Evaluation of CPU Scheduling and Synchronization Algorithms in Distributed Systems

Iyas A Qaddara[1], Amneh J Kenana[2], Khawla M Al-Tarawneh[3], Sami Sarhan[4]

[1]Part-time lecturer, Prince Abdullah Bin Ghazi Faculty for Information Technology Department of Computer Science, Al- Balqa' Applied University- Amman, Jordan, Iyas.qaddarra@bau.edu.jo
[2]Part-time lecturer, Prince Abdullah Bin Ghazi Faculty for Information Technology Department of Computer Science, Al- Balqa' Applied University- Amman, Jordan, amneh.kenana@bau.edu.jo
[3]Researcher, King Abdullah || School of Information Technology, Department of Computer Science, The University of Jordan- Amman, Jordan, Kol9220471@ju.edu.jo
[4]Professor, King Abdullah || School of Information Technology, DEPARTMENT of Computer Science, The University of Jordan- Amman, Jordan, samiserh@ju.edu.jo

This paper presents an evaluation model designed to compare various CPU scheduling and synchronization algorithms in the context of a distributed system. A simulation model is constructed with six nodes, each equipped with two active and three passive resources, and featuring a periodic task per node. Despite limitations on simulation scale due to resource constraints, the study applies four CPU scheduling algorithms and finds that the Earliest Deadline First (EDF) algorithm outperforms others in terms of system performance. Additionally, the paper explores synchronization algorithms, highlighting the CMH algorithm's focus on snapshot collecting, and distinguishing between 2PC and 3PC for transactional operations. Furthermore, the study discusses the characteristics of Raft and Paxos in ensuring consistency during distributed system operations, with Raft emphasizing identical indexes and terms for identical operations, and Paxos favoring the candidate with the higher term in the event of simultaneous candidacy.

Keywords: Earliest Deadline First (EDF), nodes, CPU.

1. Introduction

Optimal performance and responsiveness in the dynamic world of distributed systems depend heavily on the effective use of computer resources. The algorithms controlling CPU scheduling and synchronization, which are essential elements in coordinating the smooth execution of activities across dispersed settings, are at the heart of this efficiency. Robust and adaptable algorithms are more and more necessary as computational demands rise and systems get more intricate. Distributed systems provide a complex problem when it comes to managing CPU resources. Choosing the sequence in which operations are carried out on a CPU is known as CPU scheduling, and it is a crucial component of operating system architecture. Optimizing resource usage, reducing latency, and improving overall system responsiveness are all dependent on efficient scheduling algorithms. [12]

At the same time, distributing the CPU load throughout a distributed system is crucial for reaching workload balance. The system as a whole may perform worse due to resource limitations caused by uneven CPU distribution. By ensuring that every node or processor in the distributed network runs well, balancing the computational load adds to the overall stability of the system.

In a distributed system, synchronizing tasks among several processors or nodes at the same time is crucial to provide dependable and coordinated execution. By minimizing inconsistencies in data and ensuring orderly access to shared resources, using synchronization strategies can help prevent racial tensions and conflicts. To meet this need, this study does a thorough analysis of ten different CPU scheduling and synchronization techniques. We want to shed light on the advantages, disadvantages, and performance characteristics of these algorithms in the context of distributed systems by painstaking research and benchmarking. We evaluate a wide variety of methods, many of which are intended to address the difficulties associated with distributed processing. [13]

To maximize the performance of distributed systems, developers and system administrators must comprehend the subtleties of these algorithms. We hope to shed light on their complexities and offer useful insights that support the continuous improvement of distributed computing techniques.

