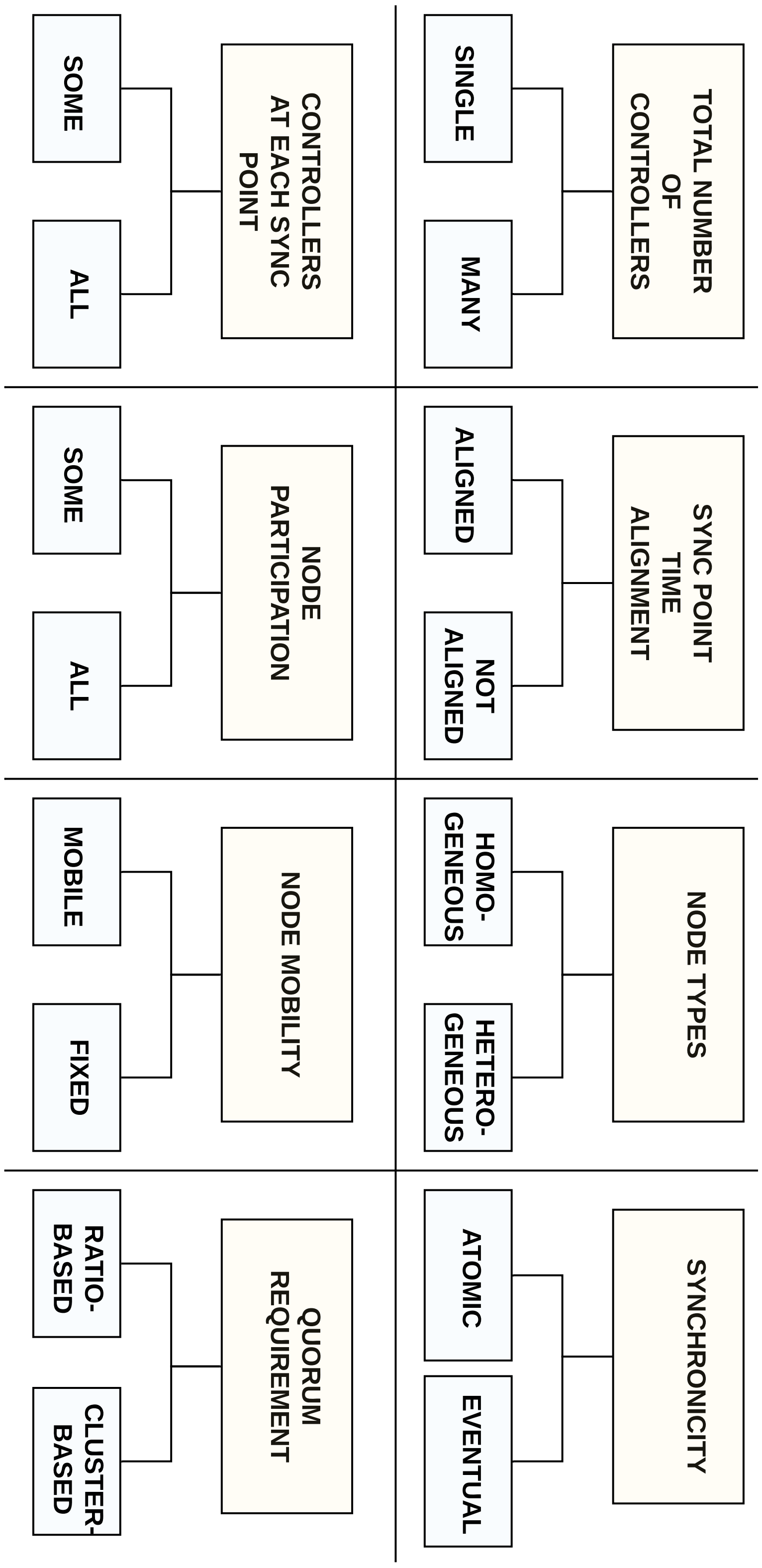


Fig. 3.1 Taxonomy of Synchronization in IoT.

synchronization, the sync task is run regardless of whether the nodes are initially synchronized or not, synchronicity is expected to increase with time.

3.2 Motivation

3.2.1 Synchronization in Fog/Edge–Based Internet of Things Systems

3.2.1.1 Use Cases

The use cases for synchronization in IoT are classified into the following three aspects:

1. **Capacity pool**: IoT and other smart devices are usually limited in their computing and sensing capabilities. Thus, there is a need for cooperation among several devices to solve a much bigger problem than can be solved by an individual device. The devices must pool their resources together in a coordinated manner to be successful. An example is a self–driving car moving on a smart higher. The car's dedicated resources must be coordinated with the smart highway resources for smooth and safe driving.
2. **Data capture synchrony**: Data synchronization [146] is an important problem in IoT data acquisition. Using synchronous tasks for capturing data allows us to precisely control the relative timing relations among the data points. In data capture synchrony, different subset of nodes can be made to synchronize at different points in time, thus achieving some level of data ordering.
3. **Resource usage synchrony**: IoT devices consume resources (e.g. power) for their operation; therefore, we need to use subset synchronization to sequence the operating order of the IoT devices to minimize the maximum resource usage profile. Take for example a smart lighting system consisting of numerous light bulbs, to light up (cover) a particular area, only a subset

of the bulbs needs to be turned on at the same time. Synchronization can be used to incrementally change the lighting intensity or maintain a constant lighting intensity.

3.2.1.2 Deployment Scenarios

Here, three application scenarios where synchronization among devices is of high importance are provided to motivate the need for synchronization and control schemes in IoT.

1. Edge Assisted Autonomous Driving and Synchronized Data Capture

In a typical edge assisted autonomous driving system [147, 148], vehicles report driving data to the edge server, while the edge server gather intelligence from the data and forward driving instructions to the vehicles as shown in Fig. 3.3.

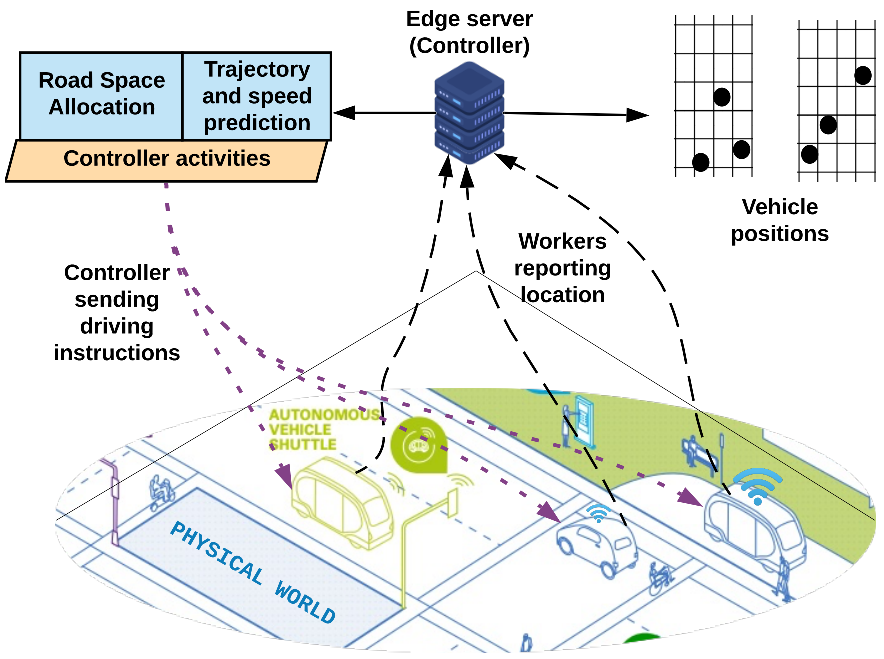

Edge server
(Controller)
Road Space Allocation
Trajectory and speed prediction
Controller activities
Vehicle positions
Workers reporting location
Controller sending driving instructions
AUTONOMOUS VEHICLE SHUTTLE
PHYSICAL WORLD

Fig. 3.2 Sample edge assisted autonomous driving scenario with the edge server as the controller and cars as workers.

If the workers report their locations asynchronously over a period of time, recreating vehicle positions at a particular point in time requires quite complex time shifting and transformation algorithms. We cannot take a global snapshot of vehicle positions in a meaningful way.

With synchronized data capture, it is easy to find how relative positions of vehicles are changing over time. More intelligent predictions and inference can be drawn from data with simple algorithms using synchronization. We can easily observe real world state by doing measurements at the same time.

2. Bridge Health Monitoring

This is an example of the *data capture synchrony* use case as shown on the right side of Fig. 3.3. Strain measurement at bridge joints and other important points in the structure is required to maintain a close watch on the health of a bridge structure [149]. To take high quality measurements, it is necessary to coordinate the data capture operations such that they are made when the loading is at a particular configuration. The loading configuration would be measured by the position of the vehicles (captured by drones) on the bridge at that instant and their weight.

The most accurate way of doing such a measurement is to actuate all involved devices (sensors, drones, and vehicles) to run the measurement function at the same time instant. If different devices run the measurement function at different time points, a complex reconstruction procedure need to be executed to determine the concurrent loading. The fine–grained measurements are only of interest while the vehicles are on the bridge. That is, they do not need to continue to take fine–grained position measurements and report them when they are not on the bridge.

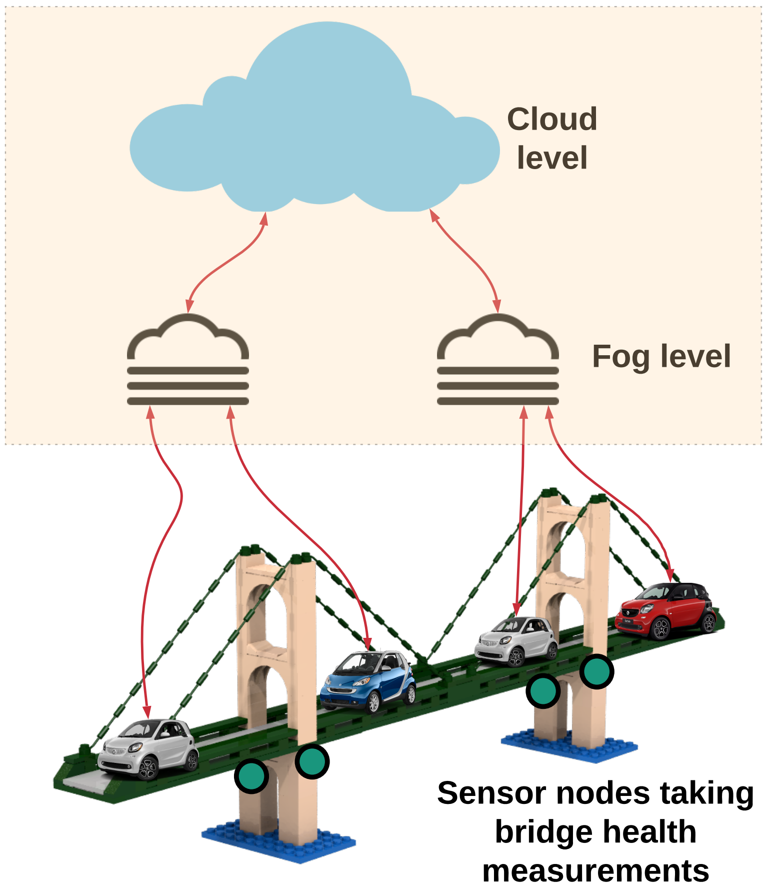


Fig. 3.3 Sample drone delivery and bridge monitoring system.

3. Smart Car Leveraging Smart Highway Resources

A smart (autonomous) car carries enormous amount of computing, storage and sensing capacities [150, 151, 152]. With fast wireless networks and edge computing, smart cars can share their capacities with other cars or with the roadside infrastructure and vice–versa. Such a swarm of cars will have significantly augmented capabilities; that is, a smart car could have more capabilities (i.e., higher level autonomy [153]) on a smart highway (SH) [154, 155] than what is capable of on a normal highway as shown in Fig. 3.4.

Smart cars need up–to–date information about the status of the road (presence of other cars and the road condition) and other driving conditions (such as weather)

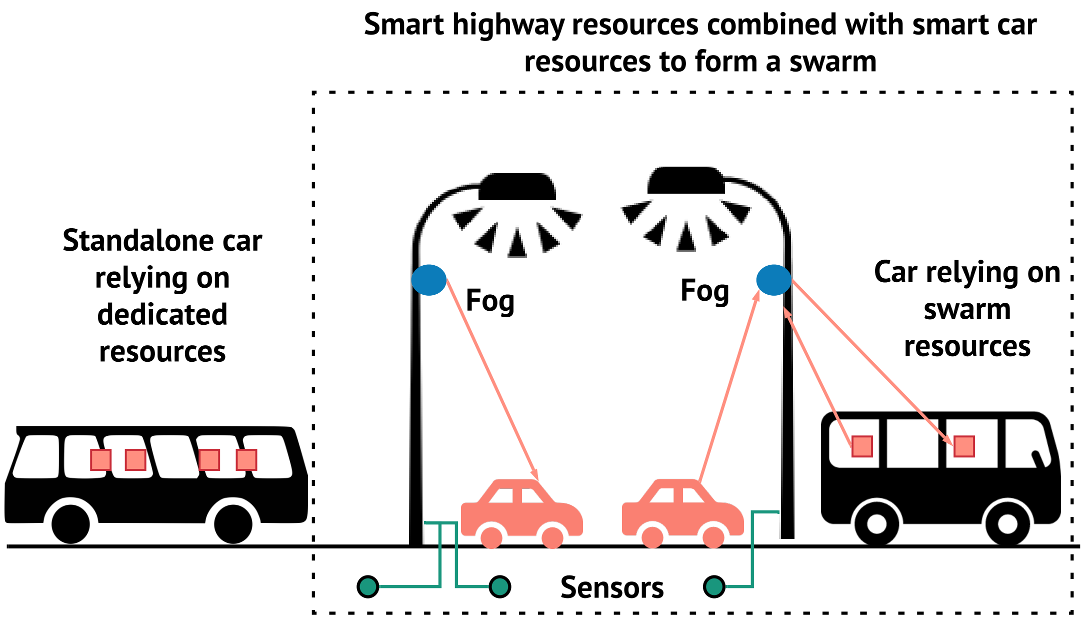

Smart highway resources combined with smart car resources to form a swarm
Standalone car relying on dedicated resources
Fog
Fog
Car relying on swarm resources
Sensors

Fig. 3.4 Smart cars relying on smart highway for improved driving experience.

to drive safely. The SH can be divided into segments, with each segment providing virtual resources to the swarm of smart cars within its range. The shared pool of resources including video cameras, pressure sensors, and speed monitors need to operate synchronously to tackle the tasks in real time without creating backlogs (i.e., a *capacity pool* use case for multipoint synchronization). The challenge here is to deal with coalitions that are short–lived (i.e., coalitions created and destroyed as cars move by) with low synchronization overhead.

3.2.2 Synchronization in Artificial Intelligence Application Tasks

3.2.2.1 Use Cases

1. Distributed machine learning: This is a multi–node machine learning system designed to increase accuracy, improve performance and scale in terms of data size by leveraging the processing and storage capacities of many nodes as shown in Fig. 3.5. The parameter server framework was developed as

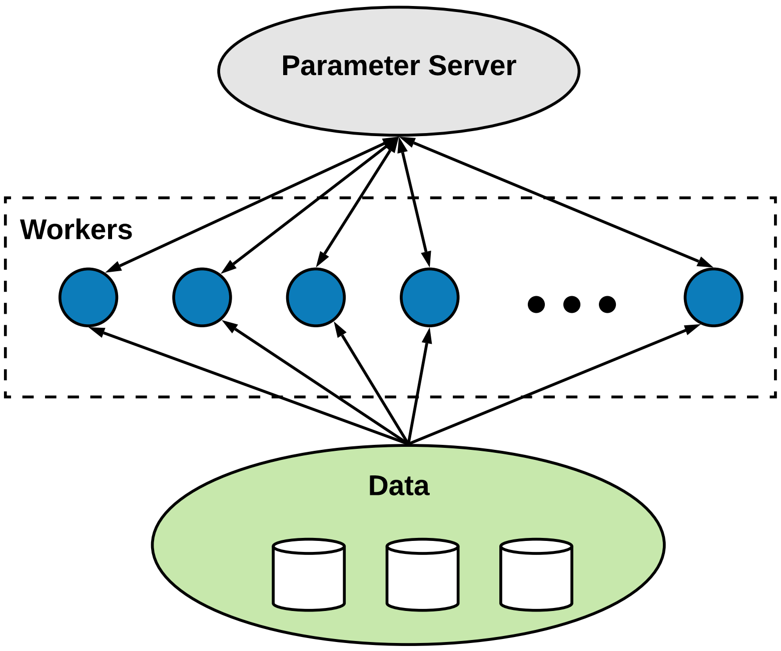


Fig. 3.5 Distributed machine learning architecture.

a means of aggregating gradient/model updates from distributed training nodes. Existing distributed machine learning frameworks [49, 156, 157, 158] use either the BSP, SSP, DSSP models or variations of the models. Most distributed ML algorithms suffer from a bottleneck of synchronization [159], especially edge–based distributed ML systems where the nodes can scale to very large numbers. The speed of convergence is greatly reliant on the synchronization approach adopted for aggregating partial updates from the distributed nodes.

2. Federated learning [160, 161, 162, 163]: This is a collaborative machine learning paradigm where computations are moved towards data as shown in Fig. 3.6. A globally shared model is pushed towards where the data are, for example, smartphones. A model can thus be collectively trained. Federated learning preserves privacy while building powerful intelligent systems. Synchronization is very useful in federated learning with the main challenges of optimization and communication. There is a need for aggregation of local model updates. The aggregation process must be

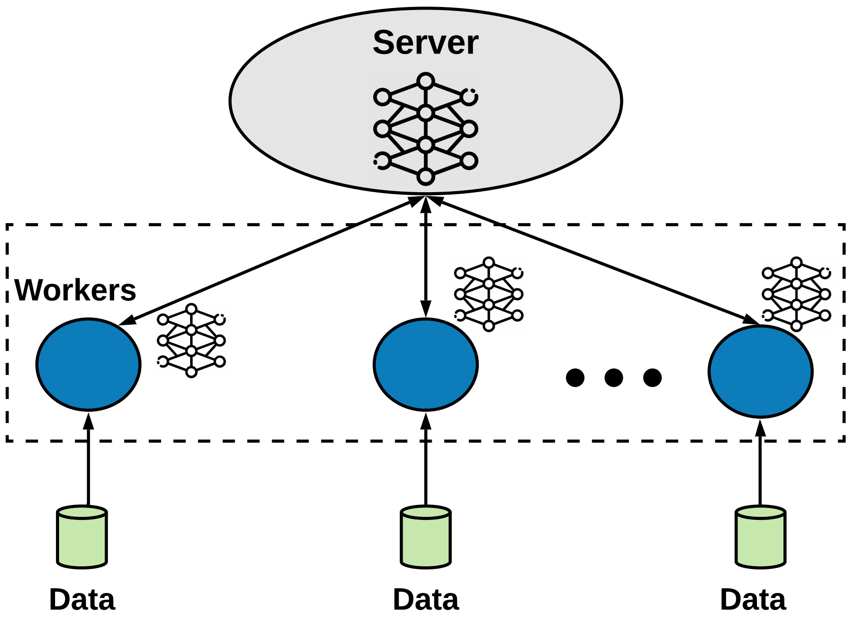


Fig. 3.6 Federated learning architecture.

synchronized in order to get good results from the learning process.

3.2.2.2 Deployment Scenario (Smart Health)

Medical data and reports are usually very sensitive and private. However, it is very difficult to collect extensive medical datasets in isolated hospitals and medical centers. The insufficiency of data and medical centers wanting to keep their patient's data private have made the performance of machine learning models to be unsatisfactory. Such a system will benefit from federated learning where a common global model is shared amongst the participating medical centers while keeping their data private. Only the trained local model is uploaded to the global model after some predefined training iterations. The challenge for training on such a system is in aggregating the local models from the individual participants and defining some synchronization update rules.

3.2.3 Summary

We look to develop synchronous scheduling schemes for fog–controlled IoT and AI application tasks where there is a need to check for the availability of nodes before sending a task for execution especially for synchronous tasks, where there are strict timing requirements. We expect that there will not be preemption of tasks unlike in co– and gang scheduling [91, 95] and as such a task cannot be stopped once it has commenced execution until it has finished the execution or failed. There is a heterogeneous task setup with a combination of local, asynchronous and synchronous tasks that need to be scheduled unlike the firefly–based (PCO) synchronization schemes [121, 122, 123] where synchronization is on a single flash message. The execution of a synchronous task cannot proceed unless the synchronicity conditions are met, whereas in firefly based (PCO) synchronization, eventual synchronization is sought for.

We look to extend beyond time synchronization [17, 18] by making sure that nodes need not only have the same notion of time, but that there is the right availability of nodes before executing a synchronous task. Node disconnection, leaving, and joining are catered for by introducing quorum checking to verify the availability and participation of nodes before executing a synchronous task. Fault tolerance is achieved by introducing time and component redundancy.

Chapter 4

Task–Based Synchronization Schemes for Fog–Controlled Internet of Things

4.1 Overview

In this chapter, we introduce our solutions for solving the problem of synchronization in a fog–controlled IoT system. Although clock synchronization [19] is necessary, it is not sufficient for simultaneous task executions among workers because workers could depart (due to disconnection or failure) or become unavailable to execute the given task due to a prior overshooting task. The synchronous tasks must be scheduled for execution only after receiving confirmation regarding the availability of the required number of nodes. We propose three synchronous scheduling algorithms based on the system model for scheduling synchronous tasks with strict synchronicity requirements on multiple nodes in the presence of other tasks. We evaluate the performance of the proposed algorithms through extensive simulations under different runtime conditions and analyze the results.

4.2 Synchronization Scheduling Schemes

Fog–controlled IoT is composed of cloud – providing global services, fogs – located closer to devices and providing low–latency services, and a massive number of IoT devices (workers) – comprised of fixed and mobile nodes. We assume that controllers can be resident in any of the three levels (cloud, fog, and device levels). We assume that all the nodes (controllers and workers) in the system are single threaded, i.e., they can only execute a single task at any point in time. The proposed algorithms use the basic system model given in Section 1.2.

At start–up, the same application is loaded onto all nodes in the system. Thus, we can keep track of the number of workers in the system. As workers join and leave, the number of workers in the system change. Since worker nodes cannot be standalone and need to be connected to a fog, the fog keeps track of the number of workers connected to it. The fog also assumes that all workers under it will run whatever task it triggers on them. We assume a heterogeneous system where worker nodes have varying but similar processing and computational capabilities, thus, worker nodes are expected to have different execution times for the same task. Due to network disruptions, mobility, or node failure, disconnections can occur between worker nodes and controllers. Disconnected workers can rejoin the system by connecting to a controller, and new workers can only join the system if they connect to a controller.

Depending on the amount of prior knowledge of the task graph (none, average or extensive) and the task arrival pattern (sporadic, periodic, or frequent), we develop three synchronization scheduling algorithms to handle a vast range of the cases. The notations used in the algorithms are described as follows. T_{curr} and T_{next} are the current task to be scheduled and the next task in the queue, respectively. t_{avail} represents the available time of a worker after finishing the execution of a previous task.

To meet with the desired QoS_{ync}, there is a need to check for the availability of workers before proceeding with executing a synchronous task $c2w_s$, we call this process *quorum checking*. The quorum check process involves workers updating the controller of their availability to run the synchronous task. We execute the quorum check task T_{quorum} to probe the controller if the required fraction of workers is available to run the synchronization task, subject to the synchronization degree α. T_{quorum} is scheduled to start at time τ which is a function of the synchronization degree, distribution of execution times and the finishing time of the task on all the workers before the synchronization point. The duration of T_{quorum} is dependent on how long it takes for the worker to ask the controller about whether there is a quorum and get a response. We perform quorum checking in two ways: naive and the sampling–based quorum checking. Table 4.1 gives a summary of the symbols used in the synchronization algorithms.

Table 4.1 Symbols for synchronization algorithms.

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

4.2.1 Naive quorum checking

The naive quorum checking requires all the workers in the system to participate in the quorum checking process. Whenever a synchronization point is reached, all available workers running the program send an *update* message to the controller to report their availabilities by running T_{update}. The workers thereafter run T_{quorum} to probe the controller to know if the desired quorum has been met.

4.2.2 Sampling–based quorum checking

Sampling–based quorum checking works by randomly selecting some workers to participate in quorum checking. We seek responses from the marked workers. Assume that there are N workers in total, with s selected ones. Ideally, at the synchronization point, we can get s responses. If we get k responses, we can estimate the number of available workers as $(k/s) * N$.

4.3 Static Synchronization Scheduling Algorithm (*SSSA*)

The SSSA shown in Algorithm 1 is run at compile time and assumes that we have prior knowledge of the task graph G. The task graph is topologically sorted with higher priority given to synchronous task and asynchronous task in that order. Ties between tasks of the same type are broken by giving priority to tasks with lower expected execution time. The topologically sorted set S guarantees that precedence constraints are maintained. SSSA produces a primary schedule and other alternative schedules that are used when the primary schedule fails quorum checking. The schedules are generated at compile time. The maximum number of quorum retries to attempt at each synchronization point is given as an input to the algorithm. The algorithmic flow is shown in Fig. 4.1. A description of the functions in Algorithm 1 is given in Table 4.2.

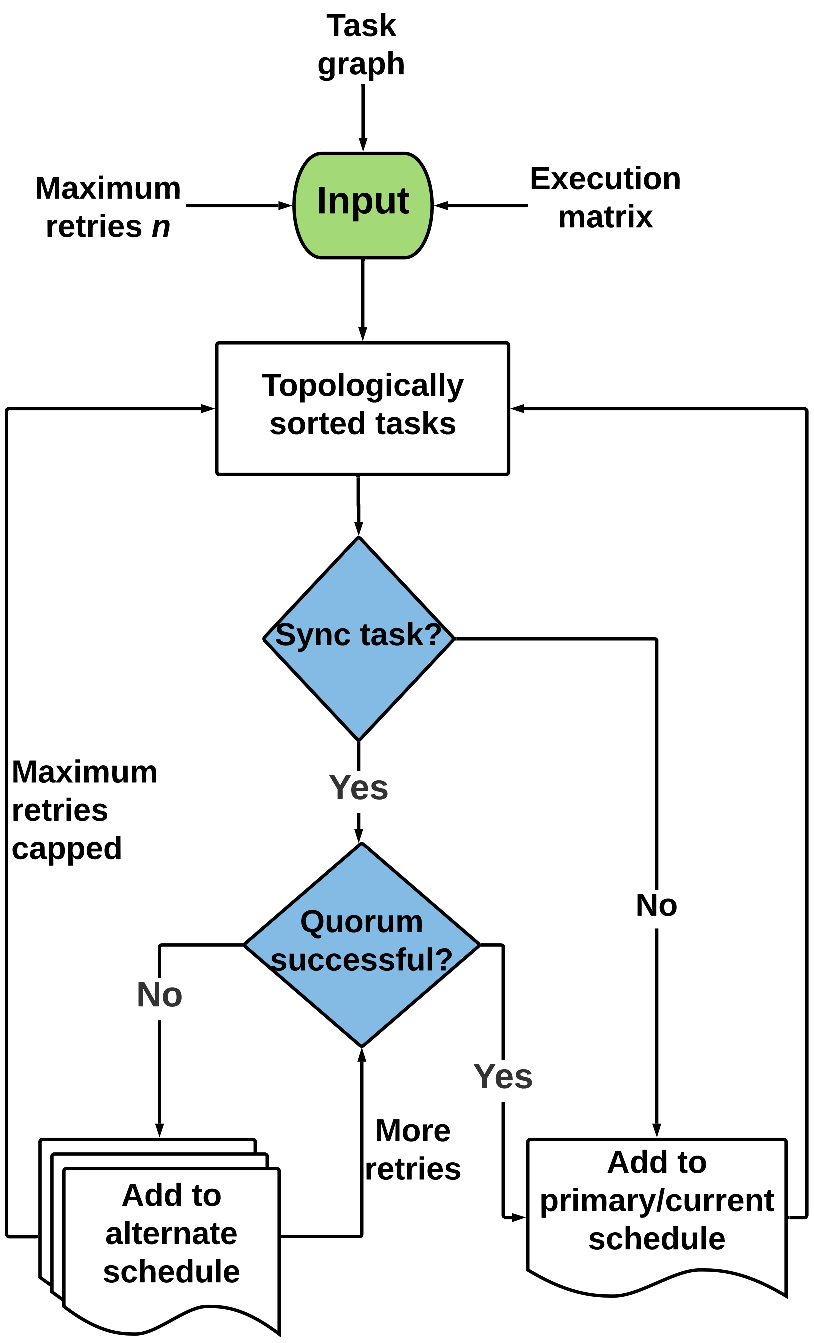


Fig. 4.1 Flow of the static synchronization scheduling algorithm.

Algorithm 1: Static synchronization scheduling algorithm

```
Input: Task graph $G$ and maximum quorum retries.
Output: Primary and alternative schedules.
set $S$ = $top\_sort(G)$ and schedule counter $m = 1$
$StatSchd(S,\ m,\ t_{avail})$:
  while $S \neq \phi$ do:
    if $type(T_{next}) = c2w_s$:
      $s\_schedule(T_{update}, m, t_{avail})$ and compute $\tau$
    if $type(T_{curr}) = c2w_s$:
      $s\_schedule(T_{quorum}, m, \tau)$ ;                    ▷ calls QuorumCheck(m)
      if retries not exceeded:
        $StatSchd(S,\ m,\ t_{avail} + \lambda)$
      else if retries exceeded:
        $remove(S,\ T_{curr})$
        $StatSchd(S,\ m,\ t_{avail})$
      $s\_schedule(T_{curr}, m, t_{avail})$
    else if $type(T_{curr}) = c2w_a \parallel type(T_{curr}) = w_l$:
      $s\_schedule(T_{curr}, m, t_{avail})$
$QuorumCheck(m)$:
  if $quorum = passed$:
    continue schedule $m$
  else:
    switch to schedule $m$++
```

Table 4.2 Explanation of functions in SSSA.