The structure of this paper is as follows: Section 1 provides a concise synopsis of the theoretical underpinnings of CPU scheduling and synchronization techniques. with special attention to important factors to consider when dealing with distributed systems. We discuss similar work in Section 2 after providing an algorithmic overview. The approach used in our thorough analysis is described in Section 3. To provide openness in our proposed methodology and support the reproducibility of our study, we share insights into the experimental setup and clarify the measurements that were employed. And lastly, we describe our research's findings and conclusions.[12]

2. LITRETURE REVIEW

[1], The purpose of this article is to review the research on CPU scheduling algorithms and compare them to determine which is the best. They discovered that while several researchers have proposed different approaches to improve CPU optimization criteria through different algorithms to improve the waiting time, response time, and turnaround time, no algorithm is better in all criteria after reviewing the Round Robin, Shortest Job First, First Come First Served, and Priority algorithms. As the results demonstrated, CPU algorithm scheduling aids in lowering process wait and response times, they generally feel that optimal CPU algorithm

scheduling enhances CPU performance. Additionally, the user can achieve good results without spending time thanks to the CPU algorithms.

[2], To get over these restrictions, they put up a distributed coordinating approach in this study. To this purpose, they propose a unique approach that remains asynchronous in its operation and ensures finite-time convergence while accounting for these possible latency fluctuations in the form of explicit delays in the communication channels during planning. Using networks with different delays and widths, which represented real-world data center deployments in accordance with standard deployment requirements, they assessed their suggested approach. We compared our asynchronous finite-time algorithm's performance to data center CPU resource allocation. Better overall system usage and responsiveness can result from more effective resource allocation.

[3], They explore advancements in scheduling in cluster and grid computing before introducing the issue of scheduling in distributed systems. The state-of-the-art in each taxonomy tree branch is then described, followed by an outline of the suggested taxonomy. Subsequently, they concentrate on cloud computing and discuss the parallels and discrepancies between scheduling in clouds and distributed systems that were previously in use. Extending the pre-cloud taxonomy, they present a taxonomy of scheduling in cloud computing. Lastly, they talk about new issues that need to be resolved as well as research challenges that the cloud computing paradigm carried over from grid and cluster computing. They reviewed a number of topics in the distributed systems scheduling literature and proposed a taxonomy that includes the application model and target system, the scheduling objectives, the frequency of the scheduler's execution, the input and output data of the scheduler, and the organization of the scheduler within the system. The taxonomy is expanded to incorporate branches that emerged with the advent of the more recent cloud computing paradigm. These branches bring with them new aspects of the target system, utility perspectives regarding the target system and scheduling objectives, and distinct application models that arise from the cloud's virtualized nature.

[4], This work proposes a unique efficient anonymization system, named PTA, that may both minimize the computational complexity of the anonymization process and anonymize transactional data with minimal information loss. The PTA system consists of three modules: a pre-processing module, a traveling salesman problem module, and an anonymization module. These modules anonymize transactional data and ensure that at least k-anonymity is attained. Numerous tests have been conducted to evaluate the effectiveness of the developed method in terms of scalability, runtime, and information loss against the most advanced anonymization methods. The suggested PTA method performs better than the compared algorithms overall, according to the results. They have unveiled PTA, a cutting-edge anonymization solution made up of three modules. In comparison to the most advanced algorithms for anonymizing transactional data, it anonymized the data with minimal information loss and very quickly. Extensive experiments have been conducted to evaluate the runtime and information loss performance of the designed system with the state-of-the-art transactional data anonymization algorithms.

[5], This paper's primary goal is to discuss the most recent techniques for virtualization-based resource allocation inside cloud computing. Additionally, a methodical analysis is

done on the role that virtualization plays in effectively supplying the clients' resources. Their research of the suggested ways shows that the requests' execution time and virtual machine availability are the primary problems that need to be taken into account in any resource allocation strategy. After outlining each of the fifteen studies that were viewed, it is suggested that the energy-saving resource allocation algorithms used by DSJF and DFCFS rely the most on the CloudSim simulator tool. Furthermore, the findings of the linked research showed that the DSJF algorithm outperforms DFCFS in terms of energy efficiency, with a potential 55% reduction in energy usage. This study reviews various existing resource allocation and scheduling algorithms for virtual machines. When it comes to making use of the cloud's shared resources, each scheduling strategy and VM allocation technique is essential. Their research indicates that the most important variables to take into account in any efficient resource allocation algorithm are the execution time of requests and the availability of virtual machine resources. The study also showed that the virtualization strategy significantly improves network speed, reduces PM costs in cloud data centers, balances load, saves energy on servers, and allocates resources in an active manner to meet client needs.

[6], The time synchronization issues for Internet of Things deployments involving apps that need a consistent sense of time are examined in this article. The clock model and other clock relation models are given in detail. Additionally, the clock synchronization techniques for various models are shown, along with a derivation and illustration of their anticipated performance. In order to help Internet of Things practitioners choose the right components for a deployment, an overview of time synchronization methods is offered. The time synchronization techniques are condensed into a survey, and the clock discipline algorithms are given in the style of a tutorial. As such, this article provides a comprehensive overview of the various time synchronization techniques that are accessible for Internet of Things deployments. The most significant finding from the time synchronization components that have been provided is that many components can be integrated to create an ideal deployment strategy; yet, a single solution cannot address every issue. There are multiple alternatives for every component in the comprehensive time synchronization methods that are offered. Still, there are a number of unresolved scientific issues. It is necessary to establish the time synchronization criteria for emerging network-calibrated clocks for ultra-low power ambient back-scattering communication devices and crystal-free radios. Secure communication over loosely synchronized networks of growing ultra-low-power entities demands further attention from the community, even if the security challenges of personal area networks of resource-constrained devices are active research fields. The components that are being described presuppose that the Internet of Things deployments have nearly autonomous wide-area, local area, and personal area networks. However, there is still a difficulty with the smooth integration of these networks' time synchronization capabilities. For heterogeneous IoT installations, in particular, expanding the precision time protocol for local area networks to personal area devices is an appealing choice. More research should be done on this and related solutions that allow software to be reused across various Internet of Things devices, and bottlenecks should be found.

[7], An evaluation methodology for applicable algorithms for resource management in cloud, fog, and edge computing is presented in this research. It offers perceptions into the state-of-

the-art at the moment and pinpoints specialized solutions made to meet certain user requirements. The study examines sixteen distinct resource management approaches and creates a taxonomy for useful assessment. It also offers a classification of the features provided by the algorithms and talks about how well-suited certain algorithms are for particular solution paradigms. This study aims to offer a methodology for evaluating and classifying various resource management approaches that can be used in edge and cloud/fog environments. Cloud, fog, and edge architects can benefit from having a clear illustration of the different resource management difficulties. A paradigm governed by cloud, fog, and edge computing may be able to help Internet of Things applications that are delay-sensitive. In addition, fog nodes typically process more data and have larger repository capacities, which can be exploited to enhance performance and lower latency and communication costs. In this work, they investigate algorithms that fall into six categories, which allows them to assess the state-of-the-art algorithms utilized in several research articles. After that, they discuss how to manage resources across edge, fog, and cloud devices.

[8], Two optimization algorithms have been developed in this research to determine the ideal DG's size and location in relation to power losses. Ant Colony Algorithm (ACO) and Genetic Algorithm (GA) are these algorithms. It is possible to tackle both large-scale linear and nonlinear problems with the proposed algorithms. GA and ACO, two heuristic techniques, are presented in this study. Depending on power losses, the recommended algorithms are utilized to determine the ideal DG's size and location. The suggested methods are verified by looking at IEEE-57 bus systems and comparing the results. The results are arranged and a graphic representation of the voltage improvement is provided. One can observe a decrease in active power losses and an improvement in the voltage profile. The outcomes showed how extremely feasible it is to apply DG based on ACO to reduce total losses of P and voltage when comparing voltage using a genetic algorithm.