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

The update sending tasks are scheduled on the workers one task before getting to a synchronization point (*Lines 6–7*). The early sending of the *updates* allows the controller to process the messages while the workers are busy executing tasks, resulting in less wait times (wasted work cycles). After calculating the expected variation τ in execution progress across workers, the quorum check task T_{quorum} is scheduled (*Line 9*). If quorum passes, we schedule the sync task (*Line 15*) and proceed with the current schedule (the m^{th} schedule) (*Line 20*). However, if quorum fails and there are still quorum retries left, an alternative schedule is generated (*Line 22*) and the synchronization process is continued after waiting for a delay λ (*Line 11*). If there are no retries left, the synchronization task is failed, we generate an alternative schedule and then proceed to the next task (*Lines 13–14*). $c2w_a$ and w_l are scheduled immediately they get to the head of the queue in S (*Lines 16–17*).

4.4 Dynamic Synchronization Scheduling Algorithm (*DSSA*)

The DSSA shown in Algorithm 2 is run at the controller and executed at runtime. Assumptions such as prior knowledge of the estimated execution time of tasks and the structure of the task graph made in the static synchronization scheduling algorithm are relaxed in the dynamic algorithm. DSSA assumes that we cannot accurately predict task arrival pattern, thus, scheduling decisions are made on the fly as tasks become available. Fig. 4.2 shows the flow of DSSA. We use $[W]$ to depict the scheduling actions that occur at the worker.

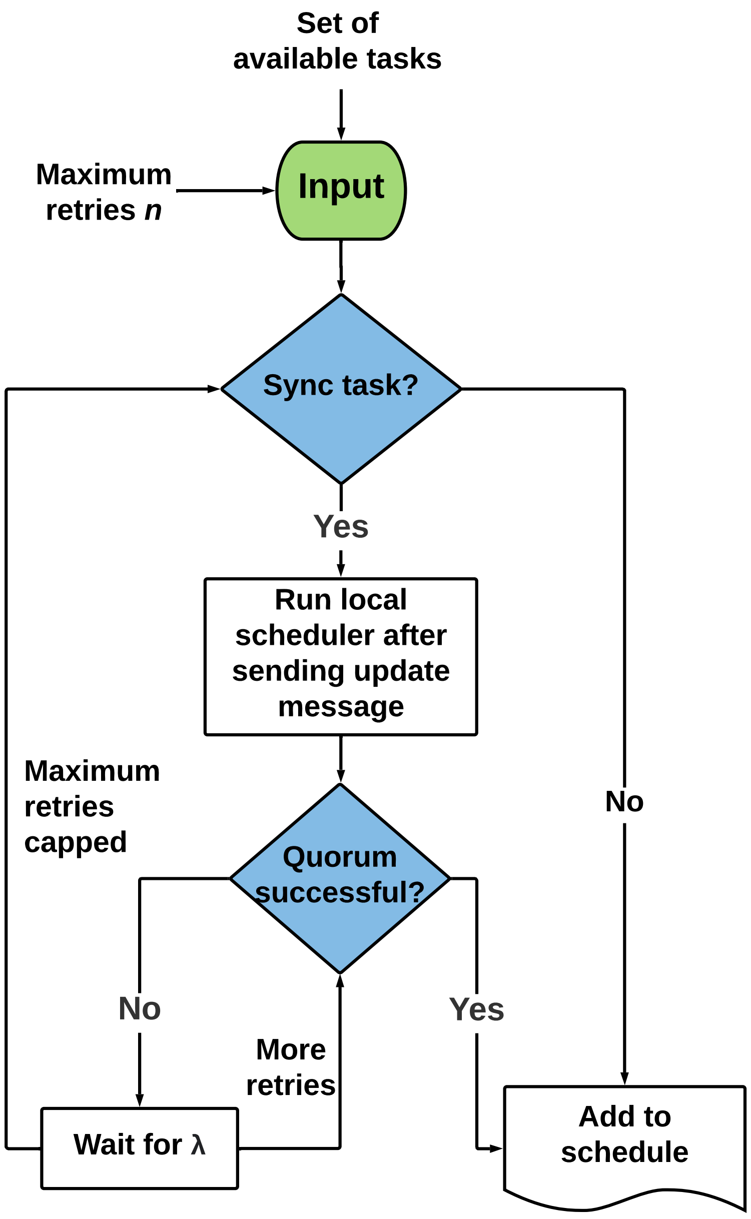


Fig. 4.2 Flow of the dynamic synchronization scheduling algorithm.

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 $T_{curr} \neq \phi$ do:
6      if $type(T_{curr}) = c2w_s$:
7        $d\_schedule(T_{update}, t_{avail})$   [W]
8        $\delta = compute\_slack()$
9        $LocalSchd(\delta)$   [W]
10       $d\_schedule(T_{quorum}, t_{avail})$   [W]
11       if $quorum = failed$ && retries not exceeded:
12         $DynaSchd(A,\ t_{avail} + \lambda)$
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 $type(T_{curr}) = c2w_a \parallel type(T_{curr}) = w_l$:
19       $d\_schedule(T_{curr})$   [W]
```

Table 4.3 Explanation of functions in DSSA.

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

Whenever we get to a synchronization point, the controller prompts the workers to schedule the update sending task (*Line 7*). Before quorum checking takes place, the controller triggers the local scheduler on ready workers (*Line 9* – Algorithm 3) to check if a local task can be run provided the execution time t_l of the local task is less than the computed slack δ (*Line 8*) in workers' available times. This is done to minimize wasted work cycles on the workers that reach the synchronization point faster than others.

Quorum checking (*Lines 10*) is performed by the workers by probing the controller to know if there is a successful quorum or not. If quorum is successful, the $c2w_s$ task is scheduled on all available workers to run at the same time. If the quorum fails and there are retries left, the workers wait for some time λ before attempting the synchronization process again (*Line 12*). If all retries fail, the synchronization task is failed (*Line 14*) and execution continues (*Line 15*). $c2w_a$ and w_l tasks are scheduled to run on at the earliest available times on the workers (*Line 19*).

4.5 Micro Batch Synchronization Scheduling Algorithm (*MBSSA*)

The MBSSA derives attributes from both SSSA and DSSA as shown in Fig. 4.3. SSSA makes assumptions that the task graph and execution time distribution are known but makes more pre–informed scheduling decisions, which becomes stale after a while, especially for long–running applications. DSSA whereas assumes that we do not have enough information on the tasks coming into the system nor their execution time distribution and thus makes scheduling decisions on the fly. MBSSA aims at grouping dynamically arriving tasks and runs the static scheduling scheme on the group of tasks. MBSSA is particularly good for systems where the communication cost between workers and controllers is high, and the arrival pattern of tasks can be accurately predicted to some degree. The arrival pattern of tasks is important because we do not want to keep tasks for too long before forming a micro–batch.

MBSSA dynamically groups a set of incoming tasks into a microbatch and schedules them statically. This eliminates the staleness in the scheduling decisions of SSSA. New tasks that come after a microbatch is formed are grouped into a new microbatch and prepared for scheduling. Micro batches could be formed based on a predefined size or time slice (tasks that arrive within a particular time interval

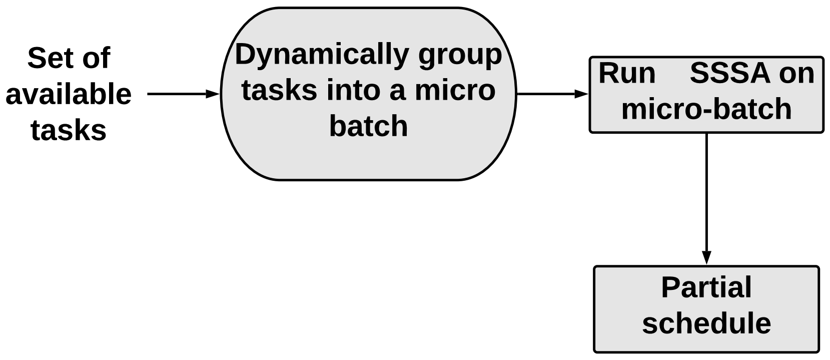


Fig. 4.3 Flow of the micro batch synchronization scheduling algorithm.

are grouped into a micro batch).

4.6 Local Scheduler

The local scheduling is an optimization scheme developed to minimize the waiting of workers that get to the synchronization point earlier than other workers. Since the workers do not communicate directly with each other but only with the controller, workers on getting to a synchronization point have no information about other workers. So, by running a local scheduler *LocalSchd(δ)*, a worker compares its current time with the predicted finish time δ across all workers as shown in Algorithm 3.

Algorithm 3: Local scheduling algorithm for task–based synchronization