[9], This study suggests optimizing the data recovery procedure and redesigning the log entry format. Through experimental simulation, the improved Paxos-based consistency algorithm's efficacy is confirmed. It is necessary to implement a distributed consistency algorithm to guarantee high availability and consistency throughout the system. Traditional Paxos-based distributed systems permit gaps in the log to enhance the performance of unstable networks. But these systems usually need a lot of network activity to fill in the gaps in the log when a node enters a recovery state; this significantly lengthens the duration for node recovery and consequently impacts system availability.

[10], They created the Nezha protocol, which offers cloud tenants without access to these technologies' excellent performance. When building Nezha, we began by noting that optimism is a typical strategy for enhancing consensus protocols: in an optimistic protocol, there is a backup slow way with a larger latency and a common-case fast path with low client latency. They begin by utilizing the multi-Paxoscode to implement and optimize Raft (Raft-2). We adjust the batch sizes of Raft-2 and Nezha to achieve optimal throughput. According to their analysis, Nezha performs better than Raft in both the open-loop and closed-loop tests. Additionally, they observe that disk writes—rather than message delays—now dominate latencies in Nezha, meaning that there is minimal change in latency with or without a proxy.

[11], The "happened before" relation defines a partial ordering, which is discussed in this paper along with a distributed technique for expanding it to a consistent total ordering of all the events. This algorithm may offer a practical method for putting a distributed system into action. They employ a straightforward technique to resolve synchronization issues to demonstrate how to use it. If the user perceives a different ordering than the one that this algorithm has produced, unexpected or unusual behavior may arise. The introduction of actual, physical clocks can prevent this. They derive an upper bound on the amount of time that these clocks can drift out of synchrony and offer a straightforward technique for synchronizing them. The literature-described techniques can be used to estimate message delays ktm and, in the case of clocks that allow for such adjustments, to modify clock frequencies dCi/dt. We think this theorem is a new result, though, as the condition that clocks are never set backwards appears to set our case apart from others that have been addressed before.

3. METHODOLOGY

In this section, the methodology will explained, Figure 1, shows the proposed methodology :

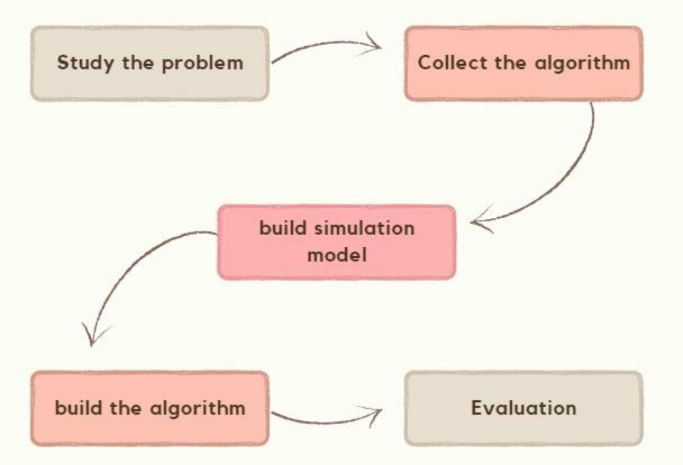


Figure 3.1 : proposed methodology

In the first step, it takes time to understand CPU scheduling and synchronization, seek out their knowledge, and try to apply the principles to single systems. After that, move on to distributing systems and understand the relation between this knowledge and the distributed systems. After that, we determine 10 algorithms that will be mentioned in the next section, study these algorithms, and give a brief overview of each one. In the third step, build the simulation model. A simulation model is a representation of the system that allows us to emulate and analyze the behavior of the system under various conditions without implementing it in a physical environment. This simulation model aims to provide insights into the performance, scalability, and behavior of the distributed system. Nodes represent individual computing units or devices in the distributed system. Resources, such as CPU, memory, and storage, are allocated and managed by the system. Tasks or Processes: Simulating the execution of applications or tasks that run on the distributed nodes. After that, the selection algorithm was built to be compatible with our simulation system model.