```
1 LocalSchd(δ):
2 $L$ = {local worker task queue}
3 while $get\_tasks(L) = w_l$ do:
4   if $t_{avail} + t_l \leq \delta$:
5       $schedule(w_l)$
6       $revise\_available\_times(t_{avail},\ max\_t\_avail)$
7   else:
8       continue
```

The local scheduler would run a local task (provided there is one) if the gap between the predicted availability of the other workers δ and the local node's availability t_{avail} is greater than the local task's length. Otherwise, the local scheduler would not schedule any local tasks and let the worker sit idle.

4.7 Experiments and Results

The purpose of the experiments is to use simulations to evaluate the performance of SSSA, DSSA, and MBSSA under a variety of different conditions.

4.7.1 Simulation Procedure

We parameterized many variables for the simulations. The simulation parameters are derived from system, application and environmental configurations. The wide range of parameter variation allows for a higher degree of exploration into many aspects of the system. The simulation parameters are

1. *Prediction accuracy of execution time*: how close the predicted execution time of a task is to the actual runtime.
2. *Execution time variance*: deviation of the execution time of a task on different workers.
3. *Sample ratio*: ratio of workers probed in sampling–based quorum checking.
4. *Synchronization task frequency*: frequency of synchronization tasks (periodic/sporadic).
5. *Synchronization task density*: number of synchronization tasks in the task graph (lightly synchronized/heavily synchronized).
6. *Update processing cost*: the individual cost of processing messages from workers during quorum checking.

7. *Synchronization degree*: ratio of the total workers required to pass quorum.
8. *Quorum retries*: number of times quorum checking is permitted to be repeated before failing the sync task.
9. *Node failure rate*: the probability that a worker will temporarily fail before rejoining at a later time.

We use randomly generated and differently structured task graphs in our simulations. A task graph consists of 30 tasks (asynchronous, synchronous, and local). We introduce machine heterogeneity and task heterogeneity into the simulations. Task execution times are generated using a normal distribution with a mean value of 100 ms and a variance of 20 ms. Machine heterogeneity is included by varying the execution time of a task while running the simulations. The variation follows a normal distribution, with the predicted value chosen as the mean and the deviation specified as a parameter in the simulation. In *MBSSA* micro batches are formed by grouping a set of tasks with a size of 5. To model the heterogeneity of the workers, we use a normal distribution to represent the execution time variation of a particular task across different workers [164]. τ is computed by adding from one up to three times (covering from 68.27% to 99.73% of the normal distribution according to the three–sigma rule) the standard deviation to the mean execution time.

4.7.2 Simulation Results

The following parameters are fixed in the simulations unless otherwise stated. The number of independent runs of each simulation is 100 while each task graph is continuously run in each simulation for 200 times, we set the communication cost between machines to 20 ms, α is set to 0.7 and λ set to 50 ms. We randomly fail machines with a probability of 0.1 after each task. The probability of a new machine joining is set at 0.1, but machines can only join at the start of the execution of a new run of the task graph. This is done to ensure that joining machines will have all the necessary data required to run all tasks down the task graph.

We measure the following parameters in our simulations. *Execution time*: the time taken for a single run of a task graph. We divide the total execution time of a simulation by the number of runs. In our results, we normalize the execution time by the number of sync points. *Quorum attempts*: the total number of times the quorum check process was attempted at all synchronization points. *Failed sync tasks*: The total number of sync tasks that failed after exceeding the total number of quorum retries.

To explore the impact of the accuracy in the prediction of the execution time of tasks on *SSSA, DSSA* and *MBSSA*, we varied the standard deviation from the predicted value from 0 to 20. A standard deviation of 0 means a perfect prediction of the execution time of the tasks. The accuracy of prediction reduces as standard deviation increases.

The number of quorum attempts increases as the prediction accuracy reduces for all three algorithms. As the prediction accuracy reduces, *SSSA* performs worse than *DSSA* in terms of number of quorum retries as seen in Fig. 4.4. The number of quorum retries increases for *MBSSA* at a much lower rate compared to *SSSA*. The average execution time of *SSSA* and *MBSSA* increases at a much faster rate than *DSSA* as the prediction accuracy decreases. At lower prediction accuracy values (higher standard deviation), *DSSA* performs better than *SSSA* and *MBSSA* as shown in Fig. 4.5.

To determine the optimal number of quorum retries for a particular system, environment and application setup, we varied the number of quorum retries keeping other parameters constant. Fig. 4.6 shows the number of failed sync tasks for different number of quorum retries. *SSSA* has the highest number of failed sync tasks, closely followed by *MBSSA* and finally *DSSA* for a single quorum retry. The number of failed sync tasks reduces for all three algorithms as the number of quorum retries increases. All the algorithms converge at 3 quorum retries, at which point the number of failed sync tasks is close to 0 and stays almost constant.

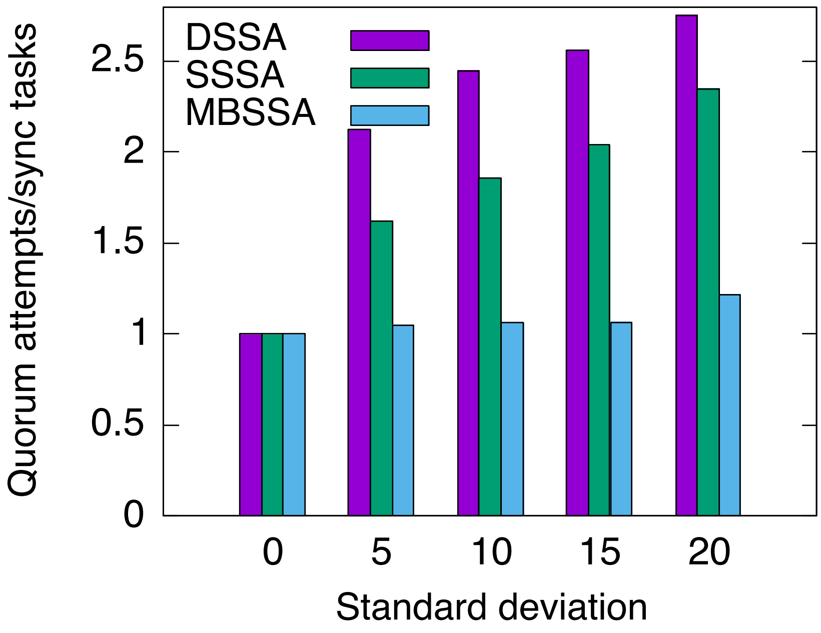


Fig. 4.4 Quorum attempts for varying prediction accuracy.

The average execution time for different quorum retry values is shown in Fig. 4.7. The rate of increase in the average execution time for *MBSSA* reduces drastically after 2 quorum retries, while it was after 3 quorum retries for *SSSA* and *DSSA*. After 2 quorum retries, the average execution time of *SSSA* becomes higher than *MBSSA* while *DSSA* has the lowest average execution time regardless of the number of quorum retries.

We investigate the impact of changing the update processing cost on *SSSA*, *DSSA* and *MBSSA*. An increasing cost of update processing has negative impact on *DSSA* as the number of quorum attempts increases as update processing cost increases. For *SSSA*, the number of quorum attempts decreases as the update processing cost increases till it gets to a steady point after an update processing cost of 10 ms while *MBSSA* remains unaffected as the update processing cost changes as evident in Fig. 4.8.

The average execution time of *SSSA* increases at a much faster rate compared to *DSSA* and *MBSSA*. *DSSA* has a lower average execution time than *SSSA* up

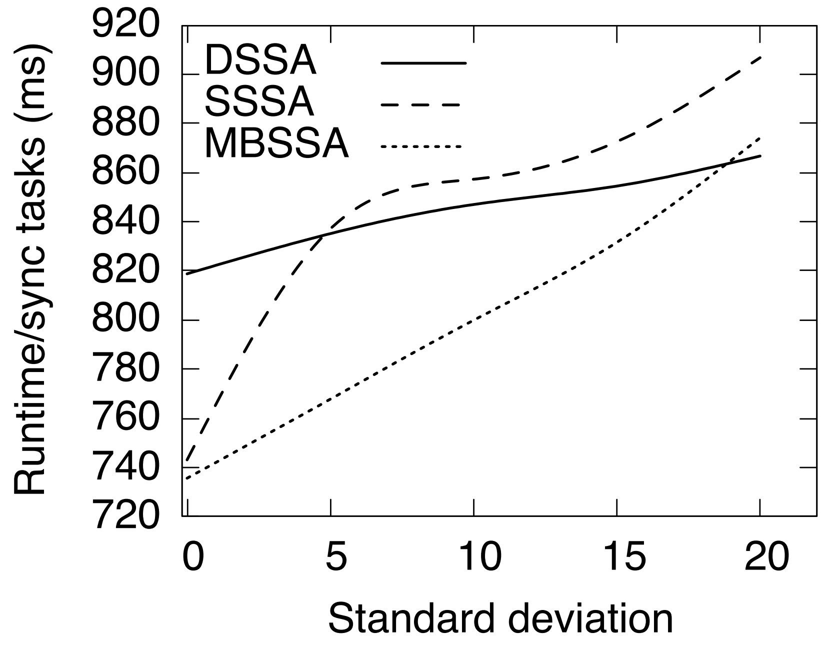


Fig. 4.5 Average execution time for varying execution time prediction accuracy.

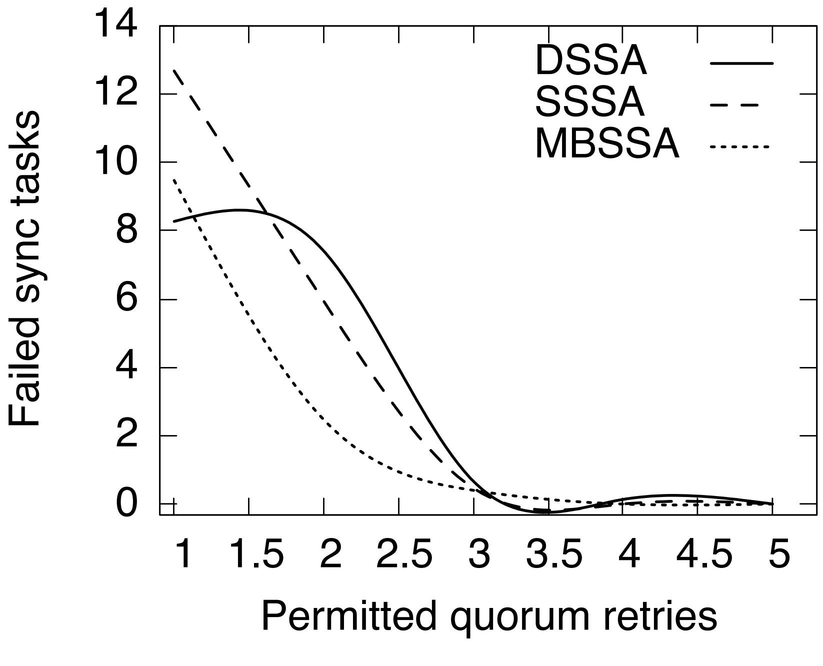


Fig. 4.6 Failed sync tasks for varying number of quorum retries.

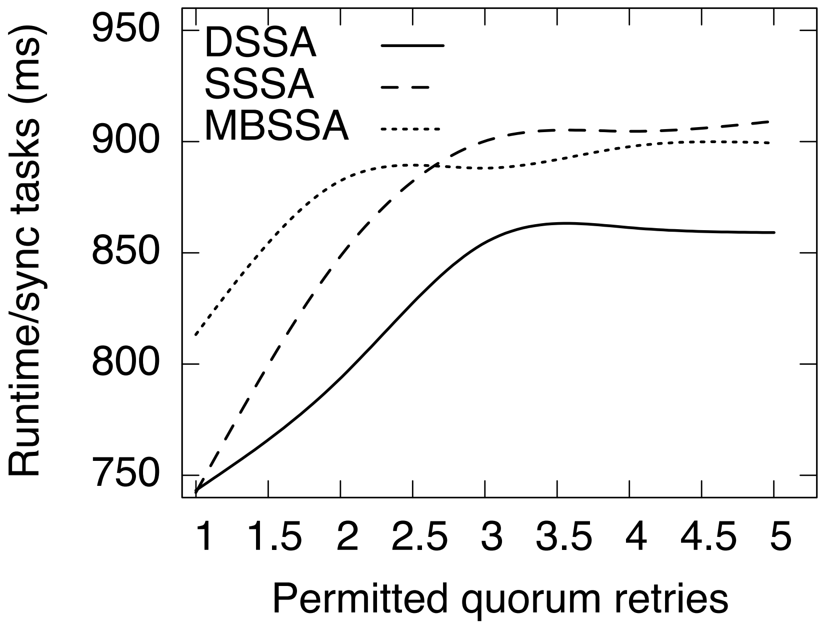


Fig. 4.7 Average execution time for varying number of quorum retries.

until a processing cost of 10 ms. At higher update processing costs, *SSSA* has a higher average execution time than *DSSA*. *MBSSA* has a lower average execution time compared to both *SSSA* and *DSSA* regardless of the cost of update processing as shown in Fig. 4.9.

We vary the density of sync tasks in task graphs used in the simulation. We set the density between 3 and 7 sync tasks for lightly synchronized and between 8 and 12 sync tasks for heavily synchronized. *MBSSA* performs better in terms of average execution time compared to *SSSA* and *DSSA* for lightly synchronized task graphs while *DSSA* performs better than *SSSA* and *MBSSA* for heavily synchronized task graphs as the number of machines increase as shown in Fig. 4.10 and Fig. 4.11.

We set the frequency of sync tasks to every 4 task for the periodic sync task frequency and randomly vary the frequency for sporadic sync task frequency. We show the results for varying the frequency of sync tasks in Fig. 4.12 – 4.13. *SSSA* has the highest average execution time for both the periodic and sporadic sync task frequency. *DSSA* has a lower average execution time for sporadic sync

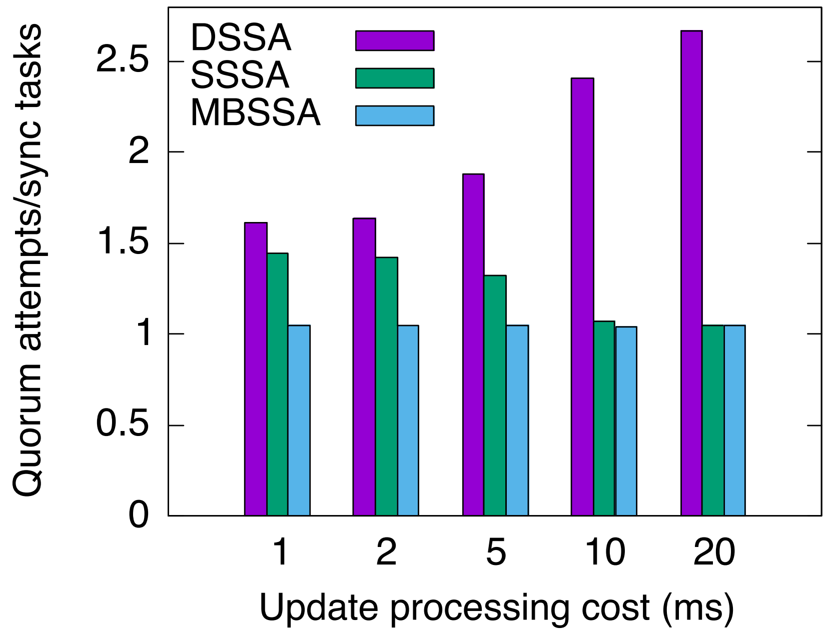


Fig. 4.8 Quorum attempts for varying update processing cost.

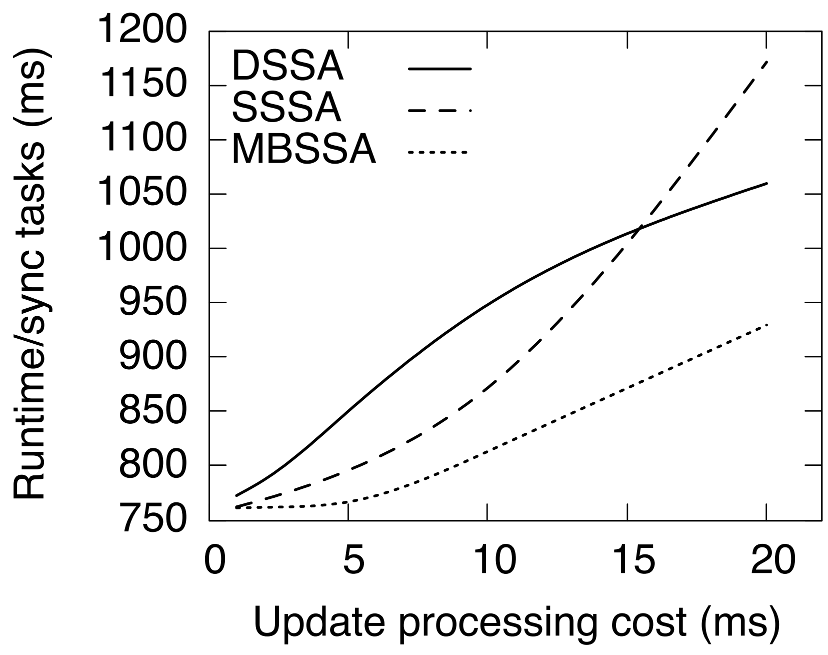


Fig. 4.9 Average execution time for varying update processing cost.

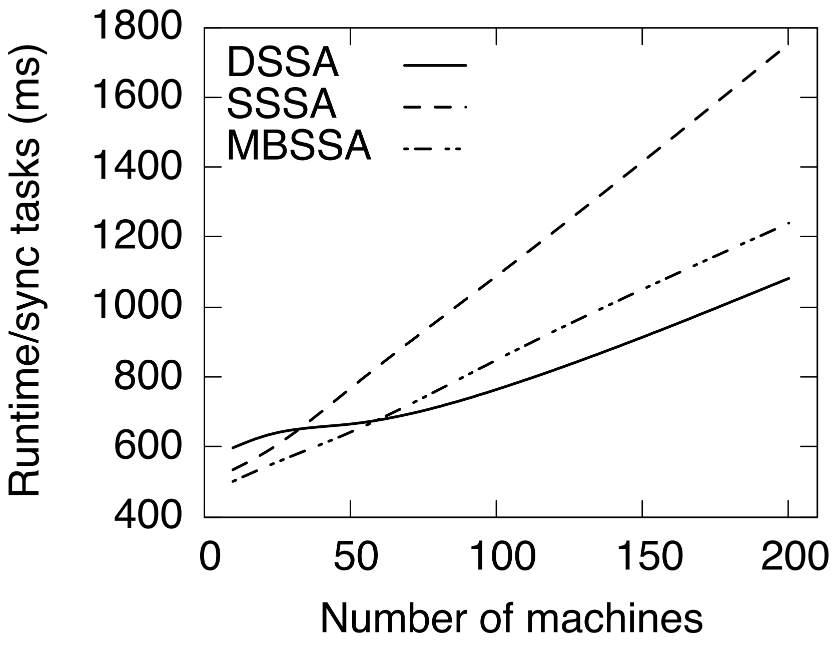


Fig. 4.10 Average execution time vs number of machines for heavy synchronization.

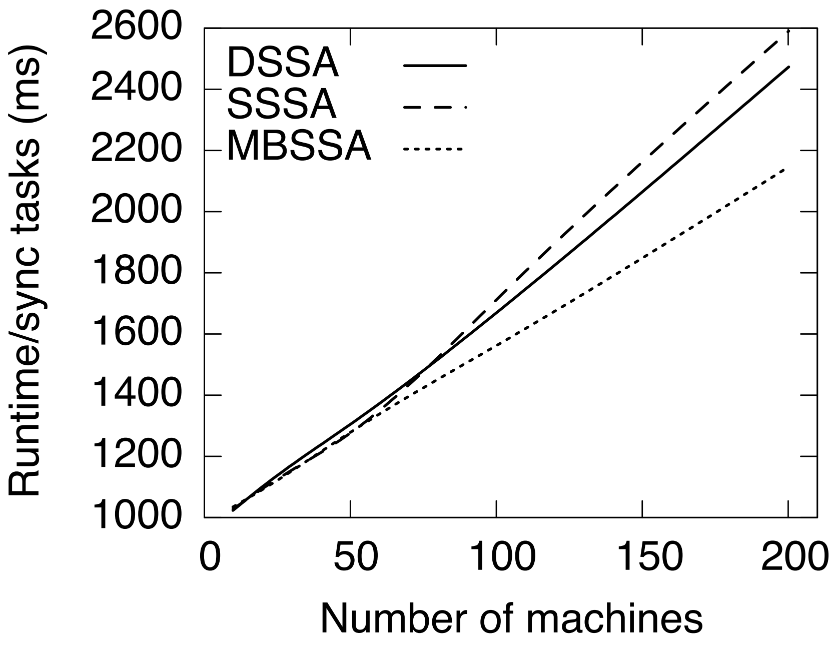


Fig. 4.11 Average execution time vs number of machines for light synchronization.

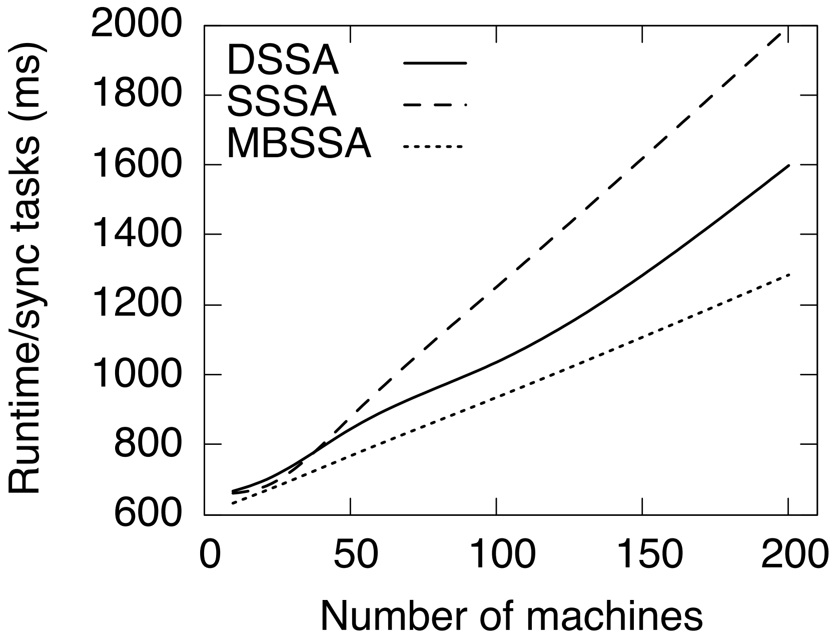


Fig. 4.12 Average execution time vs number of machines for periodic sync frequency.

task frequency and higher average execution time for periodic sync task frequency compared to *MBSSA*.

We compare the results of using the naive quorum checking and sampling–based quorum checking with a sample ratio of 0.3 as shown in Fig. 4.14. DSSA–SAMPLE has a much lower execution time compared to DSSA because in DSSA–SAMPLE only a subset of the machines is involved in the status updates and quorum check process. However, this comes at a cost as shown in Fig. 4.15 where in some cases (20% of the time) the percentage of machines that run the sync tasks is lower than the actual required percentage (70%).

We measure the impact of disconnection on our algorithms by varying the probability of nodes leaving and later rejoining the system. DSSA, SSSA, and MBSSA are affected by increasing node failure rates similarly. The number of failed sync tasks increases as the node failure rate increases as shown in Fig. 4.16.

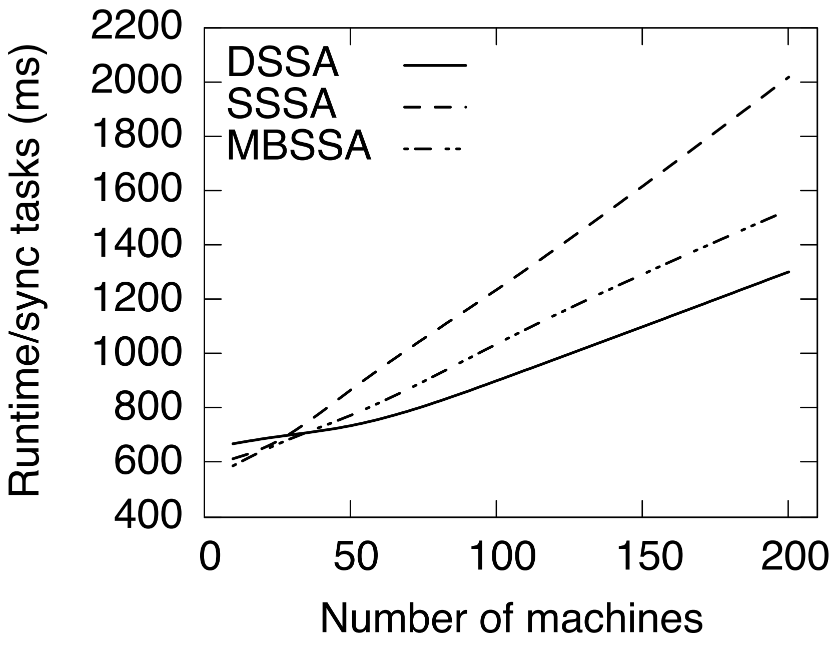


Fig. 4.13 Average execution time vs number of machines for sporadic sync frequency.

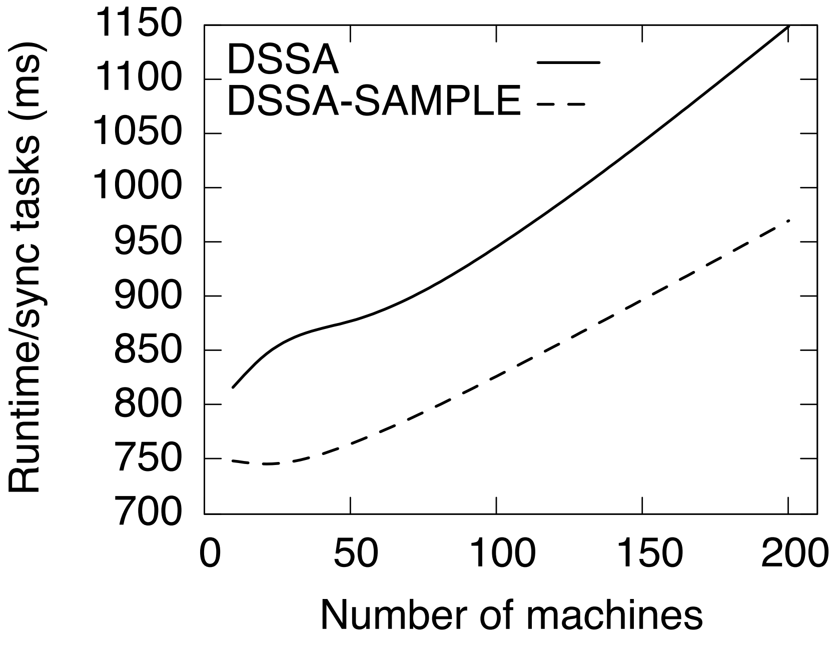


Fig. 4.14 Execution time for DSSA and DSSA–SAMPLE with varying number of machines.

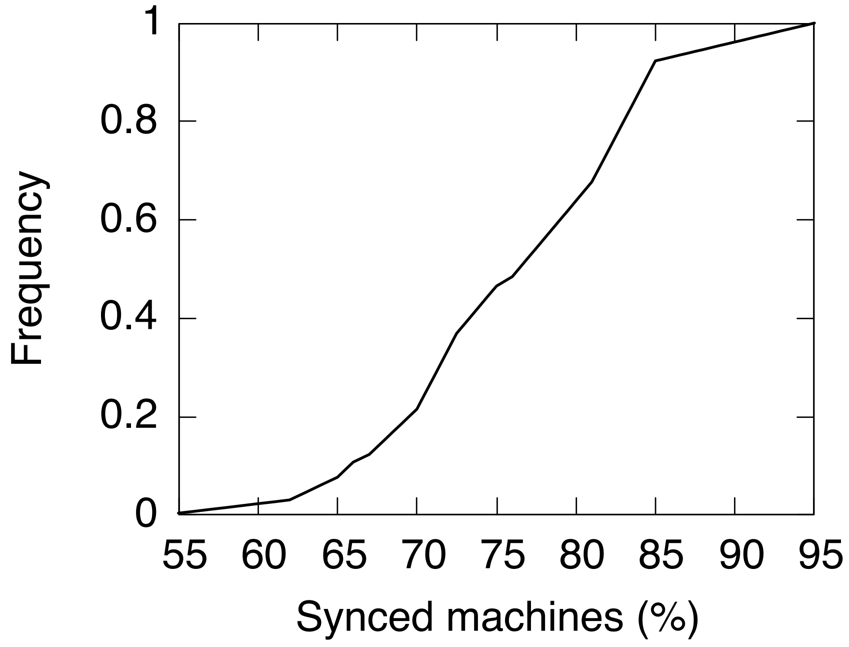


Fig. 4.15 CDF showing the percentage of machines that ran sync task.

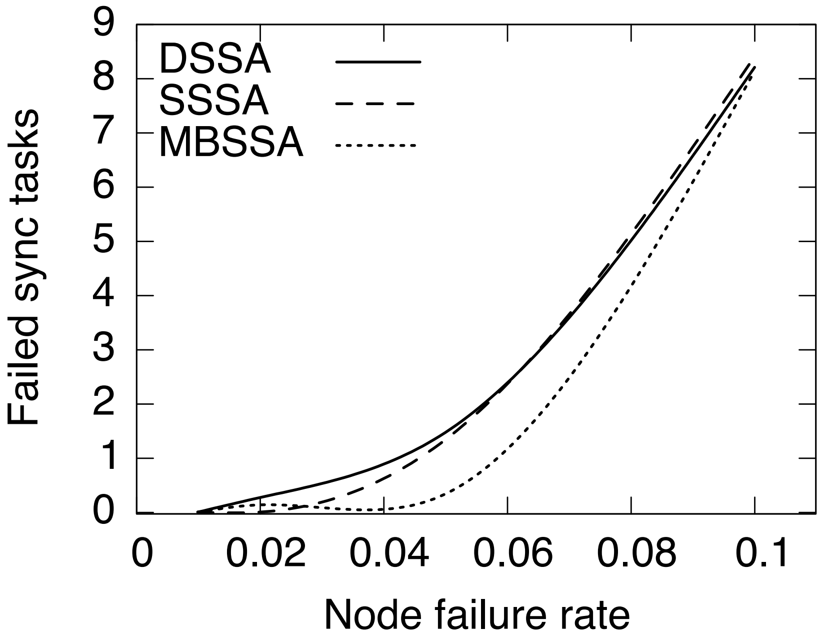


Fig. 4.16 Failed sync tasks for varying node failure rates.

4.8 Discussions and Summary

The prediction accuracy of the estimated execution time of tasks greatly impacts the performance of the algorithms. *SSSA* is the most affected, because the schedule is generated using the predicted values of task execution times. The larger the deviation of the actual runtime value from the predicted value, the worse the performance of *SSSA*. At higher prediction accuracy values *SSSA* and *MBSSA* outperform *DSSA*, but at lower prediction accuracy values, *DSSA* outperforms both *SSSA* and *MBSSA* in terms of average execution time because scheduling decisions of *DSSA* are made on the fly and thus less impacted by changing the prediction accuracy of execution time estimates.

We deduce from our simulations that there is an optimal value for the number of quorum retries that should be permitted. The number of failed sync tasks becomes constant for all three algorithms after a certain number of quorum retries (3), increasing the number of quorum retries beyond this value results in no observable performance gain as shown in Fig. 4.6. Likewise, the rate of increase in the average execution time of all three algorithms becomes so small (close to 0) after a certain number of quorum retries (3) as shown in Fig. 4.7. The value of quorum retries after which no significant gain in performance is evident is the optimal value for the number of quorum retries.

The number of quorum attempts for *SSSA* and *MBSSA* are similar under heavy synchronization but under light synchronization, *SSSA* is higher for lower number of workers. *DSSA* performs better than *MBSSA* in terms of execution time when the sync task frequency is sporadic, but worse when the frequency is periodic. Sampling–based quorum checking is more suited for cases where execution time of an application is important than the synchronization degree (that is, the number of workers that need to show up before running a sync task). All the algorithms are affected by very similar rates for increasing node failure rates in terms of failed sync tasks. Table 4.4 gives a summary of the three task–based algorithms.

Table 4.4 Comparison of the different task synchronization algorithms.

Metric	SSSA	DSSA	MBSSA
Task graph knowledge	Full knowledge expected. Take graph known at compile time	Tasks arrive in system dynamically during runtime	Tasks are grouped into batches as they arrive
Schedule generation time	Compile time	Runtime	Runtime
Runtime prediction accuracy	Very high accuracy needed for good performance	Fairly good accuracy is enough for good performance	Average accuracy is enough for good performance
Deployment scenario	Suitable for predictable and controlled systems such as IIoT	Suitable for rapidly changing system such as vehicular clouds	Suitable for systems where arrival pattern of tasks is known.

Chapter 5

Redundancy-Based Synchronization Schemes for Fog–Controlled Internet of Things

5.1 Overview

In this chapter, fault–tolerant controller–based schemes for synchronizing the execution of the tasks across a cooperating set of IoT devices are presented. With a controller–based scheme, one of the issues is the location of the controller itself. Fault tolerance is achieved using time redundancy and component redundancy. The recent emergence of fog computing as a major complementary technology to IoT is making it an ideal candidate to host our controller. The primary focus of this chapter is on developing such synchronization schemes that guarantee the desired quality of synchronization, QoS_{ync} subject to the conditions under which synchronization is deemed successful. We use a publish–subscribe status update scheme to minimize the communication overhead in reaching synchronization. Quorum checking is done to ensure the required degree of synchronization is met. We propose two redundancy–based dynamic synchronous scheduling algorithms with different synchrony requirements, and we evaluate the proposed algorithms using extensive trace–driven simulations and compare them with existing approaches.

5.2 Synchronous Scheduling Schemes with Redundancy

Two dynamic synchronous scheduling algorithms with redundancy built into them for scheduling tasks with varying time requirements on a bunch of IoT nodes are developed. The unique aspects of the algorithms are an adapted publish–subscribe status update scheme, quorum checking, redundancy, and the local scheduler, which are explained in the following subsections. The focus of the algorithms is to ensure that synchronous tasks are scheduled to run at the same time across all workers. We make minor modifications to the node model as shown in Fig 5.1. Workers can be grouped based on processing attributes or physical location, depending on the application's needs. The task model remains the same as in Section 1.2.

One input to the synchronous scheduling algorithms is a set of available tasks. The algorithms are expected to output a schedule of tasks that minimizes the overall execution time and ensures that the desired degree of synchronization is met with respect to the quorum requirements. A description of the notations used in this chapter is shown in Table 5.1.

5.2.1 Status Update

Workers need to update the controller of their availability to partake in synchronization upon reaching a synchronization point. The controller processes this information serially due to its single–threaded nature. This causes the message and communication overhead to scale linearly with the number of workers. The overhead incurred can significantly increase the time to achieve synchronization as the number of workers increases.

To mitigate this problem, we adapt the well–studied publish–subscribe

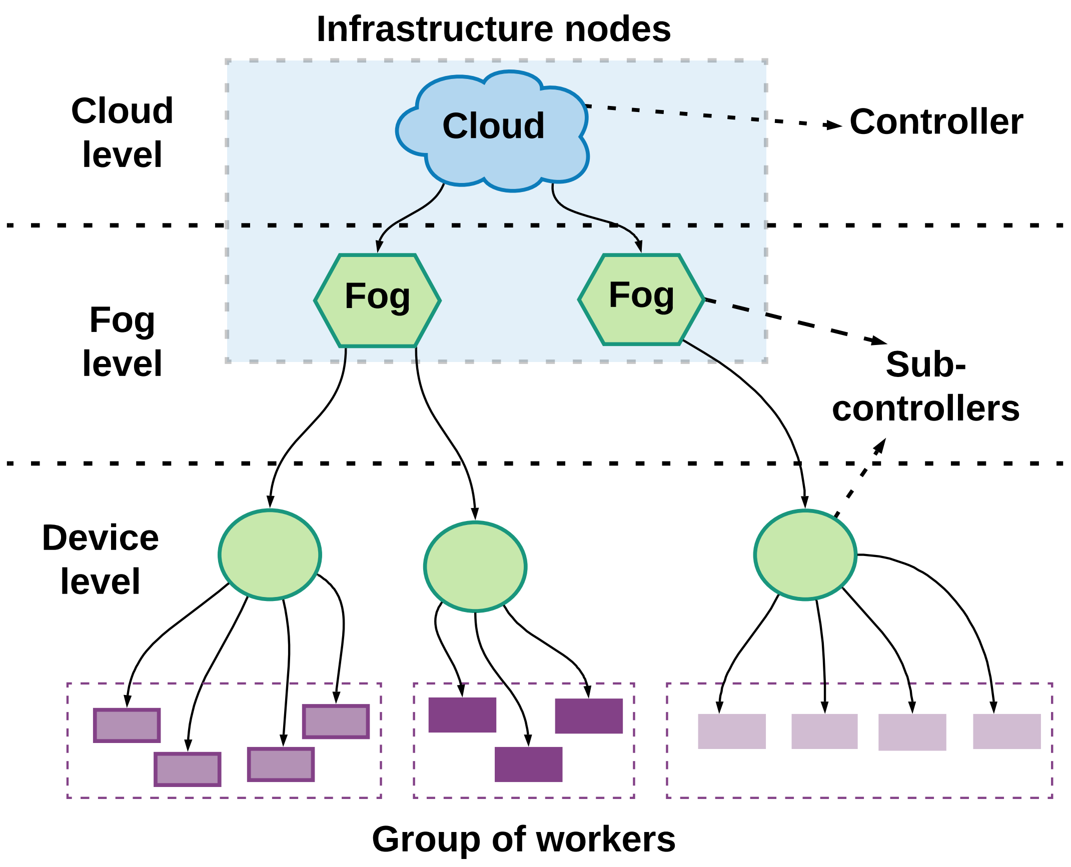


Fig. 5.1 Multi–level hierarchical node model showing controller, sub–controllers and worker nodes with worker nodes grouped.

Table 5.1 Notations for redundancy–based synchronization algorithms.

Symbol	Description
T_{curr}	current task to be scheduled
T_{rbq}	ratio–based quorum check task
T_{cbq}	cluster–based quorum check task
T_{update}	status update task
t_{avail}	available time of a worker
λ	time delay before re–attempting quorum
τ	predicted quorum check time

scheme [165, 166, 167] to reduce the number of update messages sent to the controller. Worker nodes are assigned a logical group which they join, and each group is assigned a local broker. Workers in a group publish their availability to the local broker. They also subscribe to the broker to know the peer availability. Whenever a worker detects that the required number of peer workers are present, it will publish a group availability message to the broker. The controller subscribes to the group availability message but not to the local availability. The controller thus ends up processing far less with the broker.

5.2.2 Quorum Checking

Quorum checking is done by workers to probe the controller on whether the required conditions for proceeding to run the synchronization task are met. We consider two types of quorum conditions. The first is based on the ratio of workers available, and the second is based on cluster representation. A worker is said to be available if it has finished executing its previous task and is physically present to run the next task. The two types of quorum checking are ratio–based (launched on workers by running T_{rbq}) and cluster–based (launched on workers by running T_{cbq}) quorums, both of which are previously explained in the taxonomy in Chapter 3.1.

5.3 Synchronous Scheduling Algorithms with Redundancy

As seen in Fig. 5.2, upon getting to a synchronization point, the controller sends a sync call (`syncCall()`) to the workers and the workers update the controller of their status (`pushStatus()`).

The controller computes the predicted quorum check time τ based on the status update from workers and sends the time to local schedulers on workers (`sendQCT`(τ)). The local scheduler (`localSCD`(τ)) on a worker tries to minimize wait time by comparing the available time of the worker with τ and estimates if a local task (provided there is one in the local task queue) can fit within the gap. Asynchronous and local worker tasks are scheduled as they become the current task at the earliest available times on workers. The pseudocode for the algorithm is shown in Algorithm 4. We use $[W]$ to depict the scheduling actions that occur at the worker.

We consider two types of redundancy–based synchronous scheduling algorithms. The first uses time redundancy (where synchronization is attempted based on waiting and a capped number of retries), and the second uses component redundancy (where we expect at least a certain number of workers within a cluster to be present before the quorum can be passed).

5.3.1 Synchronous Scheduling Algorithm with Time–Based Redundancy

Here, the synchronization degree represents the ratio of workers that must be available before the synchronous task can be run. Two important parameters of the algorithm are the wait time λ (specifies how long to wait if the desired number of workers are not available) and the maximum number of quorum retries permitted per synchronization point.

Whenever the ratio–based quorum check task T_{rbq} is scheduled (*Line 9*), the controller computes the ratio of available workers. If the ratio of available workers

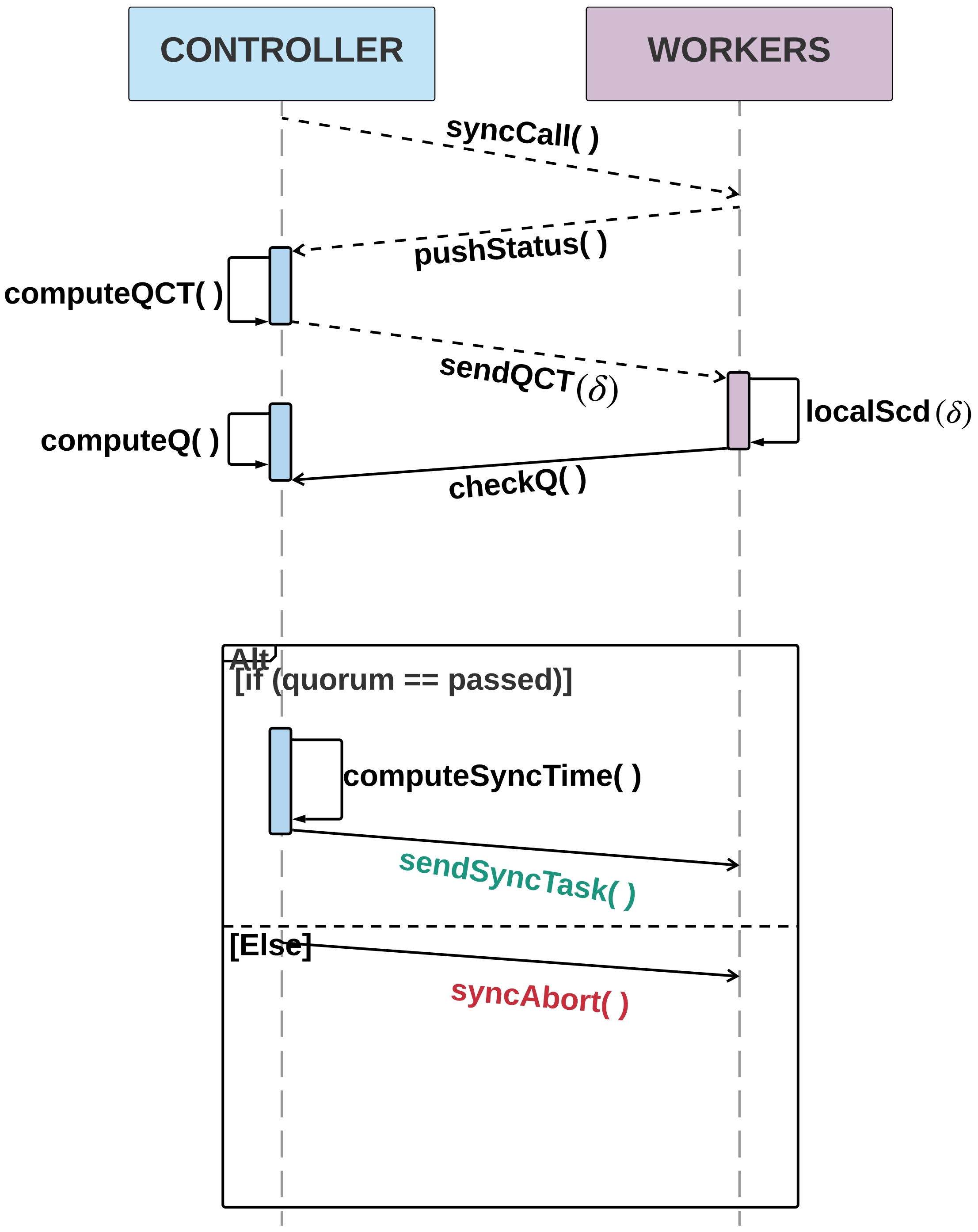


Fig. 5.2 Sequence diagram showing the interactions between the controller and workers at synchronization point.

is greater than or equal to the expected synchronization degree (*Line 12*), the controller computes the expected synchronous task start time and the synchronous task is scheduled.

Algorithm 4: Pseudocode for redundancy–based synchronous scheduling algorithms.