The last step is the evaluation, make evaluation between the relational algorithms and determine which algorithm are better in some cases.

ALGORITHMS

First, CPU scheduling algorithms:

First-Come, First-Served Scheduling

One of the easiest and most direct CPU scheduling techniques in operating systems is First, First-Served (FCFS). The procedure that comes first in FCFS scheduling is the one that gets started first. The order of arrival determines the schedule, and once a procedure begins, it does not stop until it is finished.

The way the First-Come, First-Served (FCFS) scheduling method works is that it starts processes in the ready queue and goes through them in that order. Processes are pushed to the back of the queue upon arrival, and the CPU is allotted to the first process. The selected procedure is executed continuously until it is finished. Because FCFS is a non-preemptive algorithm, once a process begins running, it does not stop until it is finished, and subsequent processes must wait. The convoy effect, in which shorter processes are delayed behind longer ones, may affect FCFS despite its simplicity and minimal scheduling cost. This might result in suboptimal system performance and fairness problems.

In FCFS, the algorithm runs on a simple FIFO basis, with the order of arrival dictating the execution sequence. Although FCFS's simplicity makes it easier to implement and comprehend, it might not be the best option in situations where process burst durations fluctuate, which could lead to long average waiting times. The algorithm is a fundamental idea in CPU scheduling that forms the basis for comprehending more intricate scheduling schemes present in contemporary operating systems [18]. Figure 2 show how the algorithm run.

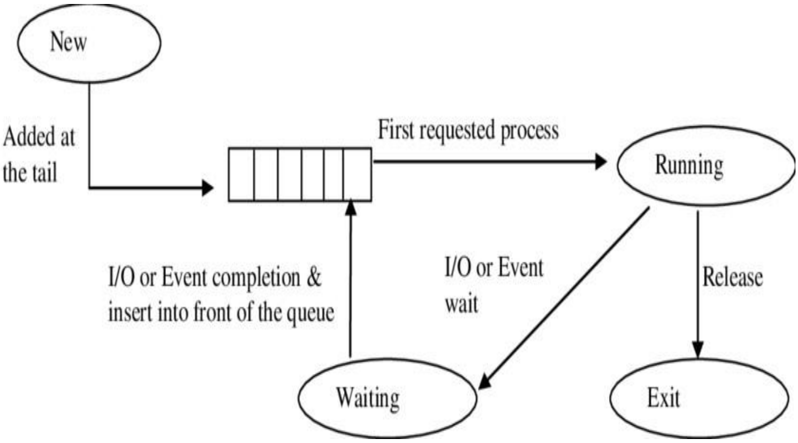


Figure 3.1: First come, First Serve scheduling.

Response times from FCFS might not always be the greatest, particularly if lengthy procedures come first. Wait times for shorter operations might cause problems with fairness.

Fair Share Scheduling

In an operating system with multiple users or tasks, Fair Share Scheduling is a CPU scheduling technique that makes sure that computer resources are distributed fairly among various users or groups. Fair share scheduling aims to maintain equity by preventing any one user or group from controlling a disproportionate amount of system resources and allocating a proportionate amount of CPU time to each user or group.

According to their allocated "shares," users or groups are allocated CPU time using the CPU scheduling method known as fair share scheduling. A set number of shares are assigned to each user or organization, signifying their right to access system resources. A larger percentage of CPU time is indicated by higher shares, which also result in higher priorities. Fixed periods are used for the allocation, and at the start of each period, priorities are dynamically modified based on shares. Processes that have not used up their entire share of resources may age in priority over time, progressively raising it to avoid resource hunger.[19]

A variety of computer scenarios, such as time-sharing networks, cloud computing, web hosting, educational settings, and multimedia applications, use fair share scheduling. It keeps the distribution of resources among the many hosted apps in server farms and web hosting equitable.