```
Let $A$ = set of available tasks
$SyncSchd(A)$:
  **while** $T_{curr} \neq \phi$ **do**:
    **if** $type(T_{curr}) = c2w_s$:
      $pushStatus()$   $[W]$
      $\tau = computeQCT()$
      $localSCD(\tau)$   $[W]$
      $revise\_available\_times(t_{avail}, max\_t\_avail)$   $[W]$
      $schedule(quorum\ check\ task)$   $[W]$
      **Time–based redundancy:**
      r = $getAvailableRatio()$
      **if** $r >= sync\ degree$:
        $schedule(T_{curr})$   $[W]$
      **else if** $r < sync\ degree$ && retries available:
        $SyncSchd(A, t_{avail} + \lambda)$
      **else if** $r < sync\ degree$ && no more retries:
        $syncAbort()$   $[W]$
      **Component–based redundancy:**
      e = $computeRedundancy()$
      **if** $e >= required\ redundancy$:
        $schedule(T_{curr})$   $[W]$
      **else**:
        $syncAbort()$   $[W]$
      $remove(A, T_{curr})$
    **else if** $type(T_{curr}) = c2w_a \parallel type(T_{curr}) = W_l$:
      $revise\_available\_times(t_{avail}, get\_ctrl\_avail\_time())$   $[W]$
      $schedule(T_{curr})$   $[W]$
  $remove(A, T_{curr})$
```

If the synchronization degree is not met and there are more retries left (*Line 14*), we delay for time λ and retry the process. However, if the synchronization degree is not met and there are no more retries left, the synchronous task is failed

(*Line 17*). An example where time–based redundancy is useful is in bridge health monitoring. If while trying to take measurements a failure occurs, the application can proceed with its processing and repeat the bridge strain measurement at a later point in time.

5.3.2 Synchronous Scheduling Algorithm with Component–Based Redundancy

In this algorithm, workers are always part of a logical cluster as they move around in the system (e.g., vehicles). At a synchronization point, after the update of the workers' status at the controller (*Line 5*) and the triggering of the local schedulers (*Line 7*), the workers run the quorum check task (T_{cbq}). The controller computes and checks whether the required level of redundancy is met by each cluster (*Line 19–20*). This computation is done by comparing the number of workers available in each cluster with the expected number of devices per cluster. In the event of a successful quorum, the synchronous task is scheduled to run. If the desired level of redundancy is not met, the synchronous task is failed (*Line 23*).

Here, a synchronous task is marked as successfully completed if and only if at least the required number of workers per cluster returns a result after executing the synchronous task. Thus, the synchronization result is abstracted at the cluster level. If the desired redundancy in output is not met, the synchronous task is considered to have failed. An example of where component–based redundancy is useful is in the drone transportation scheme. We do not want the drones to perform the transportation task if we do not have the desired number of backup drones available.

5.4 Experiments, Results and Discussions

First, experiments are conducted to measure the impact of controller location (i.e., fog or cloud) on synchronization. Then, further experiments are conducted to measure specific attributes of the performance of the proposed synchronization algorithms. Finally, the performance of the proposed synchronization algorithms is evaluated by comparing them with barrier synchronization [105, 106] and time slotted synchronization [108].

5.4.1 Impact of Controller Location on Synchronization

In this experiment, traces from the OpenCloud Hadoop cluster from Carnegie Mellon University Parallel Data Lab [1] are used. The workload is split into short tasks ranging from 0.5 s to 4 s and long tasks ranging from 5 s to 12 s. The controller–worker delay is varied from 5 ms to 500 ms. Performance is measured using the synchronization rate (SR) which is the number of synchronizations per unit time.

From Fig. 5.3, it can be observed that a task graph consisting of short jobs has much higher SR compared to a task graph consisting of long tasks. Additionally, increasing the controller–worker delay from 5ms to 500ms has very little impact on the task graph with long tasks compared to short tasks because the long tasks take a significant portion of the overall runtime, thus minimizing the impact of the controller–worker delay. A task graph with short tasks consisting of synchronous, asynchronous, and local tasks has a higher SR compared to the one consisting of only synchronous tasks. This is because with only synchronization tasks, status updates and quorum checking need to be performed at each synchronization point, therefore adding more overhead to the overall runtime. Having controllers closer to the workers increases the SR for short running tasks, thus making the case for fog–resident controllers as opposed to cloud–resident controllers.

5.4.2 Configuration of Synchronization Experiments

The following configurations are used to define the system, application, and environmental parameters. The wide range of parameter variation allows for a

[1]http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html

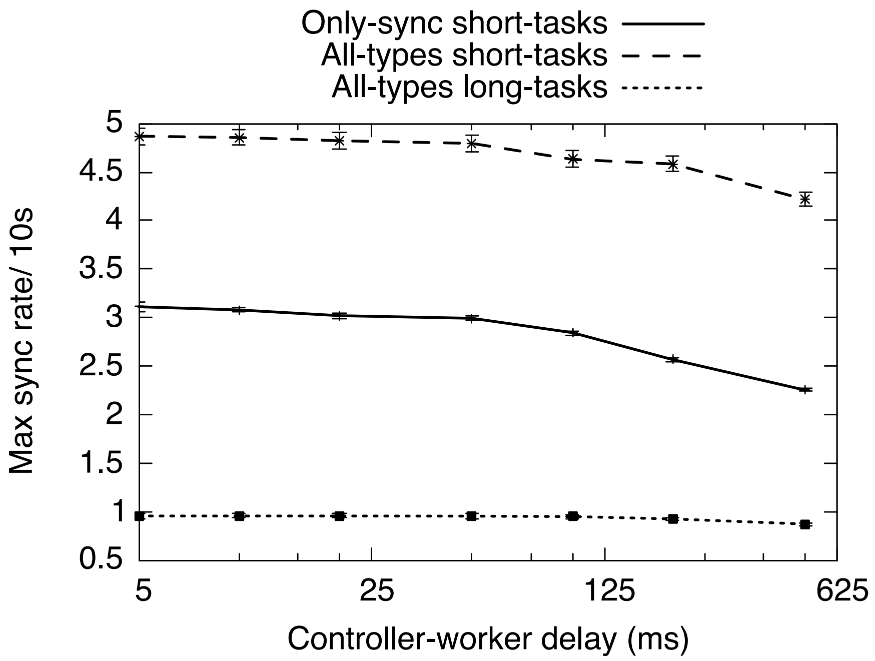


Fig. 5.3 Maximum synchronization rate per 10 s for varying controller–worker delays.

higher degree of exploration into many aspects of the system. Measurement traces from experiments using Dropbox between the period of 28th June to 3rd July 2012 [168] are used as the task dataset. The experiment consists of $900,000$ storage operations from different geographical locations and, thus, varying transfer speeds. This mimics an edge environment with varying network speeds and geographically distributed workers. The execution times vary from a minimum time of 23 s to a maximum time of 269 s.

To model the mobility of worker nodes, the Shanghai (China) taxi GPS report of Feb 20, 2007 [169] is used. The report consists of 4316 taxis reporting their location, speed, angle of movement and occupancy at given intervals over a period of 24 hours. Each taxi has a unique identification number. The taxi traces are preprocessed and only the location and timestamp details are extracted. The taxis upload their details at irregular intervals in the trace, varying from $15s$ to $63s$. Re–sampling is done at $30s$ and the position of the taxis are recorded at regular intervals, thus having a total of 2880 time points. To make it easier to map the location of workers to a 2–dimensional representation, we convert from the GPS decimal degrees longitude–latitude format to the Universal Transverse Mercator coordinate system, which represents locations on the earth's surface using a 2–dimensional Cartesian coordinate system.

The parameters used in the simulations are as follows. (i) *Synchronization degree*: ratio of the total machines required to pass quorum, (ii) *minimum cluster size*: the minimum number of workers that must be present in a cluster before it can be formed, (iii) *wait time*: The amount of time that should elapse before attempting quorum checking again, (iv) *quorum retries*: the maximum number of times quorum checking is allowed, (v) *worker size*: The maximum number of workers that can be present in the system at any point in time, (vi) *number of clusters*: The maximum number of clusters that can be formed at any point in time, and (vii) *prediction accuracy*: A measure of how accurately the predictor predicts the finish time of tasks across all workers prior to the synchronization point.

5.4.3 Default Parameter Values and Measurements

The following parameters are fixed in the simulations unless otherwise stated. The number of independent runs of each simulation is 100 while each task graph is continuously run in each simulation for 200 times, the communication cost between machines is set to 200 ms, status update cost is set to 1 s, synchronization degree is set to 0.7 and λ set to 20 s. Workers randomly fail with a probability of 0.1 after each task. The probability of a new machine joining is set at 0.1, but machines can only join at the start of the execution of a new run of the task graph. This is done to ensure that joining machines will have all the necessary data required to run all tasks down the task graph.

Task graphs consist of 30 tasks in total with varying number of synchronous tasks. 10 task graphs are used to represent different applications in our simulation runs. Heterogeneity among the worker nodes is introduced by making the execution time of a task on multiple workers follow a Gaussian distribution.

The following parameters are measured in the simulations. (i) *Runtime*: this is the time taken for a single run of a task graph normalized by the number of synchronization points, (ii) *extra quorum attempts*: the total number of times the quorum check process was attempted after the initial attempt at all synchronization points, and (iii) *failed sync tasks*: The total number of sync tasks that failed after exceeding the total number of quorum retries or due to incomplete results from clusters.

5.4.4 Scalability of the adapted publish–subscribe update scheme

The benefits of the publish–subscribe message update scheme is shown in Fig. 5.4 while varying the number of workers from 10 to 4000. Fig. 5.4 shows the runtime per synchronization point for the publish–subscribe and all–worker update methods for the ratio–based quorum checking. From the graph, it can be observed that as the number of workers increase, the runtime per synchronization point increases at a similar rate with respect to the number of workers for the all–worker update while for the publish–subscribe update there is no significant increase in the runtime per synchronization point as the number of workers increase. This is because regardless of the number of workers in the system, the number of messages sent to the controller using the publish–subscribe is bounded by the number of logical clusters formed.

5.4.5 Component redundancy

In component–based redundancy, the workers are grouped into clusters. To reach a quorum to execute a synchronous task, at least a given number of workers must be available in each cluster. However, for a synchronization task to be considered successful, at least one worker from each cluster must complete the execution of the task and return the expected output to the controller, otherwise, the synchronous task is considered to have failed.

The minimum required number of worker(s) per cluster is varied from 1 to

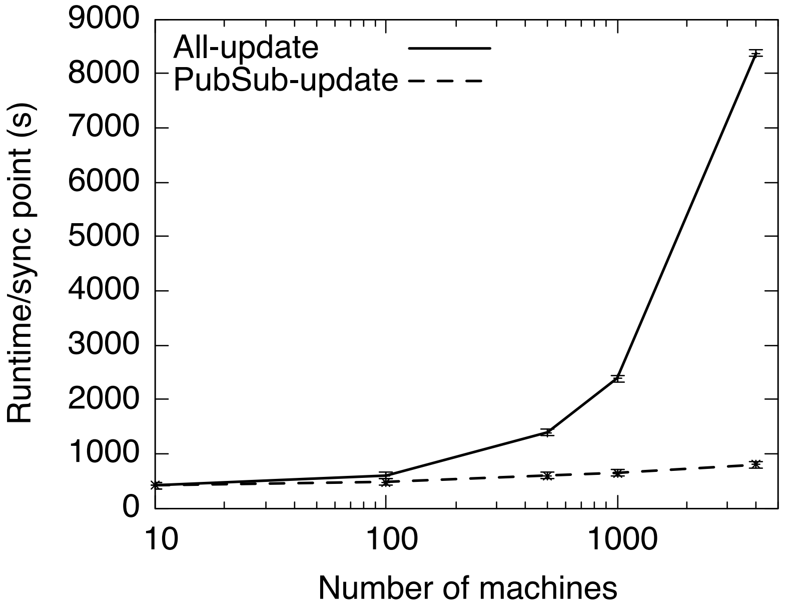


Fig. 5.4 Runtime per sync point comparing all–worker update sending and the publish–subscribe update scheme for time–based redundancy.

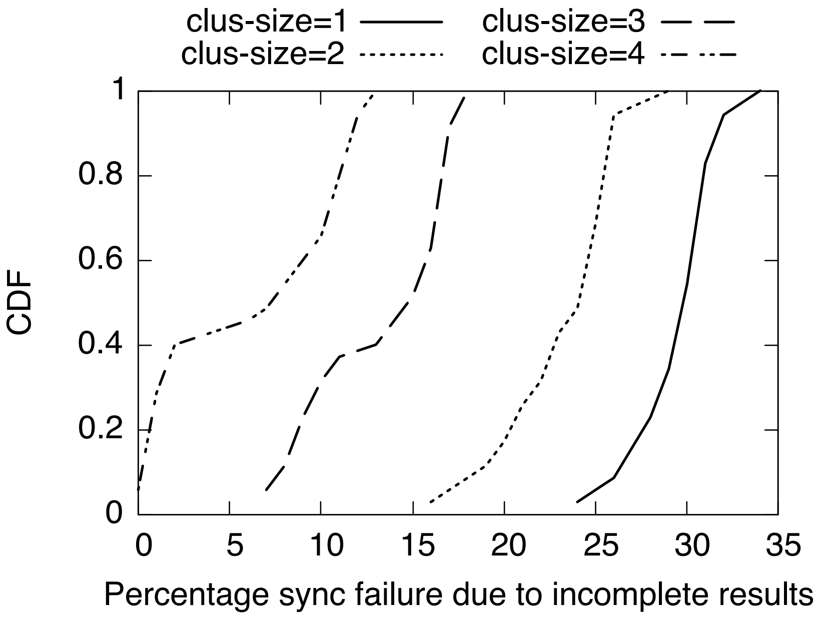


Fig. 5.5 CDF showing percentage of sync task failures due to incomplete results from clusters for minimum cluster sizes ranging from 1 to 4 for component–based redundancy.

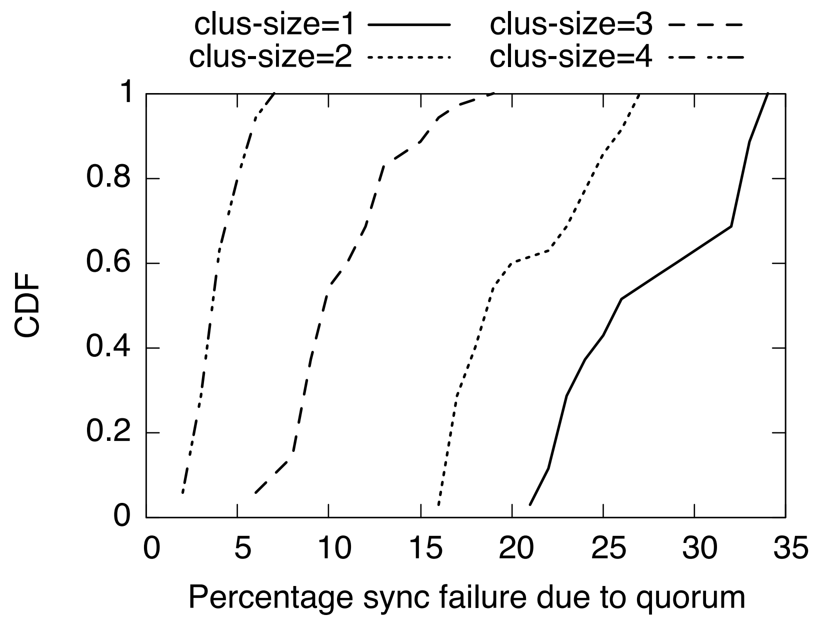


Fig. 5.6 CDF for the percentage of sync task failures caused by failed quorum for different minimum cluster sizes for component–based redundancy.

4, the synchronization task failures are measured and shown in Figs. 5.5 – 5.6. Figs. 5.5 and 5.6 show the synchronization tasks failure percentage due to failed quorum and incomplete results from clusters, respectively. The percentage of synchronization task failure due to failed quorum and incomplete results from clusters both decrease as the minimum cluster size increases. This is because the probability of having workers show up during the quorum check process increases as the minimum cluster size increases.

5.4.6 Impact of finish time prediction accuracy

The local scheduler uses the predicted availability of the workers to determine whether it can schedule a local task before starting the synchronous task. In Fig. 5.7, the impact of finish time prediction accuracy on the runtime per synchronization point and the synchronization task failure rate with component–based redundancy and publish–subscribe update schemes are

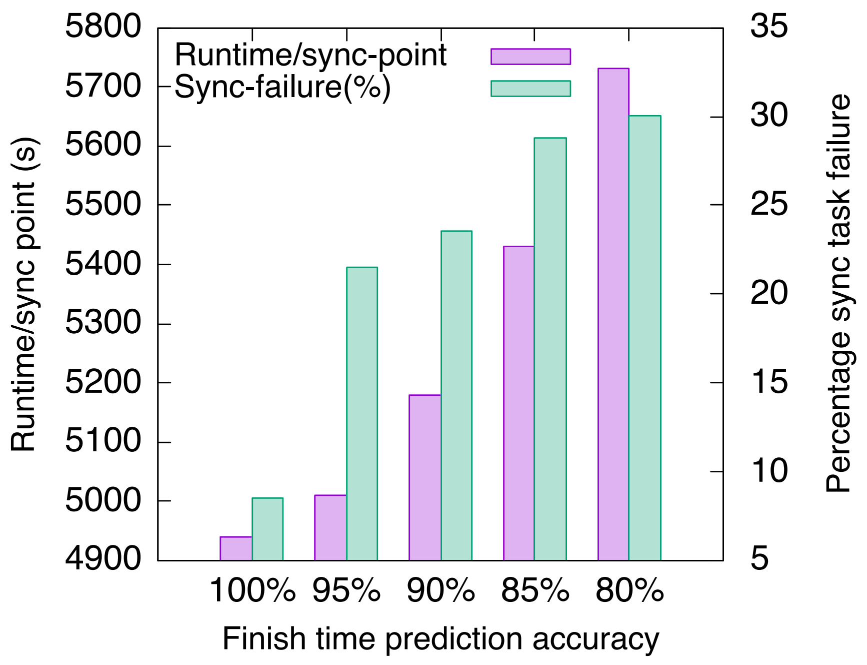


Fig. 5.7 Runtime per synchronization and percentage synchronization task failure for task finish time prediction accuracy varying from 80% to 100%.

measured.

When the prediction accuracy is 100%, it was observed that the smallest runtime per synchronization is 4940 s and the smallest synchronous task failure rate of 8%. While at a finish time prediction accuracy of 80%, an average of 5730 s was observed for the runtime per synchronization point and a 30% synchronization task failure. The runtime per synchronization point increases by 16% while the percentage synchronization task failure increases by 263.3% as the finish time prediction accuracy reduces from 100% to 80%. This shows the high impact that the finish time prediction accuracy has on the success of the synchronization task.

5.4.7 Performance Evaluation

The performance of the proposed synchronization schemes (time–redundant and component–redundant synchronization) is evaluated by comparing them with barrier synchronization and time–slotted synchronization. In barrier synchronization, on getting to a sync point, workers send an update message to the controller and wait until a signal is received from the controller saying that they can proceed to run the sync task. The condition for proceeding with the barrier execution is that all workers must reach the sync point.

In time–slotted synchronization, the workers' executions are split into time slots. Dedicated synchronization time slots are chosen with the hope that workers will be available to run the sync task at the specified time slot. The dedicated synchronization time slots are chosen by fixing the slots at $\mu + 1.5\sigma$ (accounts for an 86.6% accuracy), where μ is the average execution time and σ is the standard deviation.

Figs. 5.8 – 5.11 show the runtime per synchronization point and percentage synchronization task failure for the synchronous scheduling algorithm with the proposed time–redundant and component–redundant synchronization algorithms, barrier synchronization and time slotted synchronization, respectively. The minimum cluster size for component redundancy is fixed at 3 while comparing the synchronization schemes.

It can be observed from Figs. 5.8 and 5.10 that barrier synchronization has the highest runtime, followed by the time–redundant synchronization algorithm and then time slotted synchronization. Barrier synchronization takes longer because faster workers need to wait for stragglers at the barrier and cannot proceed until the slowest worker reaches the barrier. This is unlike the time–redundant synchronization algorithm that was proposed here, where the synchronization point is moved dynamically depending on the availabilities of workers and, the local scheduling mechanism is used to minimize wasted work cycles due to waiting. Time slotted synchronization is faster because there are dedicated synchronization slots that are not moved regardless of workers availability.

Figs. 5.9 and 5.11 show the percentage synchronization task failure due to failed quorum for the proposed time–redundant and component–redundant algorithms, and the time–slotted synchronization scheme. The percentage of sync task failures for the time–slotted synchronization scheme is higher than that for the proposed time–redundant and component–redundant synchronization algorithms as shown in Figs. 5.9 and 5.11 respectively. In Fig. 5.11, it can be observed that the percentage of sync task failures reduces as the number of workers increases. This is because there are more devices per cluster and thus, there are more redundant devices which increases the chances of reaching the desired quorum. Time slotted synchronization have higher sync task failure rates because the synchronization slots are fixed, and when there are straggling workers, synchronization cannot proceed.

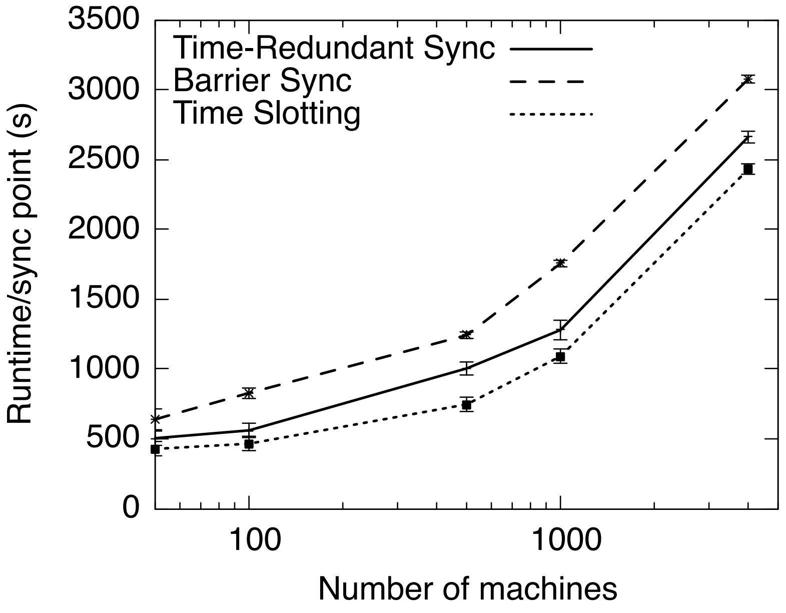


Fig. 5.8 Runtime per sync point comparing the proposed time–redundant synchronization algorithm with barrier and time slotted synchronizations.

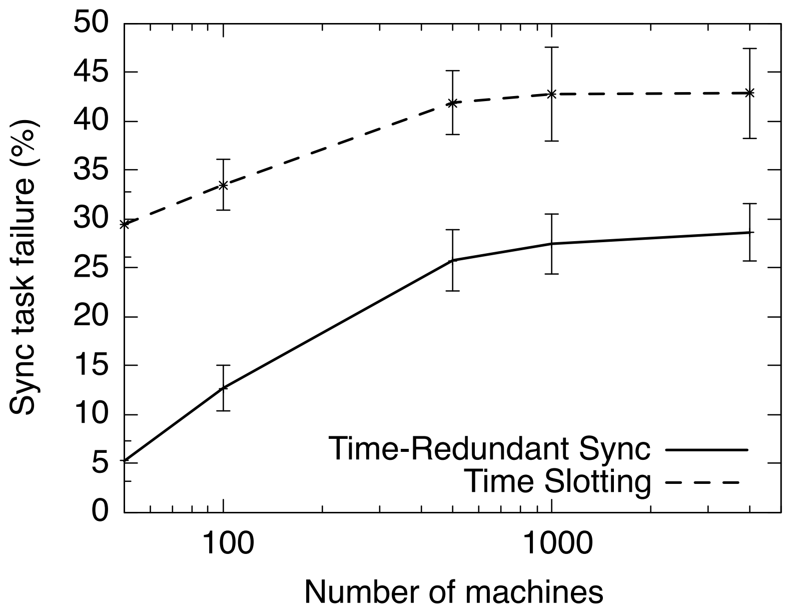


Fig. 5.9 Percentage of sync task failures caused by failed quorum for time–redundant synchronization algorithm vs time slotted synchronization.

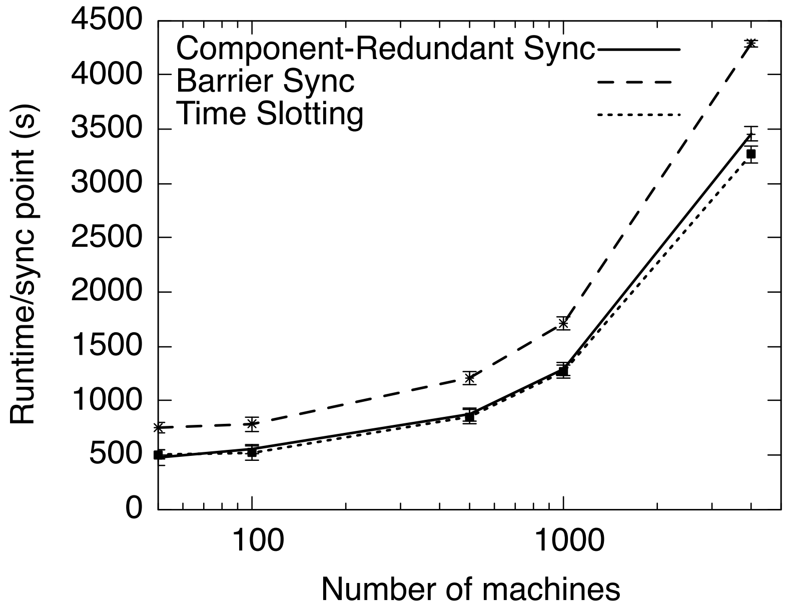


Fig. 5.10 Runtime per sync point comparing the proposed component–redundant synchronization algorithm with barrier and time slotted synchronizations.

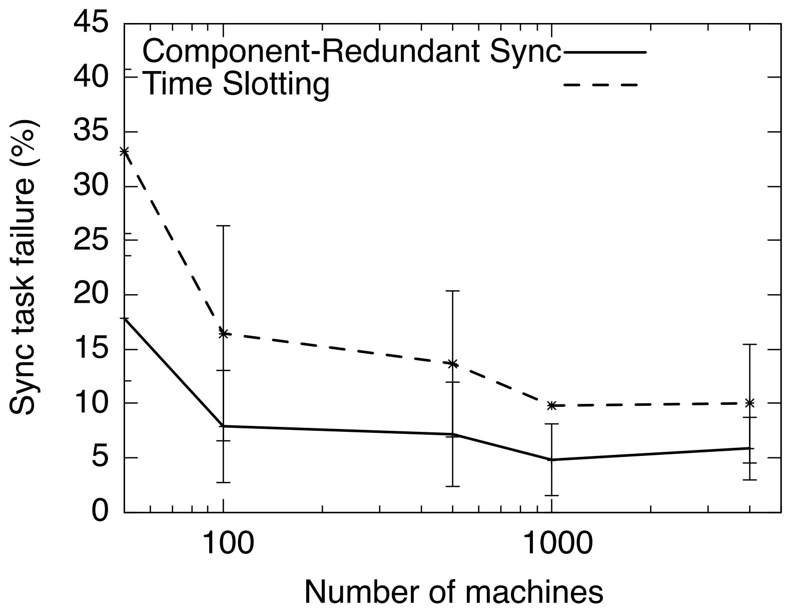


Fig. 5.11 Percentage of sync task failures caused by failed quorum for component–redundant synchronization algorithm vs time slotted synchronization.

5.5 Summary

The challenge for the synchronization algorithms is to keep synchrony between the task executions despite disconnections and task execution overshooting. The synchronization scheduling algorithms use two ideas: time–based and component–based redundancies. Experiments are conducted to evaluate the performance of the proposed algorithms and to explore the different trade–offs between the two approaches. We observe that time–based redundancy is suitable for applications where repeating the task executions is acceptable. Whereas, the component–based redundancy is needed for applications that cannot wait for task re–executions.

We find that using a publish–subscribe update scheme reduces the communication load on controllers; thus, effectively reducing the overall execution times of the synchronous tasks. It was observed that increasing the level of redundancy for component–based redundancy decreases the runtime and reduces the percentage of sync task failures. Likewise, the prediction accuracy of the finish time of tasks on the workers has a significant impact on the runtime and synchronization task failure. The proposed algorithms have shorter runtimes than barrier synchronization and have less synchronization task failures when compared to time slotted synchronization. A comparison of the proposed synchronization schemes and related works is shown in Table 5.2.