The Earliest Deadline First (EDF) scheduling algorithm

For managing the execution of activities with related deadlines, operating systems, and embedded systems employ the Earliest Deadline First (EDF) scheduling algorithm, a real-time scheduling technique. The main objective of EDF is to plan tasks such that they are completed by the deadline and that the probability of a missed deadline is as low as possible.

A new task is assigned a priority based on when it must be completed. Higher priority is assigned to tasks with earlier deadlines, The task that has the earliest deadline and the highest priority is chosen to be completed. Other parameters, such as task ID or arrival time, could be considered if several tasks have the same earliest deadline [20].

As an algorithm for preemptive scheduling, EDF allows a higher-priority job to override a lower-priority task's execution. Preemption is the situation where a new assignment is assigned that has a deadline that is closer than the ongoing one.

The chosen task keeps running until it is finished or until a higher-priority task comes up. The task that is presently running gets preempted and replaced if a higher-priority activity comes up.

The following Figure 3 displays the Earliest Deadline First Scheduling.

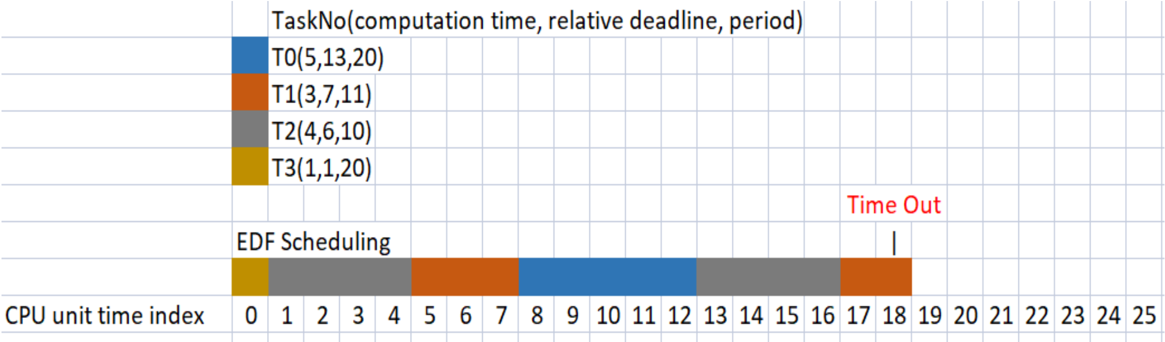


Figure 3.2: EDF Scheduling

The selected task continues to run until it is completed or until a higher-priority task becomes available. The current job is preempted and replaced if a higher-priority activity arises. The program keeps track of task deadlines continuously. In real-time systems, a task that finishes after its deadline is deemed to have been missed, and this might have negative effects [20].

The best thing about EDF is that it will discover a workable timetable if one exists. plus it works well with systems whose workloads are erratic and dynamic.

Proportional share Algorithms:

To allocate resources among competing entities in a way that represents their stated shares, proportional share algorithms are used in operating systems and resource management. The intention is to allocate resources proportionately to each entity so that each receives a fair share.

A weight, or portion, is given to each entity or job vying for resources. About other entities, this weight indicates the percentage of resources that each one should get. Resources are allocated more fairly to those with greater weights.

Based on their weights or shares, the scheduler regularly verifies which entities are qualified to receive resources. The algorithm being used and the kind of resources being assigned determine how frequently these checks are conducted.

Resources are allocated to entities in a way that is exactly proportionate to the weights or shares that have been allotted to them. The method ensures that entities with higher weights will receive a larger share of resources, while entities with lower weights will receive a lesser share.

Numerous proportionate sharing algorithms dynamically modify an entity's entitlements or priority according to its past resource consumption, behavior, and other considerations. The system may change dynamically to shift priorities and workloads.

The scheduler reassesses which entities should get resources next once entities have used the resources they allotted or finished their duties. If competing entities need resources, this cycle will continue [21].