Table 5.2 Comparison of redundancy based synchronization schemes and related works.

Metric	**Time–Redundant Sync**	**Component–Redundant Sync**	**Barrier Sync**	**Time Slotting**
Straggler mitigation	Yes, using time delay	Yes, using redundant components	Yes, using dynamic slotting	No
Quorum requirement	Yes – ratio based	Yes – minimum cluster representation	No	Yes – full participation
Adaptability to dynamic systems	Very adaptable	Very adaptable	Minimal adaptability	Not adaptable
Local tasks	Yes	Yes	Possibly	No
Deployment scenario	Suitable for dynamic systems with tolerable time delay	Suitable for systems with tolerable redundant components	Suitable for systems with known execution patterns	Suitable for systems with homogeneous machine setup

Chapter 6

Fast Synchronization for Artificial Intelligence Application Tasks

6.1 Overview

Intelligent systems such as self–driving cars and robots can work either autonomously (i.e., doing all necessary processing by themselves) or collaboratively with other self–driving cars or robots and roadside infrastructures [170]. For instance, self–driving cars can collaborate with smart highways to increase the safety and overall performance under diverse scenarios. We consider the later scenario that needs fine–grained orchestration of all tasks that are executed by the different components in the larger intelligent system [171, 172]. These systems heavily rely on artificial intelligence (AI) or machine learning (ML) and need to process the AI/ML tasks within specified timing constraints. To meet task processing requirements, intelligent systems have often relied on cloud computing [173]. However, the recent emergence of edge computing has introduced an alternative to cloud computing that can host data closer to the devices and allow faster turnaround times for the AI/ML tasks [38, 35, 174]. We consider task synchronization (i.e., time alignment of task executions) across different computing nodes for edge–hosted AI/ML applications.

In this chapter, we develop a fast synchronization scheme by minimizing the number of messages required to reach synchronization using a late notification protocol and clustering. The clustering is done such that worker nodes with a high probability of staying tightly synchronized to some bounds are put in the same cluster. To achieve fast synchronization, we limit the controller involvement in making the synchronization decisions. We evaluate the proposed synchronization scheme using trace–driven simulations. We implement the synchronization scheme in Ray[1] and compare its performance with existing synchronization models for distributed machine learning.

6.2 System Model

6.2.1 Node Model

A hierarchical computing model is used in our system as shown in Fig. 6.1. Nodes at the bottom of the hierarchy are called workers. The nodes at upper levels of the hierarchy are called controllers. The controllers could be at three levels – device, fog, and cloud levels. Worker nodes can communicate with one another leveraging the underlying fast Wi–Fi local broadcasts. This architecture is suitable for achieving fast synchronization in AI application tasks and edge–based systems because it permits the controller to monitor the progress of workers and allows clustering of workers to reduce the message overhead.

Workers are expected to update the controller of their execution progress as they execute a given program. The controller uses these updates to cluster workers. Thus, workers are always part of a logical cluster. However, the clustering details are used only at a synchronization point. The workers know their clusters and the number of workers in each cluster in the system. Tight clock synchronization is assumed across nodes in all levels of the hierarchy. The tree structure can be

[1]https://docs.ray.io/en/latest/

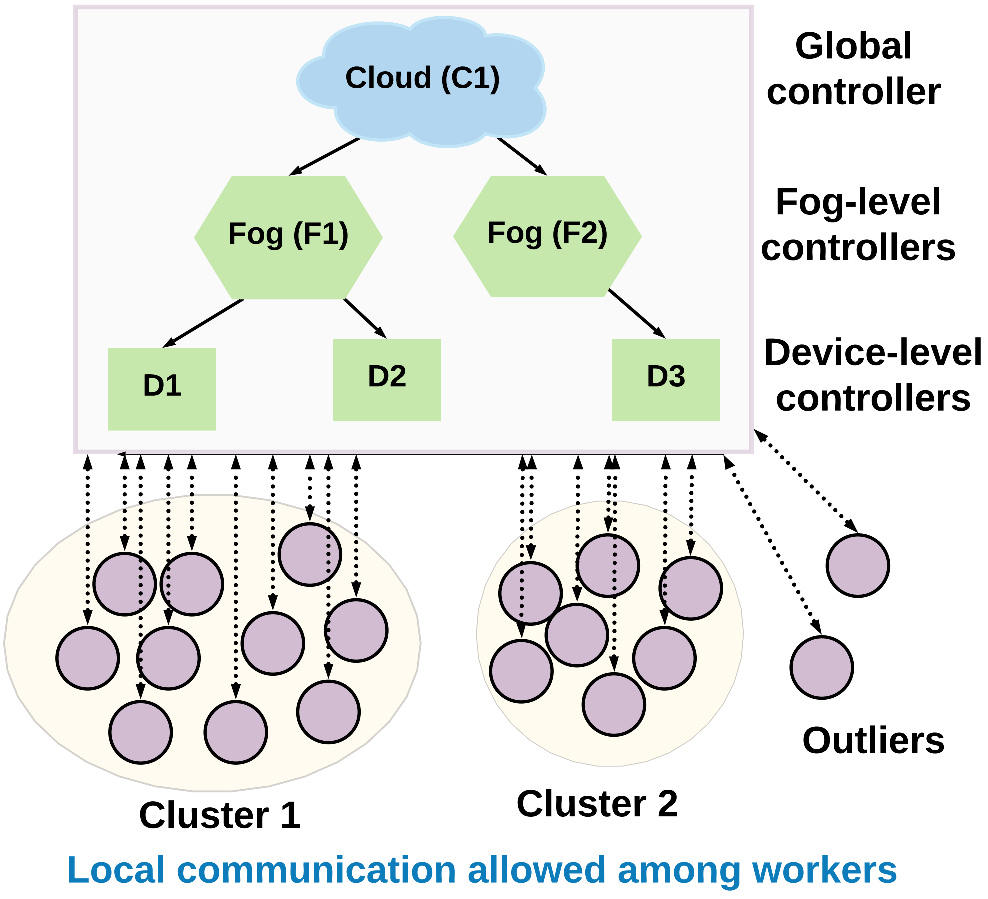


Fig. 6.1 Node model for fast synchronization in edge–based AI application tasks.

leveraged in ensuring that clocks on all nodes are synchronized.

6.2.2 Application Model

An application written for our system consists primarily of three task types – synchronous, asynchronous, and local tasks. The three tasks can be at the controller or at the worker. Synchronous and asynchronous tasks are remote calls triggered by the controller on a worker or by a worker on the controller. Local tasks are tasks that are triggered by a node on itself. Synchronous tasks triggered by the controller on workers require that all (or at least a certain ratio of) workers start the execution of the task at the same time. A return result of the execution of a synchronous task on the calling node is expected, unlike asynchronous and local tasks. In this work, we focus on remote calls from the controller to workers as it poses the most challenge of coordinating the activities of plenty workers. Synchronous, asynchronous, and local tasks on workers are denoted as T_{ws}, T_{wa} and T_{wl} respectively.

We include another task called the checkpoint task T_{cp}. The checkpoint task allows workers to report back to the controller on their execution time/progress. Check pointing is a way for the controller to monitor the progress of worker nodes and to make better clustering and scheduling decisions. Tasks are created such that workers can know when they get to the half–way point in the execution of the task. This is done such that workers can detect if they will be late in finishing the task.

6.2.3 Basic Game Model

We consider a system where nodes can be grouped into clusters such that nodes within a cluster are expected to remain synchronized within some specified bounds. All worker nodes must be connected to a controller to be considered part of the system. We can thus view the controller as a mediator (i.e., trusted third–party). The game is abstracted at the cluster level, that is, the game is between clusters and the strategies are at the cluster level. However, the utilities derived by workers are strictly based on the worker's participation in the synchronization process. The notations used in this work are shown in Table 6.1.

There are two basic choices that can be made by a worker upon getting to a sync point – $w$ait for sync or do not wait for sync (late notifications can be sent). The factors to be considered by a node in making a choice include the waiting time, the option of running another task, late notifications, and how fast the decision to sync can be made. The controller (mediator) uses the game in making synchronization decisions and scheduling the sync options. The controller specifies c sync options based on the choice that maximizes the total utility derived. At runtime, late notifications are used to change decisions, it gives a way for workers to skip a particular sync option and choose another option or quit synchronization.

Table 6.1 Notations used for synchronization game and analysis.

Symbol	Description
i	Worker $i, i = 1, 2, \ldots, N$.
C_k	Each worker is part of a cluster $k, k = 1, 2, \ldots, m$.
$\lvert C_k \rvert$	Size of cluster k.
$\lvert \mathbb{N}_s \rvert$	Number of workers that run a sync task.
$\omega_i(t)$	Cost of waiting for t time units by worker i at sync point.
t^i_{av}	Expected available time of worker i.
t^c_s	Sync time option $c, c = 1, 2, 3$.
$\mathbb{S}_c$	Utility derived by each worker for running sync task at sync option c.
$\mathbb{F}_c$	Cost of aborting sync at sync option c.
$\mathbb{L}_i(t_l)$	Utility derived by worker i for executing local task with execution time t_l.
$\delta(t)$	waiting cost before receiving late notification after time t.
α	Worker quorum for synchronization.

The various components of the game are:

1. Player: A strategic decision maker in the context of the game. These are two clusters in our case. They make the decisions whether to sync or not at any given sync option.
2. Strategy: The actions of players which include *wait for sync*, *no wait due to lateness* and *no wait due to late notification received.* All players will try to find the best strategy to maximize their payoff.
3. Payoff: This is the benefit or loss derived by a player based on a particular outcome of the game. Payoff is the difference between utility derived and

the cost incurred. Positive payoff is gotten only when synchronization is successful.

6.3 Clustering

We cluster nodes in the system for two main reasons – (i) to reduce the number of messages required for synchronization, and (ii) to help the controller in making better scheduling decisions. Clustering of workers is done by the controller using the reports received from workers whenever they reach a reporting point. Thus, the workers are continuously reporting their execution progress to the controller at each report point. The clustering of workers must be done such that workers with similar execution progress over time are put in the same cluster. That way, we are expecting that workers within the same cluster will remain closely synchronized.

In our synchronization game, we expect that only the largest two clusters will participate in synchronization. We therefore conduct initial experiments to motivate the need for clustering workers and to determine if the choice of having two major clusters is valid. We use the Density–Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm [175] to group workers into clusters. DBSCAN groups together points that are close to each other based on a distance measurement (usually Euclidean distance) and a minimum number of points. It marks the points that are in low–density regions as outliers.

We run four example neural network training tasks using Python's sklearn *MLPClassifier* and *MLPRegressor* on sklearn's Iris dataset continuously on 54 physical machines. We collect the runtime of each iteration for each task. We collect a total of $6,000$ data points and create clusters using 500 data points with overlapping data points of 100 for the next cluster created. The first clustering point uses data points $1 - -500$, the second clustering point uses data points $401 - -900$ (with data points $401 - -500$ overlapping with the first clustering point), the third uses data points $801 - -1300$ (with data points $801 - -900$ overlapping with the second clustering point) and so on. We thus have a total of

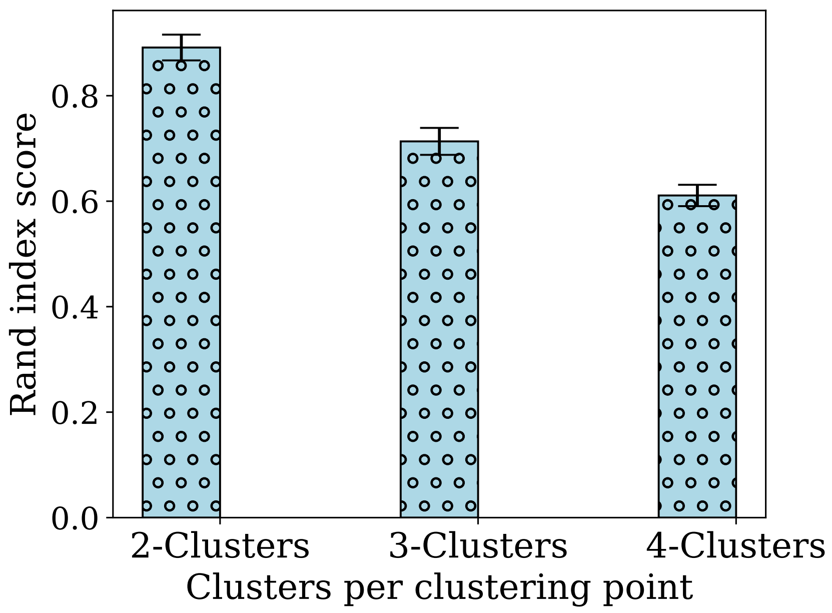


Fig. 6.2 Adjusted Rand Index scores for 2, 3 and 4 clusters per cluster point.

60 clustering points.

To evaluate the clusters created, we adjust the maximum distance and minimum number of samples per cluster to control the number of clusters formed. We measure the adjusted Rand index score (similarity measure between two clustering) for the cases where $2, 3$ and 4 clusters are formed as shown in Fig. 6.2. We use a pairwise comparison of all clusters formed at each clustering point in evaluating the adjusted Rand index score by making each clustering point the ground truth class label to be used as reference. The adjusted Rand Index score decreases as the number of clusters formed per clustering point increases. This is because more cluster stability is expected when we have two clusters only. As the number of clusters formed increase, there is a higher tendency of machines changing clusters from one clustering point to the other.

The inter– and intra–cluster distances when 2, 3 and 4 clusters are formed are shown in Fig 6.3. The average intra–cluster distance for 2 clusters formed

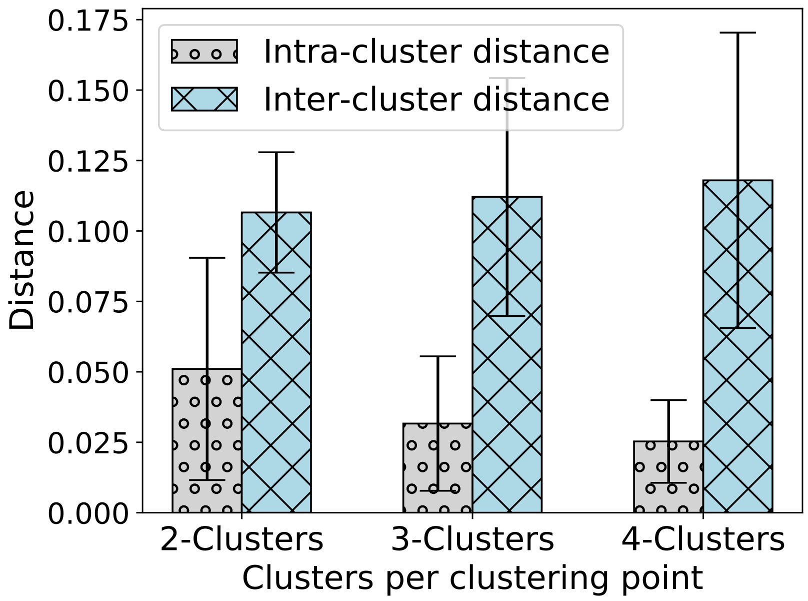


Fig. 6.3 Inter– and intra–cluster distances among clusters with 2, 3 and 4 clusters per point.

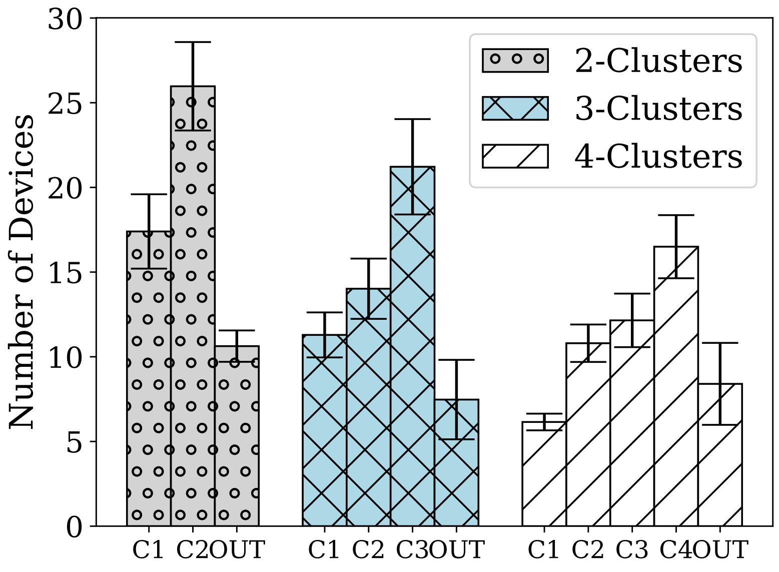


Fig. 6.4 Number of devices per cluster including outliers for 2, 3 and 4 clusters per point.

is higher than those of 3 and 4 clusters. The average inter–cluster distance for 2 clusters is lower than those of 3 and 4 clusters per clustering point. This is because for 2 clusters formed, each of the two clusters is covering a wider range and the maximum distance between machines within a cluster is higher. However, the distance between the two clusters will be reduced compared to 3 and 4 clusters per clustering point. When 2 clusters are formed per clustering point, an average of 20% of workers are outliers. For 3 clusters per clustering point the average outlier's percentage value is 15% and 17% for 4 clusters per clustering point as shown in Fig. 6.4.

6.4 Synchronization as a Game

6.4.1 Game Specification

We model the game as a cooperative game that is used to find the optimal strategy of players regarding synchronization. We choose a cooperative game because there are a different number of choices to be made depending on the choice of the other player. The goal is to form a quorum large enough for synchronization to happen. Thus, to achieve synchronization, the required quorum α of workers must be available to run the sync task. The total utility $\mathbb{U}_c$ derived from running a sync task at sync option c is equally divided across all workers regardless of the size of the cluster they are a part of.

$$\mathbb{S}_c = \frac{\mathbb{U}_c}{|\mathbb{N}_s|}$$

The total utility derived from running a sync task is a function of how soon the synchronization occurs and successful completion of the synchronization task. An earlier sync option will yield a higher utility compared to other later sync options. Therefore, players have a higher incentive to cooperate better and sooner to maximize their payoff. We consider a two–cluster game. The game has a fast and a slow cluster. Workers are grouped into clusters based on their execution progress. There could be outliers (workers that do not fall into any of the two major clusters). Outliers can be part of another cluster, but they are not considered in the game. They proceed with the execution schedule and plan created by the outcome of the game. We keep track of the clusters only at synchronization points.

6.4.2 Execution Time Distributions

The execution progress of workers is tracked by the controller through checkpoints in the application. The controller creates distributions for the two clusters for the expected finish time of the task before the sync point from their previous run of the application tasks. Each cluster is represented by a mixture distribution of two Gaussian distributions. 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 $\mathbb{D}^{lo}_{late} = G(\mu_{lo\_la}, \sigma^2_{lo\_la})$.

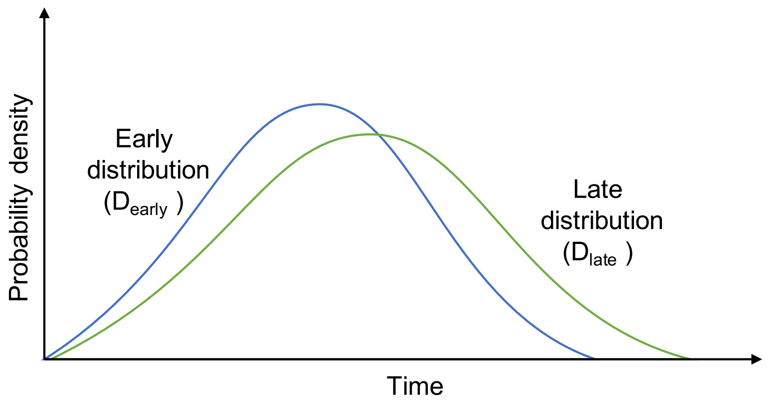


Fig. 6.5 Sample mixture model for a cluster with early and late distributions.

6.4.3 Late Notification Protocol

The main goal of clustering the workers in our system is to reduce the message overhead required in reaching synchronization. Thus, workers in a cluster are expected to remain synchronized and make the same synchronization decisions. The late strategy that involves sending late notifications to inform the other cluster of lateness requires sending messages. To reduce the number of messages required in classifying a cluster as late, we develop a late notification protocol where 3 messages are expected to be received from workers in a particular cluster for the cluster to be regarded as late. We assume that workers can detect when they will be late when they get to 50% of the current task execution based on the predicted finish time of the cluster for that task.

The first worker in a cluster that detects it will be late sends out a late notification to workers in the other cluster. After the first notification, the probability of a having a second late notification sent is $2/N$ for each late worker, thus, if half of the workers are late, we expect the second late notification to be sent. The probability of having a third late notification sent is $1/N$ for each late worker. Thus, if all the workers in a cluster are late, we expect a total of 3 late notifications to be sent.

A cluster could get stuck at a sync option if late notification broadcast messages are lost due to network partitioning. Network partitioning could cause temporary or complete isolation. A worker that is temporarily isolated is only disconnected for one or a few iterations, while a completely isolated worker is totally disconnected from other workers in the system. To ensure safety in case of temporary isolation, we embed previous late notifications in new late notifications. Thus, the second late notification will contain the first late notification. If the first late notification gets lost due to network partitioning, workers in the other cluster will get both late notifications embedded in the second notification. A worker that is completely isolated will synchronize personally until it gets connected back.

6.4.4 Extensive Form of Synchronization Game

The controller in our system needs to create a static schedule with different sync options and broadcast the schedule to worker nodes. To fix these sync options, the controller uses a game between worker nodes abstracted at the cluster–level. The controller creates the schedule based on the outcome of the game that is expected to yield the maximum payoff. The protocols for choosing which points to synchronize at by the workers is specified by the game. The game is played at the cluster level, although synchronization is done by the workers. If a cluster makes a decision, we expect that all workers within the cluster are going to make the same decision as the cluster. Unexpected behavior during runtime is tackled using late notifications.

The first pass of the extensive form of the game with two players (clusters) is shown in Fig. 6.6. The strategy profiles for both clusters include {*sync, no–sync–late, no–sync–late–notification*}. In the first pass, there are four possibilities as shown in Fig. 6.6; (i) both clusters synchronize at the first sync option (green node), (ii) sync aborted because one of the clusters is stuck at first sync option (red node), (iii) sync option skipped because both clusters are late (purple node), and (iv) sync option skipped because one cluster got late notification.

The second pass is similar to the first pass, and it originates from the nonterminal nodes in the first pass as shown in Fig. 6.7. The choices are the same as in the first pass. However, the payoff is cumulative, that is, the payoff derived after the second pass is the addition of that derived in the first pass and second pass. The second pass likewise has 3 non–terminal nodes as in the first pass. The third and final pass starts with the nonterminal nodes in the second pass. All the exit nodes in the third pass are terminal nodes. The payoff after the third pass is the sum of all payoffs at all the passes.

We make the following assumptions.

Assumption 1: *The utility $\mathbb{S}_{s1}$ derived by a worker from synchronizing at the first option is much greater than the utility $\mathbb{S}_{s2}$ derived from synchronizing at*

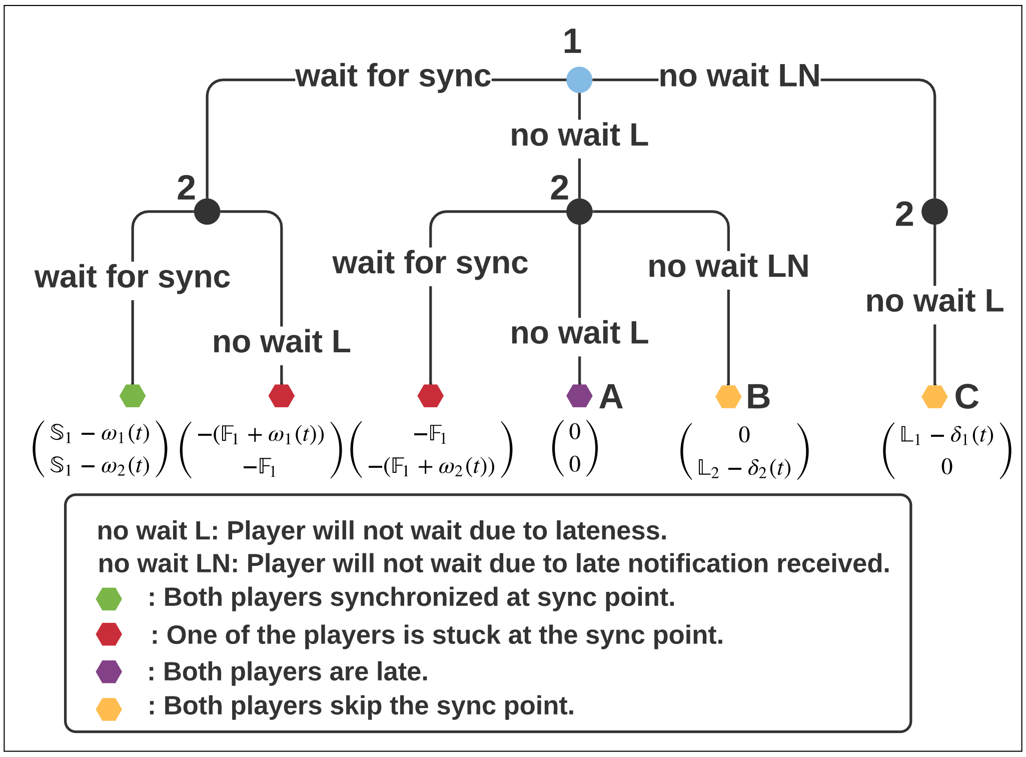

1
wait for sync
no wait LN
no wait L
2
2
2
wait for sync
no wait L
wait for sync
no wait L
no wait LN
no wait L
A
B
C
$\begin{pmatrix} \mathbb{S}_1 - \omega_1(t) \\ \mathbb{S}_1 - \omega_2(t) \end{pmatrix}$
$\begin{pmatrix} -(\mathbb{F}_1 + \omega_1(t)) \\ -\mathbb{F}_1 \end{pmatrix}$
$\begin{pmatrix} -\mathbb{F}_1 \\ -(\mathbb{F}_1 + \omega_2(t)) \end{pmatrix}$
$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$
$\begin{pmatrix} 0 \\ \mathbb{L}_2 - \delta_2(t) \end{pmatrix}$
$\begin{pmatrix} \mathbb{L}_1 - \delta_1(t) \\ 0 \end{pmatrix}$
no wait L: Player will not wait due to lateness.
no wait LN: Player will not wait due to late notification received.
: Both players synchronized at sync point.
: One of the players is stuck at the sync point.
: Both players are late.
: Both players skip the sync point.

Fig. 6.6 First pass of the extensive form of the two–player synchronization game. Nodes A, B and C are non–terminal nodes where the game proceeds to the second pass. The green node is a terminal node with synchronization successful. The red nodes are terminal nodes where synchronization failed. The vectors represent the corresponding payoffs for player 1 and player 2.

the second option and the third sync option $\mathbb{S}_3$*, regardless of any added utility* $\mathbb{L}$ *derived from running a local task.*

$$\begin{aligned} &\mathbb{S}_{s1} > \mathbb{S}_{s2} > \mathbb{S}_{s3} \\ &\mathbb{S}_{s1} > \mathbb{S}_{s2} + \mathbb{L} > \mathbb{S}_{s3} + \mathbb{L} \end{aligned} \tag{6.1}$$

Thus, the earlier the synchronization is attempted, the more the payoff that is gotten. The payoff from synchronizing at a later time can never be more than the payoff of synchronizing at an earlier time.

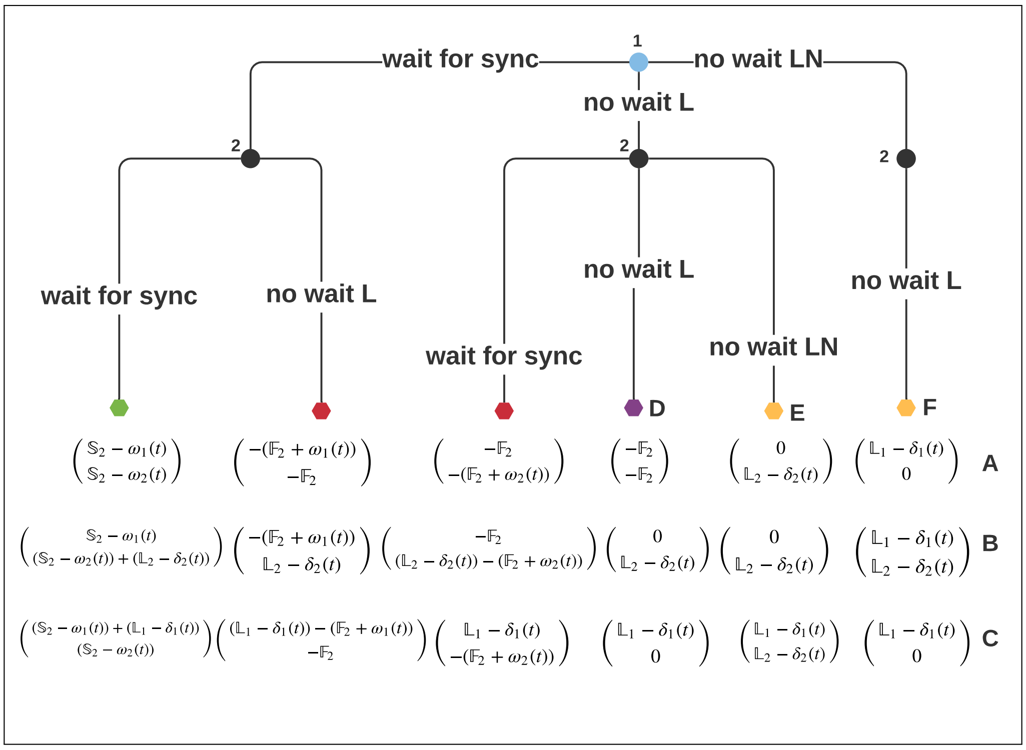

1
wait for sync
no wait LN
no wait L
2
2
2
wait for sync
no wait L
wait for sync
no wait L
no wait LN
no wait L
D
E
F
A
B
C
$\begin{pmatrix}\mathbb{S}_2-\omega_1(t)\\ \mathbb{S}_2-\omega_2(t)\end{pmatrix}$ $\begin{pmatrix}-(\mathbb{F}_2+\omega_1(t))\\ -\mathbb{F}_2\end{pmatrix}$ $\begin{pmatrix}-\mathbb{F}_2\\ -(\mathbb{F}_2+\omega_2(t))\end{pmatrix}$ $\begin{pmatrix}-\mathbb{F}_2\\ -\mathbb{F}_2\end{pmatrix}$ $\begin{pmatrix}0\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ 0\end{pmatrix}$
$\begin{pmatrix}\mathbb{S}_2-\omega_1(t)\\ (\mathbb{S}_2-\omega_2(t))+(\mathbb{L}_2-\delta_2(t))\end{pmatrix}$ $\begin{pmatrix}-(\mathbb{F}_2+\omega_1(t))\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$ $\begin{pmatrix}-\mathbb{F}_2\\ (\mathbb{L}_2-\delta_2(t))-(\mathbb{F}_2+\omega_2(t))\end{pmatrix}$ $\begin{pmatrix}0\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$ $\begin{pmatrix}0\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$
$\begin{pmatrix}(\mathbb{S}_2-\omega_1(t))+(\mathbb{L}_1-\delta_1(t))\\ (\mathbb{S}_2-\omega_2(t))\end{pmatrix}$ $\begin{pmatrix}(\mathbb{L}_1-\delta_1(t))-(\mathbb{F}_2+\omega_1(t))\\ -\mathbb{F}_2\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ -(\mathbb{F}_2+\omega_2(t))\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ 0\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ \mathbb{L}_2-\delta_2(t)\end{pmatrix}$ $\begin{pmatrix}\mathbb{L}_1-\delta_1(t)\\ 0\end{pmatrix}$

Fig. 6.7 Second pass of the extensive form of the two–player synchronization game. The payoff for each of the node options A, B and C from pass 1 are shown.

Assumption 2: *The cost of aborting sync increases downwards from the first sync option to the third sync option.*

$$\mathbb{F}_{s1} < \mathbb{F}_{s2} < \mathbb{F}_{s3} \tag{6.2}$$

What this means is that it is better to abort synchronization at the first sync option and move on with the execution plan, rather than wait until other options to abort sync.

Assumption 3: *The utility $\mathbb{S}_c$ derived from synchronizing at a sync option c is greater than the cost $\mathbb{F}_c$ of aborting sync at a sync option c which is in turn greater than the utility derived from running a local task l.*

$$\mathbb{S}_c > \mathbb{F}_c > \mathbb{L} \tag{6.3}$$

Assumption 4: *The utility* $\mathbb{L}$ *derived from running a local task is always greater than the cost of waiting to get a late notification* $\delta(t)$

$$\mathbb{L} > \delta(t) \quad \forall \text{ workers} \tag{6.4}$$

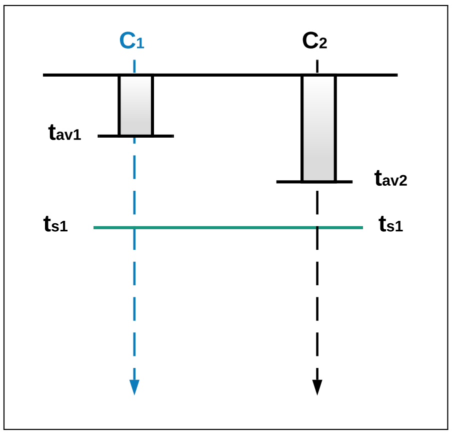
Assumption 5: *The strategy (sync, sync) at the first sync option is Pareto optimal since there is no other strategy set that gives a higher payoff.*

There are other game strategies that give an optimal solution depending on the runtime operation of the clusters. The strategies (*sync, sync*) at second and third sync options are also optimal solutions to the game depending on what happens at runtime. However, the Pareto optimal solution is the one where both clusters synchronize at the first sync option as evident from Assumption 1.

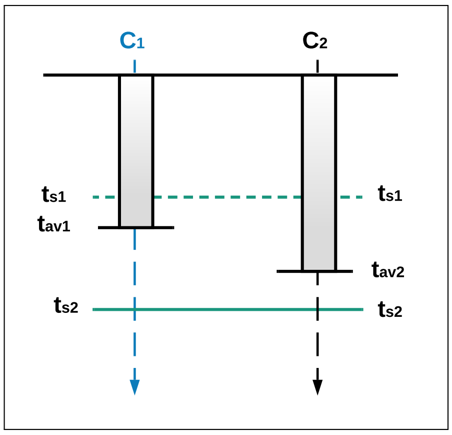
6.4.5 Synchronization Scenarios

Fig. 6.8 shows different synchronization option scenarios between two clusters. Cluster C_1 is the fast cluster while cluster C_2 is the slow cluster. t_{av1} and t_{av2} are the expected available times of cluster C_1 and C_2, both of which are gotten from the distributions $\mathbb{D}^1_{norm}$ and $\mathbb{D}^2_{norm}$ respectively. The scenarios and the payoff breakdown are explained as follows.

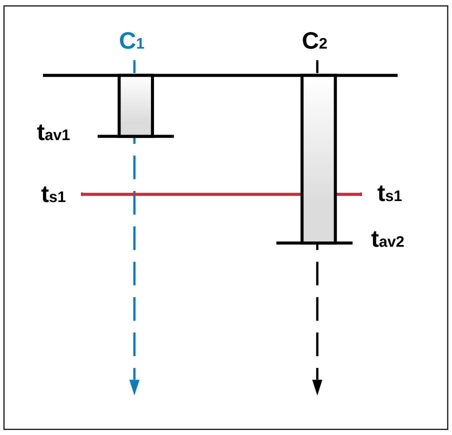
1. Both clusters waiting for synchronization (Fig. 6.8a): The synchronization point is fixed such that the desired quorum is met and the cluster property is satisfied. If both clusters choose to wait for synchronization and they finally synchronize, they get a payoff of $\mathbb{S} - \omega(t)$. The payoff for successfully synchronizing is the utility gained from executing the sync task at the first sync option minus the waiting cost incurred by each cluster respectively.



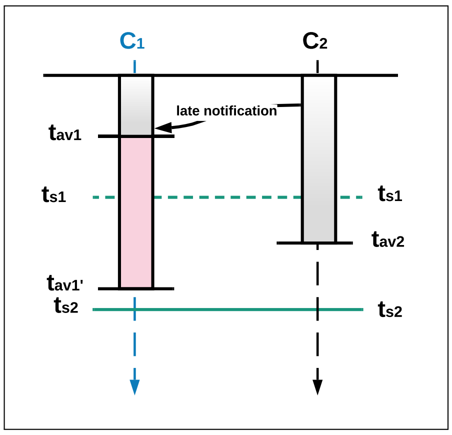
(a) C_1 and C_2 waiting to sync at sync option 1.



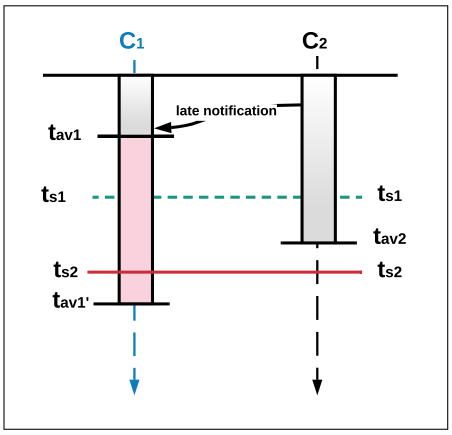
(b) C_1 and C_2 are late but did not notify each other.



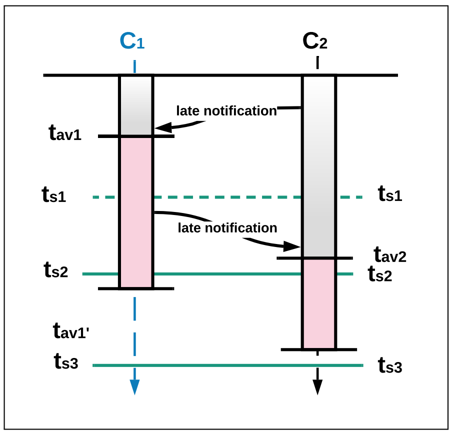
(c) Sync is aborted because $C1$ gets stuck at sync option 1.



(d) C_1 and C_2 skip sync option 1 due to late ack from C_2 to C_1.



(e) Sync is aborted because C_2 gets stuck at second sync option.



(f) C_1 and C_2 proceed to sync option 3 after running local tasks.

Fig. 6.8 Example of synchronization option scenarios.

2. Both clusters late without notification (Fig. 6.8b): When both clusters are late for synchronization and did not send out any notifications, they both proceed to the next synchronization point and get a payoff of 0 each for the current synchronization option.

3. One cluster waiting and the other late (Fig. 6.8c and 6.8e): If one of the clusters is waiting for synchronization and the other cluster is late to the sync point without any notification, the first cluster will get stuck. This is because the fast cluster will proceed to run the sync task without quorum and fail. In this case, the sync operation is aborted, and the payoff is $-(\mathbb{F} + \omega)$ for the fast cluster, while the late cluster gets a payoff of $-\mathbb{F}$. In the case of Fig. 6.8e, cluster C_1 gets an additional payoff of $\mathbb{L} - \delta$ for running a local task.

4. One cluster sends late notification (Fig. 6.8d and 6.8f): If the first cluster sends a late notification to the second cluster, the second cluster does not wait again. Both clusters skip the sync point. If the first cluster sent the notification before the second cluster becomes available, it incurs a cost of 0 or δ otherwise if it waits for some time before getting the notification. The first cluster (cluster that sent the late notification) gets a payoff of 0 while the second cluster gets a payoff of $\mathbb{L} - \delta$ which is the utility derived from running a local task minus the cost incurred in waiting before the late notification was received.

6.5 Analysis of the Synchronization Game

6.5.1 Optimal Synchronization Options

Let $(\mathbb{V}_1, \mathbb{V}_2)$ be the payoff vector for cluster 1 (fast cluster) and cluster 2 (slow cluster) respectively and the cumulative payoff $\mathbb{P} = \mathbb{V}_1 + \mathbb{V}_2$. The optimum solution S^* to the game is defined as:

$$S^* = \arg\min_{t_s^c} \max_{\mathbb{P}} \sum_{i=1}^{2} \mathbb{V}_i \tag{6.5}$$

The highest payoff is gotten when both clusters decide to wait for synchronization at the first sync option. The combination of both strategies by both clusters forms a Nash equilibrium since neither cluster can get a higher payoff by switching to a different strategy as evident in Assumption 1. Thus, if one cluster chooses to wait for synchronization, it knows that the other cluster has no incentive to not wait for synchronization. To determine the optimal number of sync options, we look at different scenarios in the game. We have two clusters arriving at the sync point: a fast and slow cluster. The first choice will be to attempt synchronization at the point where we expect to meet the desired quorum. According to Assumption 1 and the payoff got from synchronizing $(\mathbb{S}_c - \omega(t))$, a higher payoff is gotten if synchronization is attempted as soon as we have quorum such that $\omega(t_w)$ will be close to 0. Thus, the first sync option looks to minimize $(\omega(t_{w1}) + \omega(t_{w2}))$.

If the slower cluster gets late to the first sync option, there is a need for a second sync option. The second option must be fixed such that it maximizes the cumulative payoff $(\mathbb{P})$. The earlier cluster will look to get a higher payoff by running a local task. When a cluster is late, it has no incentive not to inform the other cluster by sending a late notification. The optimal option is to fix the second sync option such that if $(\mathbb{L}_1(t_l) > \omega(t_{w2})$, then the fast cluster executes a local task before attempting synchronization again. Else if $(\mathbb{L}_1(t_l) < \omega(t_{w2})$, synchronization is attempted immediately after the late cluster becomes available. This guarantees that the cumulative payoff for both clusters is maximized.

In a case where the fast cluster executing the local task overshoots the second sync option, it has no incentive not to inform the second cluster. The second cluster can in turn run a local task to improve the cumulative payoff $\mathbb{P}$ if $(\mathbb{L}_2(t_l) > \omega(t_{w2}))$ and the third sync option can be fixed after the expected finish time of the local task on the second cluster. Otherwise, if running the local task does not improve the cumulative payoff, synchronization can be attempted immediately after. Beyond this point, there is no guarantee that an optimal solution can be found that guarantees a higher cumulative payoff since both clusters would have executed local tasks and there is no other way to improve the cumulative payoff pending synchronization. Thus, it is not an optimal strategy to keep waiting for synchronization beyond this point.

In the case where a cluster is unable to make the first synchronization option, the next optimal solution is to attempt to run any local task if available and go to the second synchronization option. The explanation above still stands since no cluster has any incentive to defect from waiting if the other cluster waits. According to Assumption 2, a cluster will prefer to wait for synchronization if it expects its local task to overshoot the sync option. To get S^*, it is imperative to fix the sync options such that the number of expected workers from both clusters is greater than or equal to the desired quorum. Synchronization must be attempted at the earliest possible options.

6.5.2 Fixing the Synchronization Options

Given that we have the mixture distributions $\mathbb{D}^1$ and $\mathbb{D}^2$ for the execution times of the fast and slow clusters C_1 and C_2 respectively, and likewise the distribution of the expected execution time of local tasks on both clusters, we can fix the three synchronization options. The expected available times of the clusters can be chosen from the mixture distributions by choosing the desired percentile $p(x)$. The percentile values are used because we expect workers in a cluster to make the same decisions.

The sum Z of two normally distributed independent random variables X and Y is also normally distributed.

$$\begin{aligned} X &= G(\mu_x, \sigma_x^2) \\ Y &= G(\mu_y, \sigma_y^2) \\ Z &= G(\mu_x + \mu_y, \sigma_x^2 + \sigma_y^2) \end{aligned} \tag{6.6}$$

First Synchronization Option: For the first sync option, we are interested in getting the time value t_s^1 such that the desired percentile p on both clusters is available, and the desired quorum is met. The percentile is sampled from the early execution distribution $\mathbb{D}_{early}^1$ and $\mathbb{D}_{early}^2$ for both clusters. We use the early execution distribution to fix the first synchronization option as early as possible. Let X_1 and X_2 be the time values that correspond to the chosen percentiles on both distributions for both clusters. The time t_s^1 for the first synchronization option can be fixed by solving the following equation:

$$\begin{aligned} \text{minimize} \quad & t_s^1 \\ \text{subject to} \quad & p(x)\,|C_1| + p(x)\,|C_2| \geq \alpha N, \\ & X_1 = p(x)\{\mathbb{D}_{early}^1\}, \\ & X_2 = p(x)\{\mathbb{D}_{early}^2\}, \\ & t_s^1 = max(X_1, X_2) \end{aligned} \tag{6.7}$$

Second Synchronization Option: The second sync option is fixed such that the faster cluster, say C_1 either waits for the slower cluster, say C_2 (which is late) or executes a local task (if available) if it increases the cumulative payoff. The percentile for the expected available time t_{av}^2 is drawn from the distribution $\mathbb{D}_{late}^2$.

The second sync option t_s^2 is gotten by solving the equation:

$$
\begin{aligned}
&\text{minimize} \quad t_s^2 \\
&\text{subject to} \\
&p(x_1)\,|C_1| + p(x_2)\,|C_2| \geq \alpha N, \\
&X_1^{'} = X_1 && \text{if} \quad \mathbb{L}_1(t_l^1) < \omega(t_{w2}), \\
&X_1^{'} = X_1 + p(x_1)\{\mathbb{D}_{early}^1 + \mathbb{D}_{early}^{lo1}\} && \text{if} \quad \mathbb{L}_1(t_l^1) \geq \omega(t_{w2}), \\
&X_2^{'} = p(x_2)\{\mathbb{D}_{late}^2\}, \\
&t_s^2 = max(X_1^{'}, X_2^{'})
\end{aligned}
\tag{6.8}
$$

$X_1^{'}$ is the time point where we expect a certain percentile of the workers in the faster cluster to be available to synchronize. If cluster C_1 executes a local task, $X_1^{'}$ is gotten by getting the desired percentile from the sum of the distributions $\{\mathbb{D}_{early}^1$ and $\mathbb{D}_{early}^{lo1}\}$ as explained in Equation 6.6.

Third Synchronization Option: The last synchronization option is fixed to cater for the situation where the cluster (C_1) running the local task is late to the second sync option and sends a late notification to cluster C_2. The other cluster C_2 can decide to wait or run a local task. This is dependent on which of the choices increases the cumulative payoff. The new expected available time of C_1 is drawn from the distribution $\mathbb{D}_{late}^{lo}$. t_s^3 is fixed by solving:

$$
\begin{aligned}
&\text{minimize} \quad t_s^3 \\
&\text{subject to} \\
&p(x_1^{'})\,|C_1| + p(x_2^{'})\,|C_2| \geq \alpha N, \\
&X_1^{''} = X_1 + p(x_1^{'})\{\mathbb{D}_{early}^1 + \mathbb{D}_{late}^{lo1}\}, \\
&X_2^{''} = X_2^{'} && \text{if} \quad \mathbb{L}_2(t_l^2) < \omega(t_{w1}), \\
&X_2^{''} = X_2^{'} + p(x_2^{'})\{\mathbb{D}_{late}^2 + \mathbb{D}_{early}^{lo2}\} && \text{if} \quad \mathbb{L}_2(t_l^2) \geq \omega(t_{w1}), \\
&t_s^3 = max(X_1^{''}, X_2^{''})
\end{aligned}
\tag{6.9}
$$

X_1'' is the time point where we expect a certain percentile of the workers in the faster cluster to have finished executing the local task. We switch to the late local task execution distribution $\mathbb{D}^{lo1}_{late}$ since the cluster is late. X_1'' is drawn from the sum of the distributions $\{\mathbb{D}^{1}_{early}$ and $\mathbb{D}^{lo1}_{late}\}$. If cluster C_2 executes a local task, X_2'' is drawn from the sum of the distributions $\{\mathbb{D}^{1}_{early}$ and $\mathbb{D}^{lo1}_{early}\}$ as explained in Equation 6.6.

6.5.3 Fast Synchronization Algorithm

The synchronization algorithm shows the processes and decisions made by the controller and workers as shown in Algorithm 5 and Fig. 6.9. The algorithm outputs different runtime actions that can be taken by the clusters depending on the runtime configurations. The available times of fast and slow clusters C_1 and C_2 are t^1_{av} and t^2_{av} respectively. The runtime synchronization flow is shown in Fig. 6.9. The controller computes the synchronization schedule and fixes the three synchronization options for each synchronization point by solving Equations 6.7 – 6.9. Asynchronous tasks are run as soon as the workers become available.

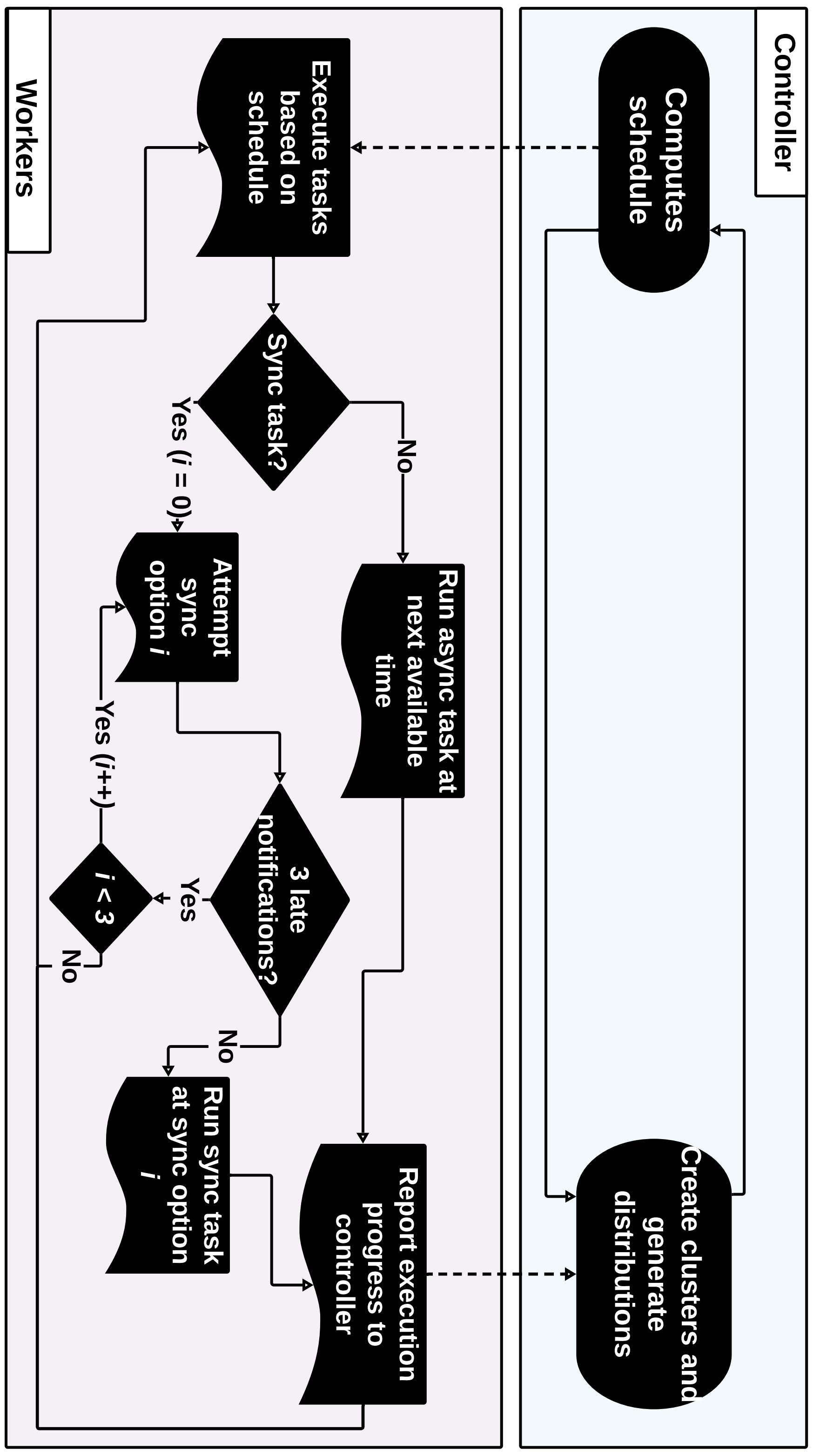

Controller
Computes schedule
Create clusters and generate distributions
Workers
Execute tasks based on schedule
Sync task?
No
Yes (i = 0)
Run async task at next available time
Attempt sync option i
3 late notifications?
Yes
No
i < 3
Yes (i++)
No
Run sync task at sync option i
Report execution progress to controller

Fig. 6.9 Controller and workers actions during runtime in fast synchronization scheme.

Algorithm 5: Synchronization algorithm