Proportional sharing algorithms are primarily used to enforce fairness by guaranteeing that each entity will eventually receive its fair share of resources. To promote fairness, resource allocation can be balanced by different means, such as choosing entities with fewer entitlements more frequently or dynamically modifying priority.

Synchronization algorithms:

Chandy-Misra-Haas (CMH) algorithm

A distributed approach called the Chandy-Misra-Haas (CMH) algorithm is used to take a consistent worldwide snapshot of a distributed system. The approach is used to guarantee a synchronized view of the system's state, analyze distributed application behavior, and troubleshoot. It works by using unique messages known as markers that spread throughout the distributed system's processes. Every process logs its local state, including its present state and the status of incoming messages, upon receiving a marker. By offering a picture of the system that may have existed at a certain point in time, the technique guarantees that the global snapshot that is captured is consistent. When diagnosing problems, confirming the accuracy of distributed algorithms, or otherwise requiring a grasp of the distributed state, CMH is very helpful.

As a result, the CMH technique is frequently used to take a reliable global snapshot of a distributed system that includes the current state of messages that are routed between processes as well as the local state of every process. [14]

Marker messages are started at the start of the process by a specific initiator process. In the distributed system, the initiator broadcasts markers to every other process. Every process logs its current local state after receiving a marker. This covers both the process's present status and the status of incoming communications. A process records its local state and then tags incoming messages to tell future communications on these channels to be captured as well. Then The process distributes the marker to the processes that are close to it. Marker propagation and local state recording keep happening in this way, which cascades effects throughout the distributed system.

Every process receives a marker, which prompts it to record its current local state and mark incoming messages. Together, the local states that have been captured provide a consistent global picture of the distributed system. [15]

Figure 1 below illustrates the (CMH) method in a distributed system.

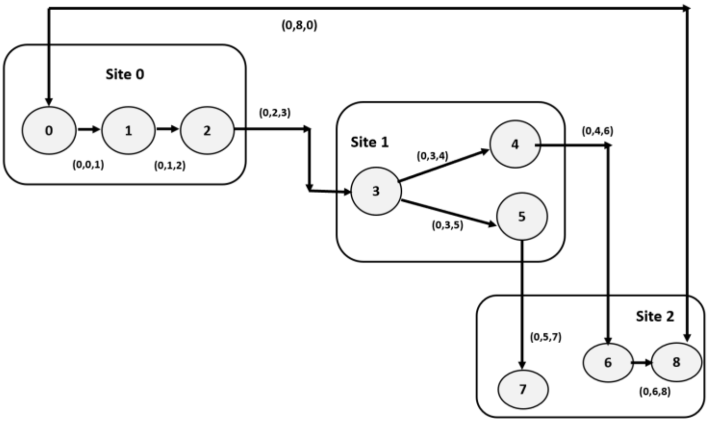


Figure 3.4: CMH algorithm

The Two-phase commit and Three-phase Commit Algorithm

In distributed systems, the Two-Phase Commit (2PC) and Three-Phase Commit (3PC) procedures are used to help a set of processes conclude. These methods are frequently used in distributed databases, transaction management, and distributed process coordinating settings.

during the tow phase, there must be a coordinator for this protocol. The coordinator receives a proposal from the customer with a value. The coordinator then attempts, in two stages, to bring a group of processes to consensus.

The coordinator gets in touch with every participant during the first phase, recommends values put forth by the customer, and then waits for their answer. Then the coordinator reviews all the replies and decides whether to commit if everyone agrees on the value or to stop if someone does not agree. The organizer gets in touch with every participant once more during the second phase to let them know whether to commit or cancel.[16] as we see in Figure 5.