```
Controller:
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
forall workers do:
   First sync option:
  if $(t_{av}^1 \leq X_1)$ and $(t_{av}^2 \leq X_2)$:
      $execute(T_{sync}, t_s^1)$;
      end;
  elif $(t_{av}^1 \leq X_1)$ and $(t_{av}^2 > X_2)$ and send($C_2$, late_notify):
      if $t_l^1 \leq t_s^1 - t_{av}^1$:
          $execute(T_{local}, t_{av}^1)$;
      proceed to line 18;
  elif $(t_{av}^1 > X_1)$ and $(t_{av}^2 > X_2)$:
      proceed to line 18;
  elif $(t_{av}^1 \leq X_1)$ and $(t_{av}^2 > X_2)$ and no_late_notify:
      abort(sync);
      end;
   Second sync option:
  if $(t_{av'}^1 \leq X_1^{'})$ and $(t_{av'}^2 \leq X_2^{'})$:
      $execute(T_{sync}, t_s^2)$;
      end;
  elif $(t_{av'}^1 > X_1^{'})$ and $(t_{av'}^2 \leq X_2^{'})$ and  send($C_1$,late_notify):
      if $t_l^2 \leq t_s^2 - t_{av}^2$:
          $execute(T_{local}, t_{av}^2)$;
      proceed to line;
  elif $(t_{av}^1 \leq X_1)$ and $(t_{av}^2 > X_2)$ and no_late_notify:
      abort(sync);
      end;
   Third sync option:
  if $(t_{av''}^1 \leq X_1^{''})$ and $(t_{av''}^2 \leq X_2^{''})$:
      $execute(T_{sync}, t_s^3)$;
      end;
  elif $(t_{av''}^1 \leq X_1^{''})$ and $(t_{av''}^2 > X_2^{''})$:
      abort(sync);
      end;
```

The function $execute(T_{sync}, t_s^n)$ means that the sync task T_{sync} should be executed at sync option n and start executing at time t_s^n. For the first sync option, if both clusters become available before the predicted available times (Line 5), the sync task is executed by the workers at the first sync option (Line 6). However, if the slower cluster is late and sends a late notification to the faster cluster, the faster cluster can run a local task before proceeding to the second option if the local task can fit in the space (Lines 8–10). If both clusters are late to the first sync option, they both proceed to the second sync point. Synchronization is aborted whenever a late cluster does not send a late notification (Lines 14–15 and 25–26).

At the second sync option, the same operations apply as in the first sync option. However, if the faster cluster is late in executing the local task, the slower cluster can likewise decide to execute a local task before proceeding to the third sync option if it can fit (Lines 22–23). The sync task is executed at the third sync option only if both clusters are available at the predicted available times (Lines 29–30). Else, synchronization is aborted, and that particular synchronization point is considered to have failed.

6.6 Simulations and Results

6.6.1 Simulation Configuration

We use a task graph (DAG) with a mixture of synchronous, asynchronous, and local tasks. The task graph is similar to those used in Bulk Synchronous Parallel (BSP), Stale Synchronous Parallel (SSP) [141] and Dynamic Stale Synchronous Parallel (DSSP) [144] approaches for synchronizing parameter updates in distributed machine learning and neural networks. The models usually have the following four steps. (i) Compute gradients using local weights, (ii) push the gradients to the parameter server to compute the global weights, (iii) pull new computed global weights from the parameter server, and (iv) update local weights using global weights.

These models assume that workers are only involved in the model training and updating process. However, in our work, we consider the case where workers are not only involved in model training, but also in the data capture process and usage of the model's output. We introduce local tasks to show activities where the workers need to do some personal computations for effective functioning of the running application. Local tasks are triggered at runtime based on the application's needs and configurations.

The execution time of a single task on workers is based on a mixture distribution. One for the fast execution and the other for slow execution. We use traces from the clustering experiments as the dataset for the execution time of tasks in our simulations. The times are split into two to depict short ($\mu = 25ms$) and long tasks ($\mu = 80ms$).

The parameters in the simulations are as follows. (i) *Synchronization degree*: ratio of the total machines required to pass quorum, (ii) *worker size*: The maximum number of workers present in the system at any point in time, (iii) *simulation rounds*: The number of times the task graph is continuously run, and (iv) *clustering frequency*: This is the rate at which re–clustering is done by the controller.

6.6.2 Default Parameter Values and Measurements

The following parameters are fixed in the simulations unless otherwise stated. The number of independent runs of each simulation is 100, while each task graph is continuously run in each simulation for 200 times (rounds). Worker–worker message cost is set at ($\mu = 2ms, \sigma = 0.3$) and worker–controller message cost is set at ($\mu = 25ms, \sigma = 2$). The synchronization degree is fixed at 0.7. Local tasks execution times vary from $5ms$ to $10ms$. Clustering cost is set at $20ms$. The same task graph is run on all workers.

The following parameters are measured in the simulations. (i) *Runtime/sync point*: the time taken for a single iteration of a task graph divided by the number of sync points, (ii) *sync success/failure*: the total number of times synchronization was successful or failed at different synchronization options, and (iii) *participation*: the ratio of the total devices that synchronized at a sync point.

6.6.3 Simulation Results and Discussions

6.6.3.1 Single vs Flexible Clustering

We measure the impact of re–clustering on the runtime per sync point, quorum participation, sync success at different options, and sync failure. We consider single (fixed) clustering where workers are clustered only once in the system, thus, workers belong to the same cluster all through the execution. We likewise consider the case where clustering is done after a few iterations (set to 5).

Fig. 6.10 and 6.11 shows the runtime per sync point for single and iterative clustering for varying number of workers, respectively. The runtime per sync point for single clustering is smaller compared to that for iterative clustering. This is

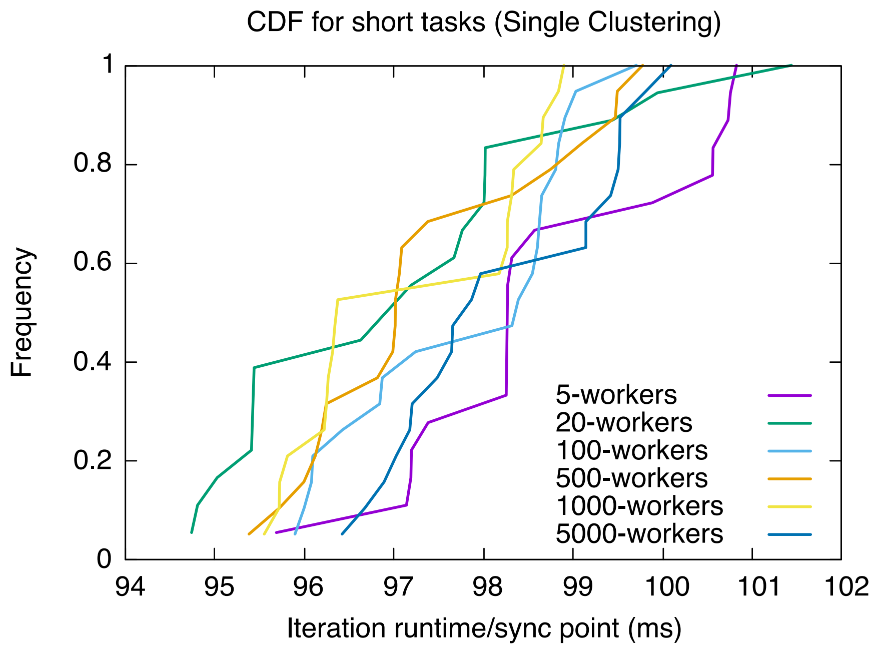


Fig. 6.10 Runtime for short tasks (fixed clustering).

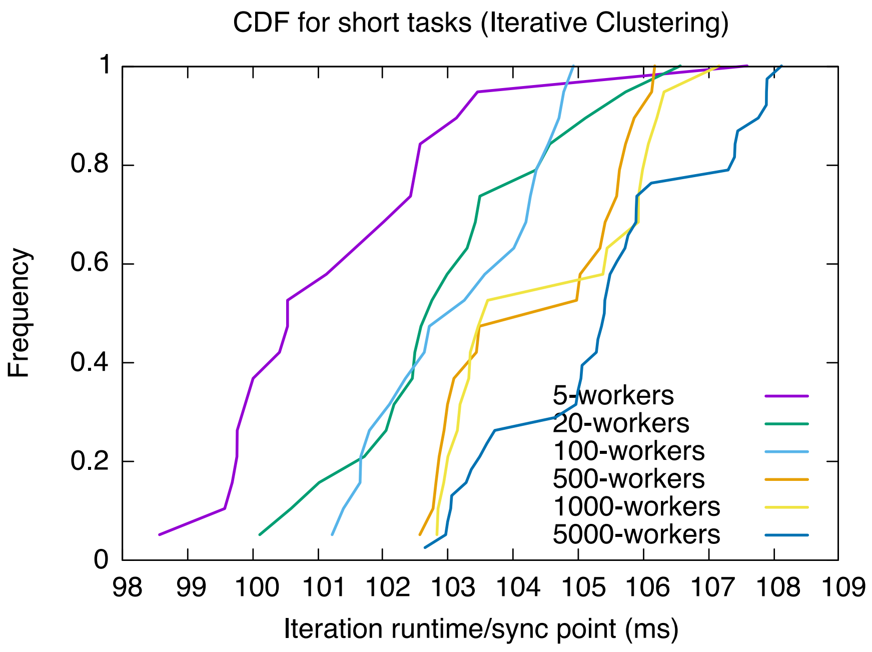


Fig. 6.11 Runtime for short tasks (flexible iterative clustering).

due to extra cost incurred in re–clustering. However, iterative clustering has more sync successes at the first sync option compared to single clustering as well as less failed synchronizations as shown in Fig. 6.12 and 6.13. This is because the schedule generated by the cluster using the execution progress distributions of the clusters is updated as re–clustering is done, thus, improving the accuracy of the schedule. Single clustering has more sync participation than iterative clustering for varying number of workers as seen in Fig. 6.14 and 6.15. This is because more workers are expected to be available at the second and third sync options, as there are more sync successes at those options for single clustering.

6.6.3.2 Worker Heterogeneity

To measure the effect of heterogeneity of workers on our algorithm, we vary the execution time deviation of tasks across workers and explore the impact it has on runtime per sync point and quorum participation. Increasing the standard deviation of a task among several workers increases the possibility of having

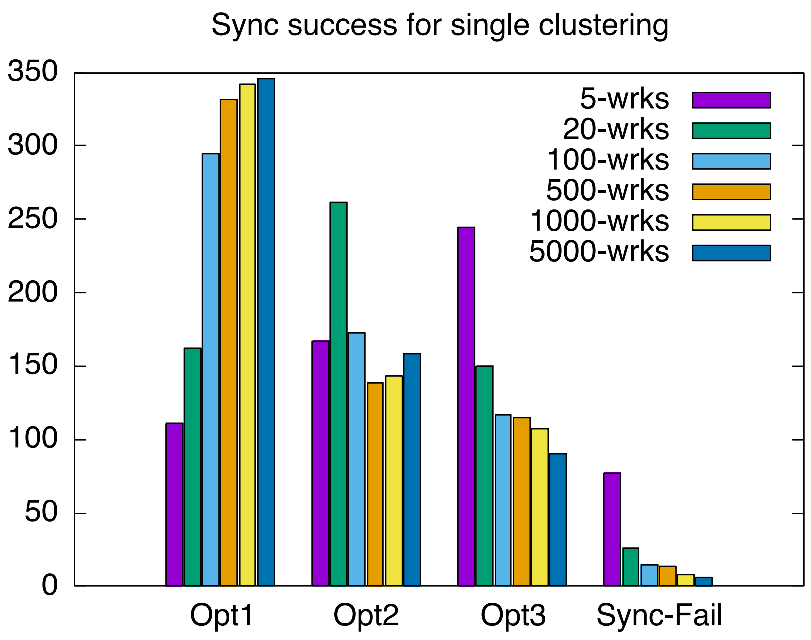


Fig. 6.12 Number of successful and failed synchronizations at different sync options (fixed clustering).

stragglers. The task execution time deviation is varied from $1.5ms$ to $6ms$ for 100 workers and short tasks as shown in Fig. 6.16. The runtime per sync point increases as the variance of task execution time across workers is increased from $1.5ms$ to $6ms$. The average sync participation for all execution time variances is about 0.75 with execution time variance of 1.5 having a slightly higher average. As the execution time variance increases, we have higher sync participation deviation as shown in Fig. 6.17.

6.6.3.3 Comparison with Other Synchronization Protocols

We evaluate the performance of our algorithm (*Fast_Sync*) by comparing it with the BSP, SSP and DSSP synchronization protocols frequently used in training distributed machine learning models. For BSP, we fix the synchronization barrier at the time point where the last worker finishes executing the task before the sync point. Thus, fast workers need to wait for slow workers at the synchronization

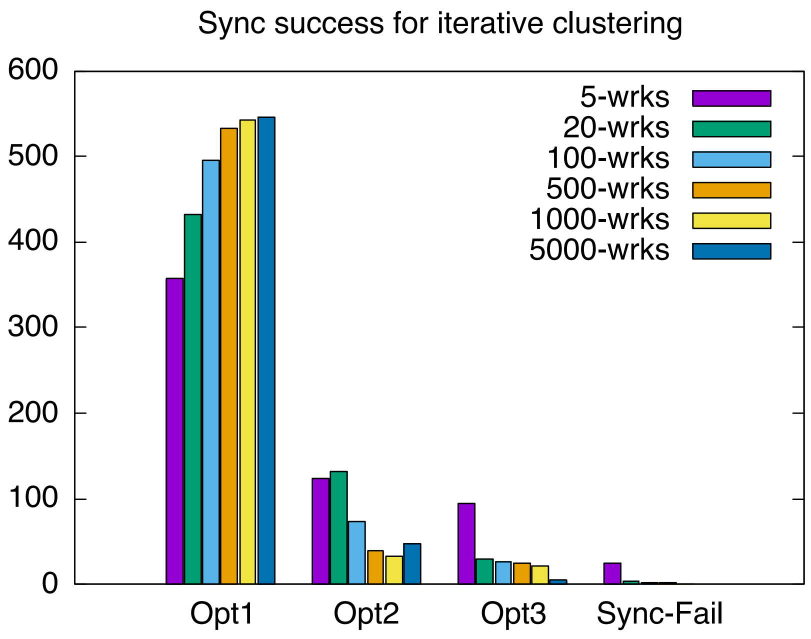


Fig. 6.13 Number of successful and failed synchronizations at different sync options (flexible clustering).

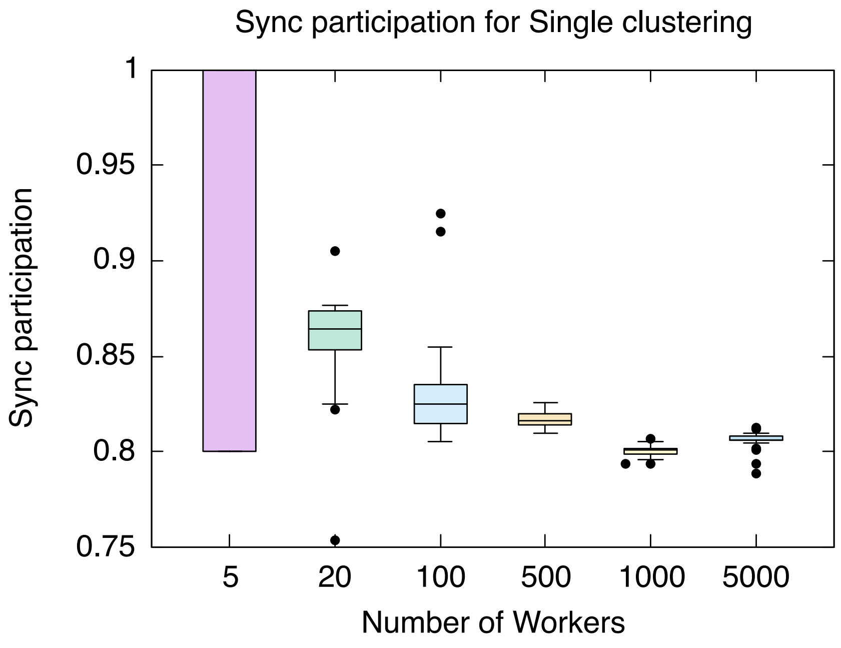


Fig. 6.14 Ratio of workers that synchronized for short tasks (fixed clustering).

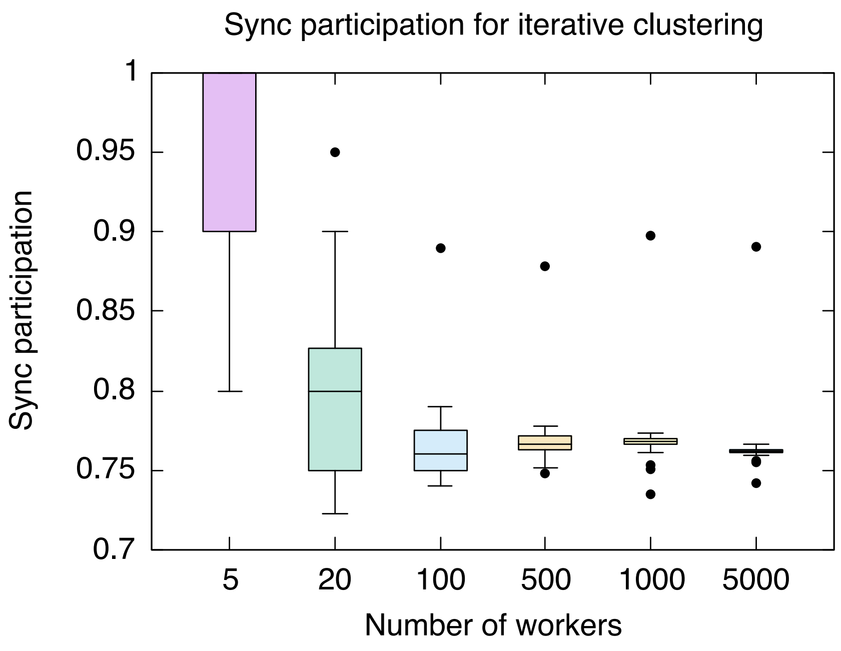


Fig. 6.15 Ratio of synchronized workers for short tasks (flexible iterative clustering).

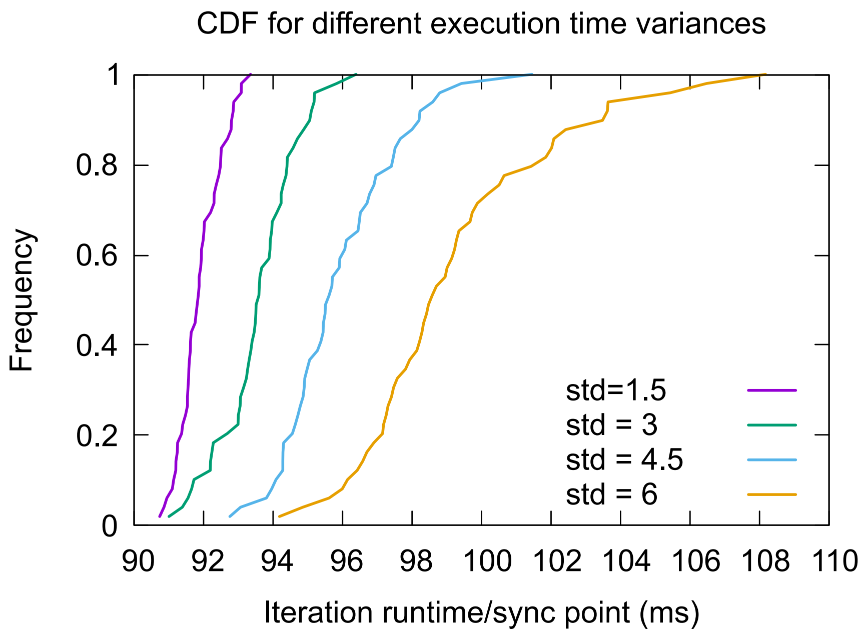


Fig. 6.16 Runtime for different task execution time variances.

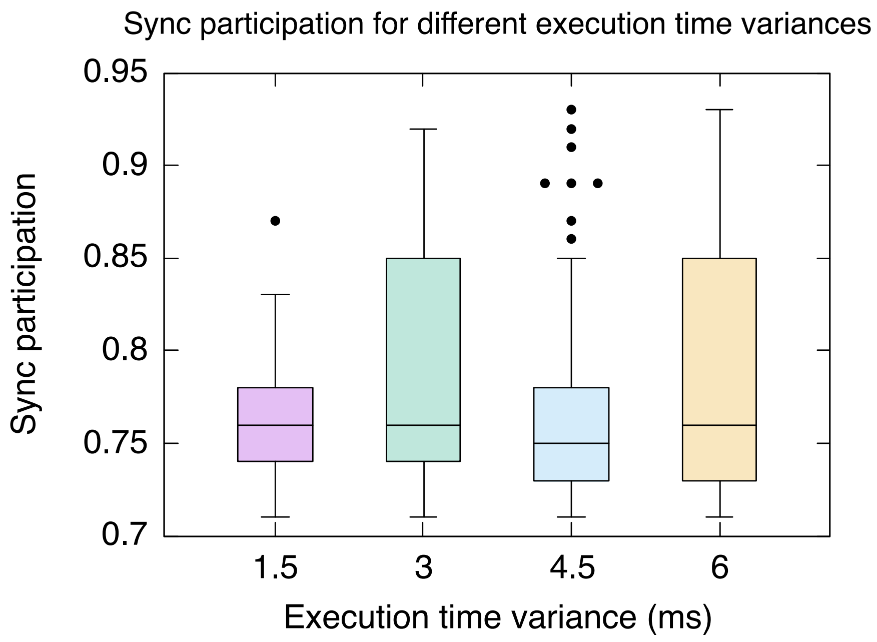


Fig. 6.17 Ratio of synchronized workers for different task execution time variances.

barrier. For SSP, we set the staleness threshold s to 3 and, 5 with each threshold unit being equivalent to $5ms$. For DSSP, we set s to 3 and the $r_{max} = 7$; this is the maximum allowable execution distance between the fastest and lowest worker beyond s. We consider a task graph with two asynchronous tasks, a single sync task, and two local tasks. For each iteration, we split the execution time into computation, communication, and clustering times.

To measure the effect of worker heterogeneity on the algorithms, we measure the execution runtime for our algorithm and the other synchronization models with worker execution time variance of $2ms$ and $10ms$ across 20 workers as shown in Fig. 6.18 and 6.19 respectively. Fig. 6.18 shows the average runtime for our algorithm ($Fast\_Sync$), BSP, SSP and DSSP with worker execution time variance of $2ms$ while Fig. 6.19 shows for varying worker execution variance of $10ms$. Our algorithm performs best in both cases, both in terms of computation time and communication cost. DSSP performs almost as well as our algorithm, with BSP performing worst in both cases. Our algorithm outperforms DSSP because DSSP

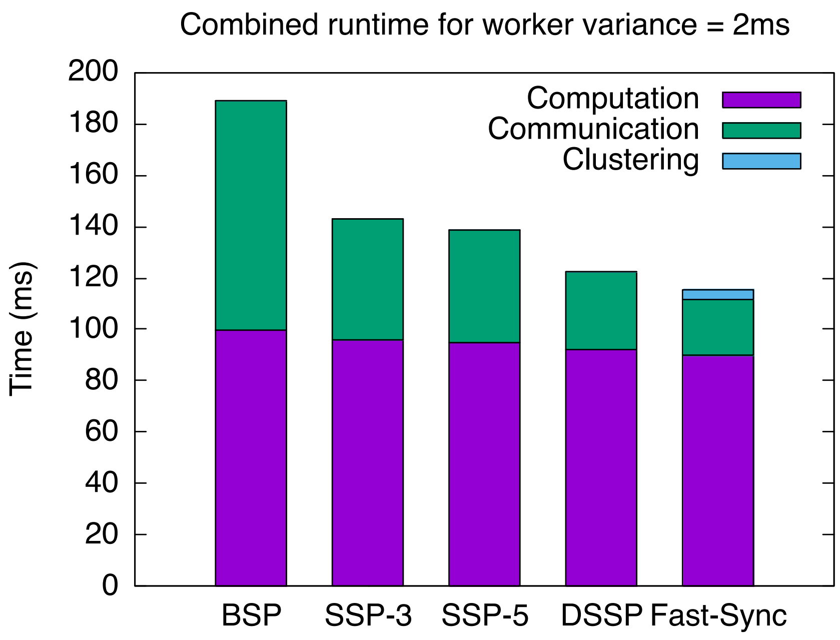


Fig. 6.18 Average computation and communication times for worker execution variance of $2ms$ with 20 workers.

is heavily reliant on the assumption that worker execution times do not vary (or varies minimally) over different iterations.

We vary the worker–controller communication cost from $25ms$ to $75ms$ to measure the impact of network on all algorithms as shown in Fig. 6.20. Our algorithm incurs the least communication overhead for any given worker–controller communication cost. DSSP performs better than both SSP's, while BSP performs worst. The communication overhead incurred by our algorithm increases at a lower rate compared to the other algorithms as the worker–controller communication cost is increased. This is because our algorithm has a bounded number of messages sent within the system.

We increase the number of workers from 5 to 100 to measure how scalable the algorithms are in terms of average communication overhead. Our algorithm outperforms other algorithms for increasing number of workers as shown in Fig. 6.21. Increasing the number of workers has a smaller impact on our algorithm compared to the other algorithms, where the communication overhead

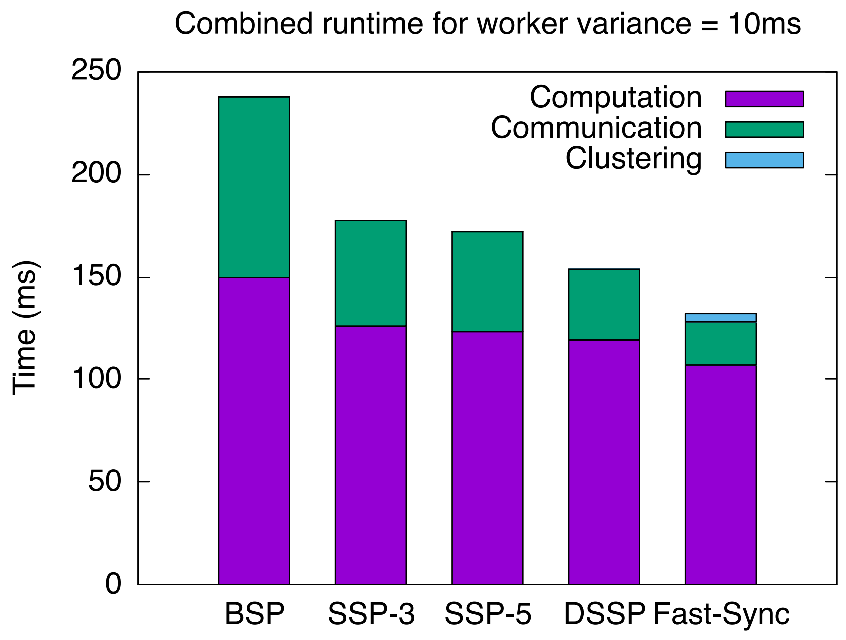


Fig. 6.19 Average computation and communication times for worker execution variance of 10ms with 20 workers.

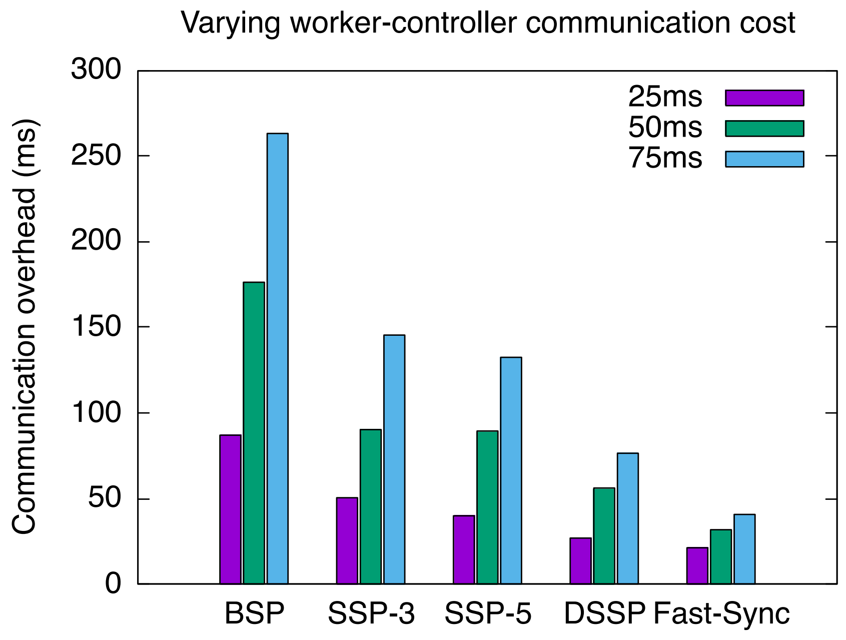


Fig. 6.20 Average communication overhead for varying worker–controller message cost for 20 workers.

is directly proportional to the number of workers. All other algorithms have a significant increase in the communication overhead as the number of workers is increased from 5 to 100.

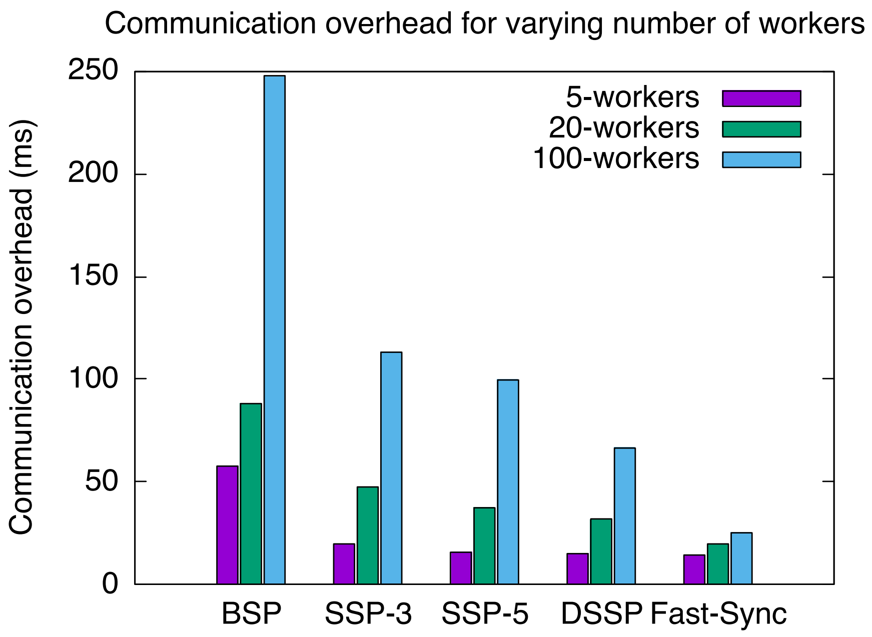


Fig. 6.21 Average communication overhead for varying number of workers.

The BSP, SSP, and DSSP algorithms have been proven to converge. The BSP algorithm will always converge, but the runtime is heavily affected by stragglers. The SSP and DSSP algorithms will converge provided the staleness threshold is within some bound. Our scheme ensures that we have at least a certain ratio of devices to the available before synchronization proceeds. This helps in ensuring that the algorithm will converge.

6.6.4 Simulation Validation

The BSP, SSP, and DSSP synchronization models for training distributed machine learning models and neural networks have been well studied. The results in [142] and [141] show that SSP converges to a consensus faster than BSP. The time to reach convergence for SSP reduces as the number of machines increases, unlike in BSP. Both the BSP and SSP have been found to converge provided the staleness threshold for the SSP is within some bounds. The results in [144] show that DSSP converges faster than SSP in the same corresponding range.

Our results are similar to those in [142], [141] and [144] with regard to execution time. BSP takes a longer time to complete a specified number of iterations, while SSP and DSSP take less time. However, our algorithm outperforms BSP, SSP, and DSSP in terms of execution time.

6.7 Synchronized Distributed Training

6.7.1 Training Details

We train a deep residual neural network model [176], ResNet20 with 20 layers and $270,000$ parameters on the CINIC–10 classification dataset with 10 classes [177] in batches of 128. The dataset contains images from CIFAR–10[2] and ImageNet database images[3] which is split into three parts (train, validation and test), each with $90,000$ images. We combine the train and validation dataset in our experiments for training and use the test dataset for evaluating the accuracy of the model.

The ResNet20 model is trained on both a homogeneous and heterogeneous cluster on Amazon Web Services (AWS) EC2 spot instances to mimic an edge computing system. The homogeneous cluster consists of 3 *g4dn.4xlarge* instance types each with 1 GPU, 16 virtual CPUs, $65GB$ RAM and a network of up to 25 Gigabit. The heterogeneous cluster is used to depict a case where workers have varying computing, processing, and network capabilities. Thus, some workers are expected to be faster than others. The heterogeneous cluster consists of a mixture of 3 AWS EC2 instance types: *g4dn.4xlarge* (1 GPU, 16 virtual CPUs, $65GB$

[2]https://www.cs.toronto.edu/~kriz/cifar.html
[3]http://image-net.org/download-images

RAM and a network of up to 25 Gigabit), *g4dn.2xlarge* (1 GPU, 8 virtual CPUs, 32GB RAM and a network of up to 25 Gigabit) and *g3s.xlarge* (1 GPU, 4 virtual CPUs, 31GB RAM and a network of up to 10 Gigabit). We use the density–based spatial clustering of applications with noise (DBSCAN) clustering algorithm [175] to group workers into clusters. DBSCAN groups together points that are close to each other based on a distance measurement (usually Euclidean distance) and a minimum number of points. It also marks the points that are in low–density regions as outliers.

6.7.2 Evaluation and Discussions

We evaluate the performance of our algorithm by comparing its performance against the ASP, BSP, and SSP (with different staleness threshold) parameter server models. We implement all the frameworks, including our synchronized distributed training algorithm in Ray[4]; a Python framework for developing distributed applications. We measure the training times, training iterations, and testing accuracy for all models for different runtime configurations.

We run the experiments for varying number of workers. Fig. 6.22 shows the number of training iterations required to reach a 70% testing accuracy for the trained ResNet20 model. BSP requires the least number of training iterations for all sets of workers as shown in Fig. 6.22, but each iteration for BSP takes much longer as shown in Fig. 6.23. This is because BSP uses a barrier and all updates from workers must be applied at the parameter server before the workers proceed to the next iteration. The SSP variations with staleness values of 3; SSP3 and 5; SSP5 require less training iterations and times to reach 70% testing accuracy compared to the ASP implementation for the homogeneous cluster setup. Our algorithm spends the least number of iterations and time in reaching 70% accuracy for varying number of workers. The time to reach 70% accuracy decreases as the number of workers increases. This is because more batches are trained when there are more workers.

[4]https://docs.ray.io/en/latest/index.html

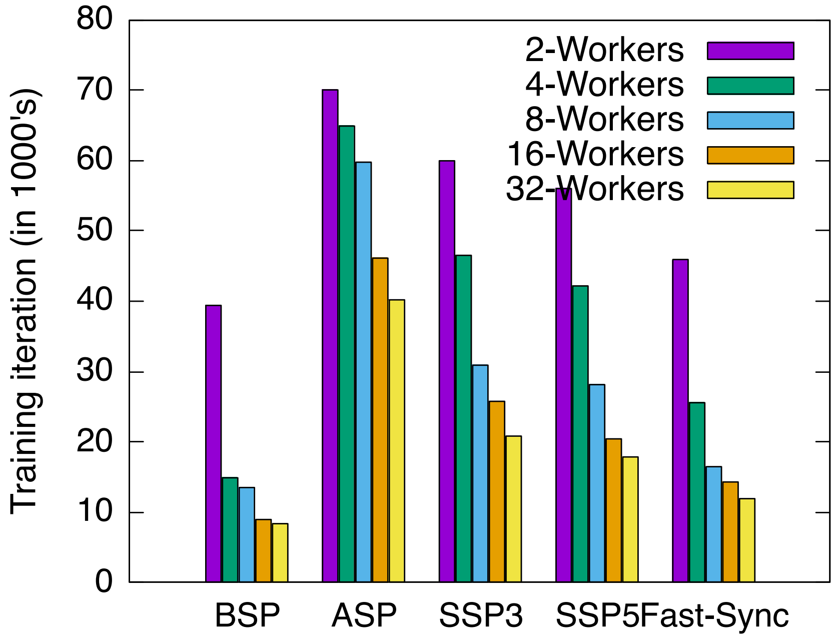


Fig. 6.22 Number of training iterations required to reach 70% testing accuracy for varying number of homogeneous workers.

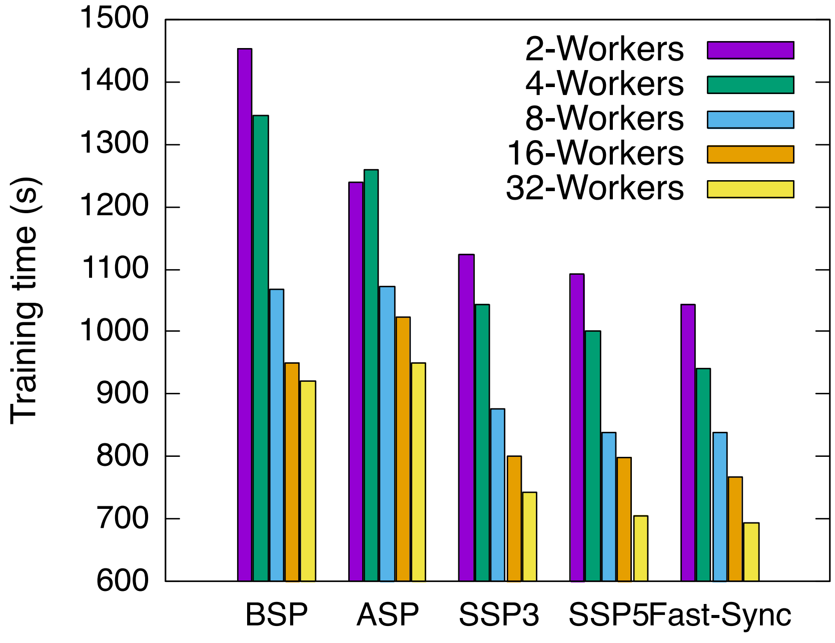


Fig. 6.23 Amount of time required to reach 70% testing accuracy for varying number of homogeneous workers.

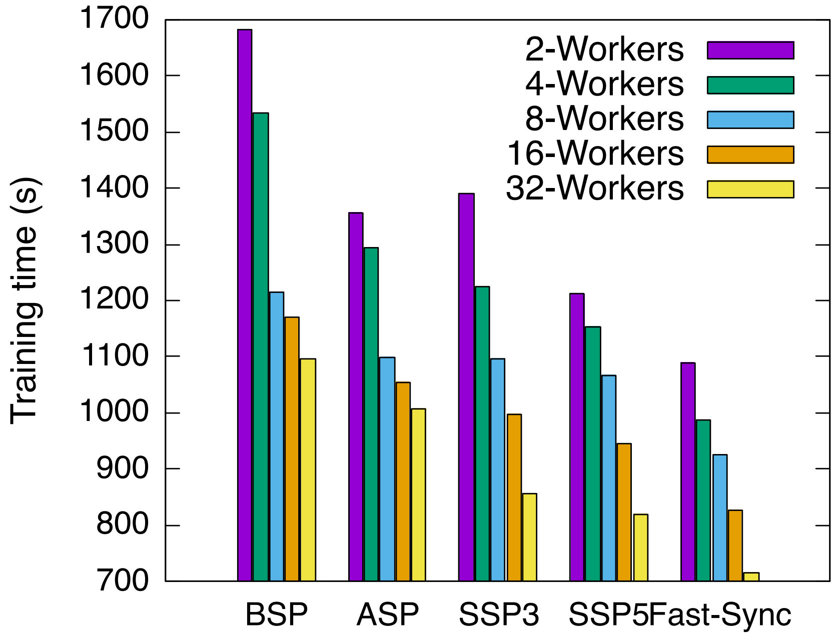


Fig. 6.24 Amount of time required to reach 70% testing accuracy for varying number of heterogeneous workers.

To explore the effect of heterogeneity and to introduce some stragglers among the workers, we train the ResNet20 model in both the homogeneous and heterogeneous setup. The training time to reach 70% accuracy is shown in Fig. 6.23 and 6.24 for the homogeneous and heterogeneous cluster setup respectively. The training time to reach 70% testing accuracy increased for all frameworks, with ASP being less impacted with an average training time increase of 8% closely followed by our algorithm with a 12% increase in training time. BSP was most impacted with an increase of 22% followed by SSP3 and SSP5 respectively.

Finally, we measure the testing accuracy vs training time for 8 workers for both the homogeneous and heterogeneous cluster setups. BSP reaches 45% accuracy faster than other frameworks for the homogeneous cluster setup at 200s training time. Beyond this point, all other frameworks reach higher testing accuracy compared to BSP. Our algorithm performs as well as ASP for earlier training times and as well as the SSP implementations for later training times as

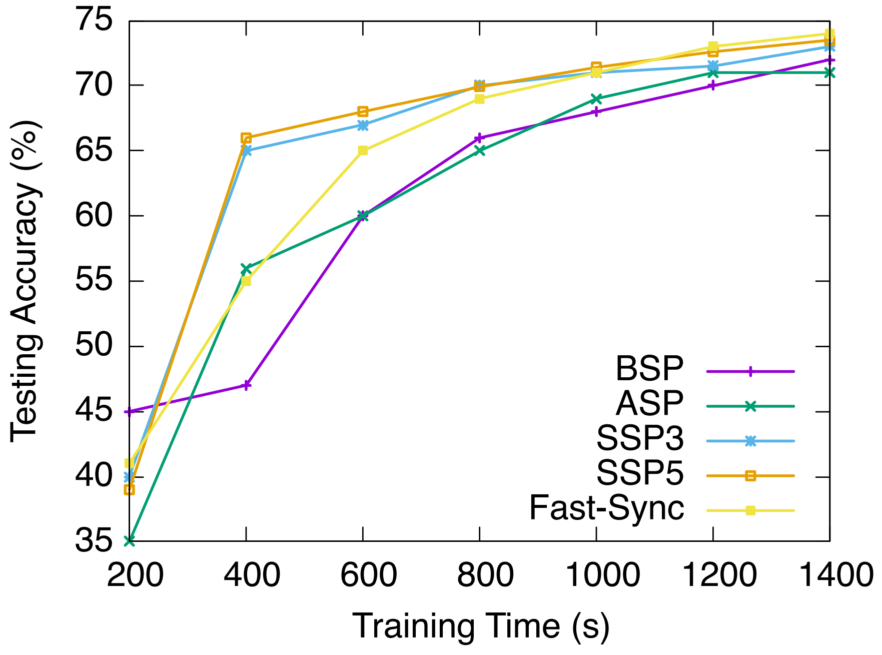


Fig. 6.25 Training time versus testing accuracy for 8 homogeneous workers.

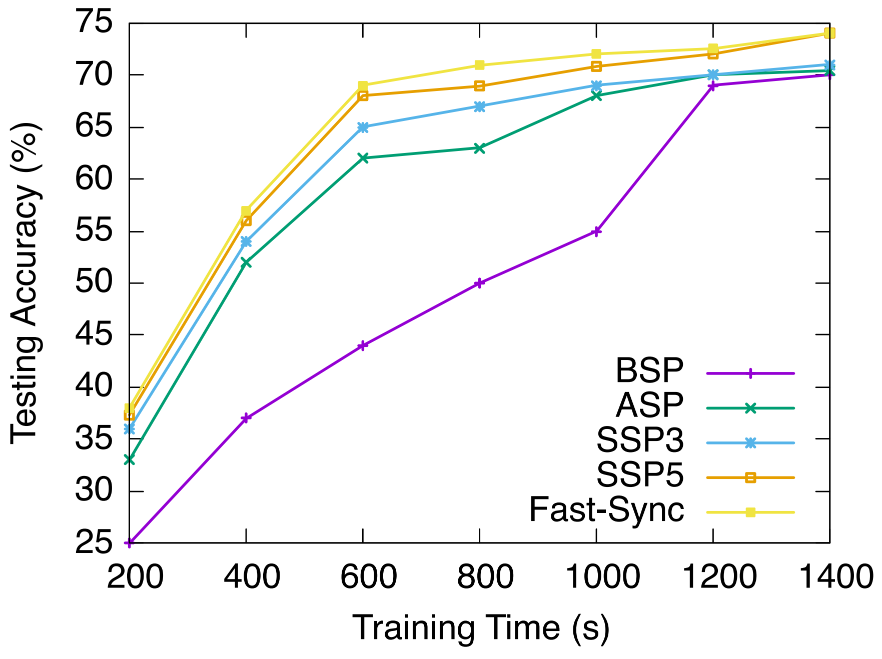


Fig. 6.26 Training time versus testing accuracy for 8 heterogeneous workers.

shown in Fig. 6.25. For the heterogeneous cluster setup, our algorithm achieves an accuracy higher or as good as SSP and ASP for all training times as shown in Fig. 6.26. This is because our algorithm uses clustering to group workers together and the communication among workers is greatly reduced compared to the other models. A table of comparison of our synchronization scheme and related parameter server training models is provided in Table 6.2.

6.8 Deployment Challenges

The following are the potential challenges of deploying our fast synchronization scheme in a real edge–AI system.

1. Heterogeneity on the edge: One of the main issues of distributed or decentralized edge systems is related to the inter–operability of heterogeneous devices and technologies [178]. Although our algorithm mitigates the effects of stragglers, heterogeneous devices with a wide range of execution times for the same tasks will be a bottleneck in realizing fast synchronization.
2. Node faults and failures: Our algorithm incorporates fault tolerance using the late notification protocol, clustering and quorum requirements. However, in a distributed edge system with a high rate of node faults, failures, and recoveries, it becomes a more difficult task to achieve fast synchronization. Even with the optimally fixed synchronization options, an unstable system will be a major issue in reaching synchronization.
3. Network Connectivity: Connectivity among nodes is very important in having fast synchronization. The controller needs to be able to send messages (partial schedules, current cluster composition, etc.). The workers also need to be able to communicate among one another and with the controller. An unstable network will be a huge hindrance to the performance of our algorithm.

Table 6.2 Comparison of our fast synchronization scheme and 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

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we explore synchronization in fog–controlled IoT and AI application tasks. We motivate the need for synchronization in fog–controlled IoT and AI application tasks with use cases and application scenarios. We provide a definition and taxonomy for synchronization in IoT. We develop a system model for mapping applications with and without synchronization requirements to a fog–controlled IoT system. We propose three task–based synchronization algorithms and two redundancy–based synchronization algorithms for task synchronization in IoT. We further propose a game theoretic synchronization approach for AI application tasks.

For the task–based synchronization, we design three synchronization algorithms; static (*SSSA*), dynamic (*DSSA*) and micro batch (*MBSSA*) synchronization algorithms. We evaluate their performance using extensive simulations. We observe from our simulations that *SSSA* is best suited for systems where there is an accurate or almost accurate estimate of the execution time of tasks. Thus, in systems such as IIoT where the behavior of machines is controlled and predictable, *SSSA* will be a good choice for synchronization scheduling. In systems such as vehicular cloud computing and smart things where the behavior of individual components cannot be predicted as they are easily affected by mobility and environmental conditions, *DSSA* is a better choice for achieving synchronization. *MBSSA* is best suited for systems where the arrival pattern of tasks is sporadic and the cost of communication between workers is high. Thus, the tasks can be grouped together and sent as a micro–batch, thereby reducing the cost that would have been incurred by sending the tasks individually.

To introduce fault tolerance into task synchronization in IoT, we design two dynamic synchronization schemes that use the following ideas, respectively: time–based and component–based redundancies. We conduct trace–driven experiments to benchmark and evaluate the performance of the task–based and redundancy–based synchronization algorithms compared to existing solutions.

We observe that time–based redundancy is suitable for applications where repeating task execution is acceptable. While component–based redundancy is suitable for applications where redundancy at the device level is needed, specifically, applications that cannot wait for task reexecutions. Updating the controller with the progress of task execution is important for synchronization scheduling. We find that using a publish–subscribe update scheme reduces the communication load on controllers. Thus, effectively reducing the overall execution time of the synchronous tasks. We observe that an increase in the level of redundancy for component–based redundancy decreases the runtime and reduces the percentage of sync task failure. We also observe that the prediction accuracy of the finish time of tasks on workers has a significant impact on the runtime and synchronization task failure. The proposed redundancy–based algorithms have shorter runtimes compared to barrier synchronization and have fewer synchronization task failures when compared to time–slotted synchronization.

Additionally, we present a game theoretic synchronization approach for AI application tasks. Our approach reduces the number of messages needed in reaching synchronization through the use of clustering and a disconnection–tolerant late notification protocol. Existing protocols such as BSP, SSP, and DSSP decide the synchronization time only when the workers get to the synchronization point. A lot of messages are then needed in reaching a consensus on synchronization. We develop a game to help in deciding the optimal number of synchronization options and in fixing them. Thus, during runtime, workers do not need to communicate with each other to reach, postpone, or abort synchronization. The only messages sent during the synchronization process are late notifications, which are bounded.

We report on a simulation study that evaluates the benefits of our fast synchronization scheme. In particular, we explore the performance of our synchronization scheme under different operation conditions. We show that our scheme performs well with increasing number of workers and increasing heterogeneity among workers. We compare our scheme with the BSP and SSP (with different staleness thresholds) and show that our scheme performs better or as well as both BSP and SSP.

We further implement the fast synchronization algorithm in Ray (a Python framework for distributed applications) and evaluate its performance by comparing it with ASP, BSP and SSP models under different cluster setups. We train a ResNet20 model on all the frameworks on AWS EC2 using the CINIC–10 dataset. We show that our algorithm performs better or as well as other models for varying number of workers and different cluster configurations.

7.2 Future Work

One area of future work is to handle device mobility across the fogs in fog–controlled IoT. In particular, with vehicular clouds, we can have vehicles joining and leaving different fog zones as they travel. The synchronization scheduler needs to control the vehicle–to–fog associations to minimize synchronization task failures due to mobility. We look to develop prediction models to forecast the availability of nodes in a rapidly changing network model, which will eliminate the need for quorum checking. Extending the micro batching idea to not just tasks but devices to achieve more localization is a future research direction. We also look to apply machine learning for completion time prediction and incorporate that into synchronization scheduling.

As part of future work, we would like to test our task synchronization approaches in a real setting. An implementation will explore real network conditions, connection between nodes, communication delays, mobility, faults, and failures.

Another area of future work is to extend the current game–theoretic synchronization approach to having more than 2 clusters. This way, we have more fine–grained clusters that will have higher probability of staying tightly synchronized. Finally, we hope to fully implement our synchronization scheme into a framework and programming language for AI application tasks. This would allow us to evaluate our synchronization scheme under real–life scenarios.

References

[1] S. Jeschke, C. Brecher, T. Meisen, D. Özdemir, and T. Eschert, *Industrial Internet of Things and Cyber Manufacturing Systems*, pp. 3–19. Cham: Springer International Publishing, 2017.

[2] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial internet of things: Challenges, opportunities, and directions," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 11, pp. 4724–4734, 2018.

[3] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial internet of things (iiot): An analysis framework," *Computers in industry*, vol. 101, pp. 1–12, 2018.

[4] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber, "A vision of swarmlets," *IEEE Internet Computing*, vol. 19, no. 2, pp. 20–28, 2015.

[5] E. A. Lee, B. Hartmann, J. Kubiatowicz, T. S. Rosing, J. Wawrzynek, D. Wessel, J. Rabaey, K. Pister, A. Sangiovanni-Vincentelli, S. A. Seshia, *et al.*, "The swarm at the edge of the cloud," *IEEE Design & Test*, vol. 31, no. 3, pp. 8–20, 2014.

[6] H. Chaouchi, ed., *The Internet of Things: Connecting Objects*. Hoboken, NJ, London: Wiley-ISTE, 2010.

[7] X. Cui, "The internet of things," in *Ethical ripples of creativity and innovation*, pp. 61–68, Springer, 2016.

[8] F. Wortmann and K. Flüchter, "Internet of things," *Business & Information Systems Engineering*, vol. 57, no. 3, pp. 221–224, 2015.

[9] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International journal of communication systems*, vol. 25, no. 9, p. 1101, 2012.

[10] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[11] S. F. Abedin, M. G. R. Alam, N. H. Tran, and C. S. Hong, "A fog based system model for cooperative iot node pairing using matching theory," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, pp. 309–314, IEEE, 2015.

[12] L. Apvrille, T. Tanzi, and J.-L. Dugelay, "Autonomous drones for assisting rescue services within the context of natural disasters," in *General Assembly and Scientific Symposium (URSI GASS), 2014 XXXIth URSI*, pp. 1–4, IEEE, 2014.

[13] A. Valsan, B. Parvathy, V. D. GH, R. Unnikrishnan, P. K. Reddy, and A. Vivek, "Unmanned aerial vehicle for search and rescue mission," in *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, pp. 684–687, IEEE, 2020.

[14] S. Mayer, L. Lischke, and P. W. Woźniak, "Drones for search and rescue," in *1st International Workshop on Human-Drone Interaction*, 2019.

[15] S. Savazzi, M. Nicoli, and V. Rampa, "Federated learning with cooperating devices: A consensus approach for massive iot networks," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4641–4654, 2020.

[16] S. Karnouskos, "The cooperative internet of things enabled smart grid," in *Proceedings of the 14th IEEE international symposium on consumer electronics (ISCE2010), June*, pp. 07–10, 2010.

[17] X. Guo, M. Mohammad, S. Saha, M. C. Chan, S. Gilbert, and D. Leong, "Psync: Visible light-based time synchronization for internet of things (iot)," 2016.

[18] A. Elsts, X. Fafoutis, S. Duquennoy, G. Oikonomou, R. J. Piechocki, and I. Craddock, "Temperature-resilient time synchronization for the internet of things," *IEEE Transactions on Industrial Informatics*, 2017.

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

[20] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[21] L. M. Vaquero and L. Rodero-Merino, “Finding your way in the fog: Towards a comprehensive definition of fog computing,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[22] A. V. Dastjerdi and R. Buyya, “Fog computing: Helping the internet of things realize its potential,” *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

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

[24] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

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

[26] G. Premsankar, M. Di Francesco, and T. Taleb, “Edge computing for the internet of things: A case study,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, 2018.

[27] Q. C. Li, H. Niu, A. T. Papathanassiou, and G. Wu, “5g network capacity: Key elements and technologies,” *IEEE Vehicular Technology Magazine*, vol. 9, no. 1, pp. 71–78, 2014.

[28] P. Khodashenas, J. Aznar, A. Legarrea, C. Ruiz, M. Siddiqui, E. Escalona, and S. Figuerola, “5g network challenges and realization insights,” in *2016 18th International Conference on Transparent Optical Networks (ICTON)*, pp. 1–4, IEEE, 2016.

[29] I. Chih-Lin, S. Han, Z. Xu, Q. Sun, and Z. Pan, “5g: rethink mobile communications for 2020+,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2062, p. 20140432, 2016.

[30] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey,” *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[31] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, “A convolutional neural network for modelling sentences,” *arXiv preprint arXiv:1404.2188*, 2014.

[32] S. Albawi, T. A. Mohammed, and S. Al-Zawi, “Understanding of a convolutional neural network,” in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, Ieee, 2017.

[33] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[34] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[35] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[36] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[37] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, "In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning," *IEEE Network*, vol. 33, no. 5, pp. 156–165, 2019.

[38] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.

[39] A. H. Sodhro, S. Pirbhulal, and V. H. C. de Albuquerque, "Artificial intelligence-driven mechanism for edge computing-based industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4235–4243, 2019.

[40] S. B. Calo, M. Touna, D. C. Verma, and A. Cullen, "Edge computing architecture for applying ai to iot," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 3012–3016, IEEE, 2017.

[41] Y. Bengio, Y. LeCun, *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[42] T. J. Sejnowski, *The deep learning revolution.* Mit Press, 2018.

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

[44] M. Rhudy, "Time alignment techniques for experimental sensor data," *International Journal of Computer Science and Engineering Survey*, vol. 5, no. 2, p. 1, 2014.

[45] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, “Project adam: Building an efficient and scalable deep learning training system,” in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 571–582, 2014.

[46] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, *et al.*, “Exploiting bounded staleness to speed up big data analytics,” in *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14)*, pp. 37–48, 2014.

[47] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, “Large scale distributed deep networks,” *Advances in neural information processing systems*, vol. 25, pp. 1223–1231, 2012.

[48] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, “Scaling distributed machine learning with the parameter server,” in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 583–598, 2014.

[49] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, “Petuum: A new platform for distributed machine learning on big data,” *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[50] A. Gautam and S. Mohan, “A review of research in multi-robot systems,” in *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on*, pp. 1–5, IEEE, 2012.

[51] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, “Fog-enabled multi-robot systems,” in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pp. 1–10, IEEE, 2018.

[52] S. Cervini, “System and method for efficiently executing single program multiple data (spmd) programs,” Mar. 8 2011. US Patent 7,904,905.

[53] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[54] Z. Qin, G. Denker, C. Giannelli, P. Bellavista, and N. Venkatasubramanian, “A software defined networking architecture for the internet-of-things,” in *2014 IEEE network operations and management symposium (NOMS)*, pp. 1–9, IEEE, 2014.

[55] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology," *RFC 7426*, 2015.

[56] R. Wenger, X. Zhu, J. Krishnamurthy, and M. Maheswaran, "A programming language and system for heterogeneous cloud of things," in *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pp. 169–177, IEEE, 2016.

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

[58] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[59] Y. Jararweh, A. Doulat, A. Darabseh, M. Alsmirat, M. Al-Ayyoub, and E. Benkhelifa, "Sdmec: Software defined system for mobile edge computing," in *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pp. 88–93, IEEE, 2016.

[60] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[61] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[62] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proc. of the Sixth International Conference on Advances in Future Internet*, pp. 48–55, Citeseer, 2014.

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

[64] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, "Vehicular edge computing and networking: A survey," *Mobile Networks and Applications*, pp. 1–24, 2020.

[65] J. Feng, Z. Liu, C. Wu, and Y. Ji, "Ave: Autonomous vehicular edge computing framework with aco-based scheduling," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 12, pp. 10660–10675, 2017.

[66] Y.-L. Lee, P.-K. Tsung, and M. Wu, "Techology trend of edge ai," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–2, IEEE, 2018.

[67] J. P. Queralta, T. N. Gia, H. Tenhunen, and T. Westerlund, "Edge-ai in lora-based health monitoring: Fall detection system with fog computing and lstm recurrent neural networks," in *2019 42nd international conference on telecommunications and signal processing (TSP)*, pp. 601–604, IEEE, 2019.

[68] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge {AI}," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[69] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "Softran: Software defined radio access network," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 25–30, 2013.

[70] Y.-J. Ku, D.-Y. Lin, C.-F. Lee, P.-J. Hsieh, H.-Y. Wei, C.-T. Chou, and A.-C. Pang, "5g radio access network design with the fog paradigm: Confluence of communications and computing," *IEEE Communications Magazine*, vol. 55, no. 4, pp. 46–52, 2017.

[71] B. Liang, V. Wong, R. Schober, D. Ng, and L. Wang, "Mobile edge computing," *Key technologies for 5G wireless systems*, vol. 16, no. 3, pp. 1397–1411, 2017.

[72] S. Park and Y. Yoo, "Network intelligence based on network state information for connected vehicles utilizing fog computing," *Mobile Information Systems*, vol. 2017, 2017.

[73] A. Kumari, S. Tanwar, S. Tyagi, N. Kumar, M. S. Obaidat, and J. J. Rodrigues, "Fog computing for smart grid systems in the 5g environment: Challenges and solutions," *IEEE Wireless Communications*, vol. 26, no. 3, pp. 47–53, 2019.

[74] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–43, 2017.

[75] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proceedings of the 2015 workshop on mobile big data*, pp. 37–42, 2015.

[76] M. Rausch, B. Müller, B. Hedenetz, and A. Schedl, "Clock synchronization in a distributed system," Mar. 25 2008. US Patent 7,349,512.

[77] Q. Li and D. Rus, "Global clock synchronization in sensor networks," *IEEE Transactions on computers*, vol. 55, no. 2, pp. 214–226, 2006.

[78] K. Xie, Q. Cai, and M. Fu, "A fast clock synchronization algorithm for wireless sensor networks," *Automatica*, vol. 92, pp. 133–142, 2018.

[79] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "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, 2018.

[80] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi, "Towards making nekostat a proper measurement tool for the validation of distributed systems," in *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, pp. 377–386, IEEE, 2007.

[81] W. Steiner, F. Bonomi, and H. Kopetz, "Towards synchronous deterministic channels for the internet of things," in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pp. 433–436, IEEE, 2014.

[82] S. K. Mani, R. Durairajan, P. Barford, and J. Sommers, "A system for clock synchronization in an internet of things," *arXiv preprint arXiv:1806.02474*, 2018.

[83] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, "Clock synchronization for wireless sensor networks: a survey," *Ad hoc networks*, vol. 3, no. 3, pp. 281–323, 2005.

[84] C. Lenzen, T. Locher, P. Sommer, and R. Wattenhofer, "Clock synchronization: Open problems in theory and practice," in *International Conference on Current Trends in Theory and Practice of Computer Science*, pp. 61–70, Springer, 2010.

[85] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 39–49, 2004.

[86] P. Jia, X. Wang, and K. Zheng, "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, 2019.

[87] L. Schenato and F. Fiorentin, "Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks," *Automatica*, vol. 47, no. 9, pp. 1878–1886, 2011.

[88] U. Schmid, "Synchronized utc for distributed real-time systems," *Annual Review in Automatic Programming*, vol. 18, pp. 101–107, 1994.

[89] K. Fan, S. Sun, Z. Yan, Q. Pan, H. Li, and Y. Yang, "A blockchain-based clock synchronization scheme in iot," *Future Generation Computer Systems*, vol. 101, pp. 524–533, 2019.

[90] N. Xu, X. Zhang, Q. Wang, J. Liang, G. Pan, and M. Zhang, "An improved flooding time synchronization protocol for industrial wireless networks," in *2009 International Conference on Embedded Software and Systems*, pp. 524–529, IEEE, 2009.

[91] H. D. Karatza, "Scheduling gangs in a distributed system," *International Journal of Simulation: Systems, Science Technology, UK Simulation Society*, vol. 7, no. 1, pp. 15–22, 2006.

[92] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pp. 109–114, IEEE, 2000.

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

[94] G. L. Stavrinides and H. D. Karatza, "Scheduling different types of gang jobs in distributed systems," in *2019 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 1–5, IEEE, 2019.

[95] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic coscheduling on workstation clusters," in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 231–256, Springer, 1998.

[96] K. Deng, K. Ren, M. Zhu, and J. Song, "A data and task co-scheduling algorithm for scientific cloud workflows," *IEEE Transactions on Cloud Computing*, vol. 8, no. 2, pp. 349–362, 2015.

[97] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[98] E. Frachtenberg, F. Petrini, S. Coll, and W.-c. Feng, "Gang scheduling with lightweight user-level communication," in *Parallel Processing Workshops, 2001. International Conference on*, pp. 339–345, IEEE, 2001.

[99] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[100] J. Kirsch and Y. Amir, "Paxos for system builders: An overview," in *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, (New York, NY, USA), pp. 3:1–3:6, ACM, 2008.

[101] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made epr: decidable reasoning about distributed protocols," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–31, 2017.

[102] Á. García-Pérez, A. Gotsman, Y. Meshman, and I. Sergey, "Paxos consensus, deconstructed and abstracted," in *European Symposium on Programming*, pp. 912–939, Springer, Cham, 2018.

[103] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: Fast and scalable paxos on rdma," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 94–107, 2017.

[104] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[105] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 169–182, ACM, 2013.

[106] P. Jakovits, S. N. Srirama, and I. Kromonov, "Stratus: A distributed computing framework for scientific simulations on the cloud," in *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 1053–1059, IEEE, 2012.

[107] L. G. Valiant, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011.

[108] E. Vogli, G. Ribezzo, L. A. Grieco, and G. Boggia, "Fast join and synchronization schema in the ieee 802.15. 4e mac," in *2015 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pp. 85–90, IEEE, 2015.

[109] I. Ozil and D. R. Brown, "Time-slotted round-trip carrier synchronization," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, pp. 1781–1785, IEEE, 2007.

[110] S. Zeng, B. He, and J. Jiang, "A scheduling algorithm for synchronization task in embedded multicore systems," *Journal of Computational Information Systems*, vol. 10, no. 19, pp. 8531–8541, 2014.

[111] F. Nemati, M. Behnam, and T. Nolte, "Multiprocessor synchronization and hierarchical scheduling," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pp. 58–64, IEEE, 2009.

[112] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 469–478, IEEE, 2009.

[113] R. Rajkumar, *Synchronization in real-time systems: a priority inheritance approach*, vol. 151. Springer Science & Business Media, 2012.

[114] A. Vrancic, "Synchronization of distributed systems," Sept. 26 2006. US Patent 7,114,091.

[115] S. G. Yoo, S. Park, and W.-Y. Lee, "A study of time synchronization methods for iot network nodes," *International journal of advanced smart convergence*, vol. 9, no. 1, pp. 109–112, 2020.

[116] A. Tyrrell, G. Auer, and C. Bettstetter, "Fireflies as role models for synchronization in ad hoc networks," in *Proceedings of the 1st international conference on Bio inspired models of network, information and computing systems*, p. 4, ACM, 2006.

[117] R. E. Mirollo and S. H. Strogatz, "Synchronization of pulse-coupled biological oscillators," *SIAM Journal on Applied Mathematics*, vol. 50, no. 6, pp. 1645–1662, 1990.

[118] Y. Kuramoto, "Self-entrainment of a population of coupled non-linear oscillators," in *International symposium on mathematical problems in theoretical physics*, pp. 420–422, Springer, 1975.

[119] B. Ermentrout, "An adaptive model for synchrony in the firefly pteroptyx malaccae," *Journal of Mathematical Biology*, vol. 29, no. 6, pp. 571–585, 1991.

[120] Y. Sun, Q. Jiang, and K. Zhang, "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, 2012.

[121] Y.-W. Hong and A. Scaglione, "A scalable synchronization protocol for large scale sensor networks and its applications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 5, pp. 1085–1099, 2005.

[122] P. Yadav, J. A. McCann, and T. Pereira, "Self-synchronization in duty-cycled internet of things (iot) applications," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 2058–2069, 2017.

[123] O. Babaoglu, T. Binci, M. Jelasity, and A. Montresor, "Firefly-inspired heartbeat synchronization in overlay networks," in *Self-Adaptive and Self-Organizing Systems, 2007. SASO'07. First International Conference on*, pp. 77–86, IEEE, 2007.

[124] R. Leidenfrost and W. Elmenreich, "Establishing wireless time-triggered communication using a firefly clock synchronization approach.," in *WISES*, pp. 1–18, Citeseer, 2008.

[125] I. Bojić and M. Kušek, "Fireflies synchronization in small overlay networks," in *32nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2009*, 2009.

[126] H. Yin, P. G. Mehta, S. P. Meyn, and U. V. Shanbhag, "Synchronization of coupled oscillators is a game," *IEEE Transactions on Automatic Control*, vol. 57, no. 4, pp. 920–935, 2011.

[127] R. Pagliari and A. Scaglione, "Scalable network synchronization with pulse-coupled oscillators," *IEEE Transactions on Mobile Computing*, vol. 10, no. 3, pp. 392–405, 2011.

[128] J. Klinglmayr, C. Kirst, C. Bettstetter, and M. Timme, "Guaranteeing global synchronization in networks with stochastic interactions," *New Journal of Physics*, vol. 14, no. 7, p. 073031, 2012.

[129] K. Siddique, Z. Akhtar, E. J. Yoon, Y.-S. Jeong, D. Dasgupta, and Y. Kim, "Apache hama: An emerging bulk synchronous parallel computing

framework for big data applications," *IEEE Access*, vol. 4, pp. 8879–8887, 2016.

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

[131] J. Keuper and F.-J. Pfreundt, "Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, pp. 1–11, 2015.

[132] D. Alistarh, Z. Allen-Zhu, and J. Li, "Byzantine stochastic gradient descent," *Advances in Neural Information Processing Systems*, vol. 31, pp. 4613–4623, 2018.

[133] M. M. Amiri and D. Gündüz, "Machine learning at the wireless edge: Distributed stochastic gradient descent over-the-air," *IEEE Transactions on Signal Processing*, vol. 68, pp. 2155–2169, 2020.

[134] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, pp. 2595–2603, 2010.

[135] L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč, "Sarah: A novel method for machine learning problems using stochastic recursive gradient," *arXiv preprint arXiv:1703.00102*, 2017.

[136] T. White, *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[137] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[138] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010.

[139] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[140] X. Zhao, M. Papagelis, A. An, B. X. Chen, J. Liu, and Y. Hu, "Elastic bulk synchronous parallel model for distributed deep learning," in *2019 IEEE*

International Conference on Data Mining (ICDM), pp. 1504–1509, IEEE, 2019.

[141] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231, 2013.

[142] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[143] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. Xing, "High-performance distributed ml at scale through parameter server consistency models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[144] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1507–1517, IEEE, 2019.

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

[146] F. Liu and W. Guo, "The design and implementation of mina-based smart home data synchronization system," in *2015 Fifth International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC)*, pp. 1612–1616, Sept 2015.

[147] H. Wang, B. Kim, J. Xie, and Z. Han, "E-auto: A communication scheme for connected vehicles with edge-assisted autonomous driving," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2019.

[148] L. Gillam, K. Katsaros, M. Dianati, and A. Mouzakitis, "Exploring edges for connected and autonomous driving," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 148–153, IEEE, 2018.

[149] S. Muddala, D. K. Divya, P. Nimbalkar, and R. Patil, "Iot based bridge monitoring system," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 5, no. 2, pp. 2044–2047, 2019.

[150] D. J. Glancy, "Autonomous and automated and connected cars-oh my: first generation autonomous cars in the legal ecosystem," *Minn. JL Sci. & Tech.*, vol. 16, p. 619, 2015.

[151] M. Campbell, M. Egerstedt, J. P. How, and R. M. Murray, "Autonomous driving in urban environments: approaches, lessons and challenges," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1928, pp. 4649–4672, 2010.

[152] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 163–168, IEEE, 2011.

[153] M. N. Hasan, S. Didar-Al-Alam, and S. R. Huq, "Intelligent car control for a smart car," *International Journal of Computer Applications*, vol. 14, no. 3, pp. 15–19, 2011.

[154] J. Marquez-Barja, B. Lannoo, D. Naudts, B. Braem, C. Donato, V. Maglogiannis, S. Mercelis, R. Berkvens, P. Hellinckx, M. Weyn, *et al.*, "Smart highway: Its-g5 and c2vx based testbed for vehicular communications in real environments enhanced by edge/cloud technologies," in *EuCNC2019, the European Conference on Networks and Communications*, IEEE, 2019.

[155] U. Z. A. Hamid, H. Zamzuri, and D. K. Limbu, "Internet of vehicle (iov) applications in expediting the implementation of smart highway of autonomous vehicle: A survey," in *Performability in Internet of Things*, pp. 137–157, Springer, 2019.

[156] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[157] Y. Hu, D. Niu, J. Yang, and S. Zhou, "Fdml: A collaborative machine learning framework for distributed features," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2232–2240, 2019.

[158] A. Elgabli, J. Park, A. S. Bedi, M. Bennis, and V. Aggarwal, "Gadmm: Fast and communication efficient framework for distributed machine learning.," *Journal of Machine Learning Research*, vol. 21, no. 76, pp. 1–39, 2020.

[159] W. Wang, C. Zhang, L. Yang, J. Xia, K. Chen, and K. Tan, "Divide-and-shuffle synchronization for distributed machine learning," *arXiv preprint arXiv:2007.03298*, 2020.

[160] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[161] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[162] M. Mohri, G. Sivek, and A. T. Suresh, "Agnostic federated learning," in *International Conference on Machine Learning*, pp. 4615–4625, PMLR, 2019.

[163] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[164] A. M. Al-Qawasmeh, A. A. Maciejewski, H. Wang, J. Smith, H. J. Siegel, and J. Potter, "Statistical measures for quantifying task and machine heterogeneities," *The Journal of Supercomputing*, vol. 57, no. 1, pp. 34–50, 2011.

[165] J.-Y. Huang, P.-H. Tsai, and I.-E. Liao, "Implementing publish/subscribe pattern for coap in fog computing environment," in *2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 175–180, IEEE, 2017.

[166] A. Ganesh, "Publish/subscribe model in a wireless sensor network," Sept. 15 2009. US Patent 7,590,098.

[167] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Wireless Networks*, vol. 10, no. 6, pp. 643–652, 2004.

[168] R. Gracia-Tinedo, M. S. Artigas, A. Moreno-Martinez, C. Cotes, and P. G. Lopez, "Actively measuring personal cloud storage," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pp. 301–308, IEEE, 2013.

[169] S. Liu, Y. Liu, L. M. Ni, J. Fan, and M. Li, “Towards mobility-based clustering,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 919–928, ACM, 2010.

[170] Y. C. Shin and C. Xu, *Intelligent systems: modeling, optimization, and control*. CRC press, 2017.

[171] S. Wang, J. Wan, D. Zhang, D. Li, and C. Zhang, “Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination,” *Computer Networks*, vol. 101, pp. 158–168, 2016.

[172] J. Rios-Torres and A. A. Malikopoulos, “A survey on the coordination of connected and automated vehicles at intersections and merging at highway on-ramps,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 5, pp. 1066–1077, 2016.

[173] L. Lovén, T. Leppänen, E. Peltonen, J. Partala, E. Harjula, P. Porambage, M. Ylianttila, and J. Riekki, “Edge ai: A vision for distributed, edge-native artificial intelligence in future 6g networks,” *The 1st 6G Wireless Summit*, pp. 1–2, 2019.

[174] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, “Edge intelligence: the confluence of edge computing and artificial intelligence,” *IEEE Internet of Things Journal*, 2020.

[175] K. M. Kumar and A. R. M. Reddy, “A fast dbscan clustering algorithm by accelerating neighbor searching using groups method,” *Pattern Recognition*, vol. 58, pp. 39–48, 2016.

[176] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[177] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey, “Cinic-10 is not imagenet or cifar-10,” *arXiv preprint arXiv:1810.03505*, 2018.

[178] J. Portilla, G. Mujica, J.-S. Lee, and T. Riesgo, “The extreme edge at the bottom of the internet of things: A review,” *IEEE Sensors Journal*, vol. 19, no. 9, pp. 3179–3190, 2019.