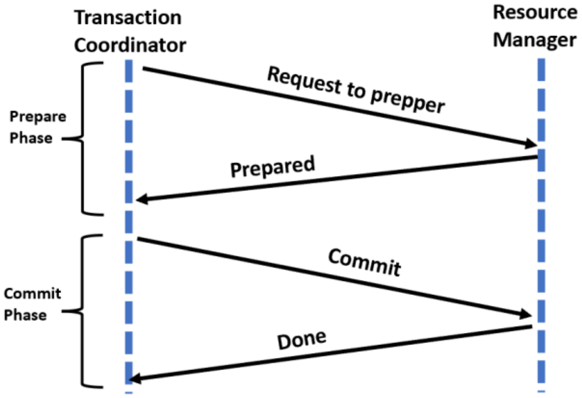


Figure 3.5: Two-phase commit

All the prerequisites listed above are satisfied. Because the participants do not suggest values and instead just vote yes or no to the values the coordinator suggests, there is an agreement. Because the coordinator forces each participant to make the same decision about whether to commit or abort, there is validity. The coordinator will only be certain of termination if all participants relay their answers. But mistakes might happen with this.

The Three Phase Commit is an expansion of the two-phase commit method, which divides the commit step into the following two stages.

When all participants in the first phase of the Two-Phase Commit (2PC) say "yes" the coordinator moves on to the "Prepare to Commit" step. During this stage, each player receives locks without carrying out the real commitment. When everyone replies “yes” at the "Get Ready to Commit" stage, the coordinator then gives the go-ahead for everyone to carry out the last commitment.

When a participant or both the coordinator and participant node fail during the commit phase, the pre-commit phase aids in our recovery. The recovery coordinator finds the new pre-commit useful in the following ways when it assumes control following coordinator failure during phase 2 of the preceding two PCs. If it discovers through participant queries that certain nodes are in the commit phase, it presumes that the commit decision was taken by the coordinator who was in place before to the crash. As a result, it can guide the protocol toward commit. Likewise, if a participant reports that they did not get prepared to commit, the new coordinator may conclude that the previous coordinator was ineffective even before initiating the prepare to commit phase [17].

show in Figure 6

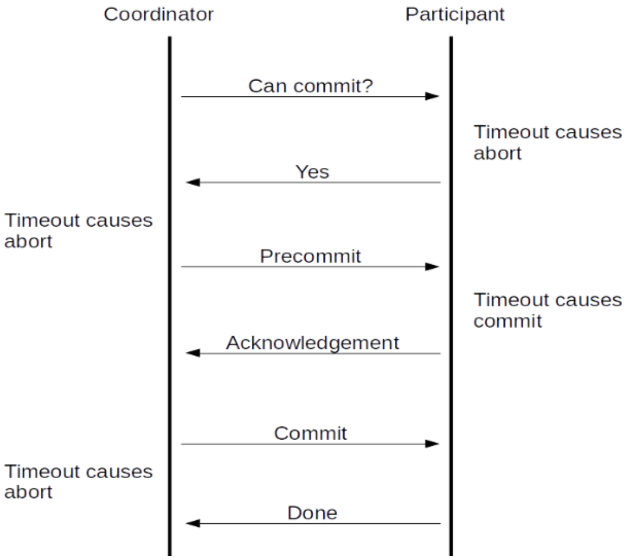


Figure 3.6 : Three-phase commit

2-phase commit and 3-phase commit are two closely related approaches. Table 3.1 provides a comparison between each of them.

Table1 3.1: Comparison between 2Phase commit VS 3Phase Commit

Features	2 Phase commit	3 phase commit
Blocking	blocking on participant or coordinator failure in the decision phase	reduces the possibility of extended blockage by adding a third step to avoid this danger.
Fault Tolerance	Both algorithms operate under the all-or-nothing premise. Participants are left with no choice except to rely on logging and recovery methods if the coordinator fails.	
Complexity	Simpler	complex

Raft algorithm

A consensus method called the Raft algorithm was created for distributed systems, in which several computers, or nodes, must come to an understanding of a shared state despite potential delays or failures. Raft, which Diego Ongaro and John Ousterhout first presented, is a more understandable approach for reaching agreement in distributed contexts than its predecessors. It uses a leader-follower model, in which data consistency and coordinated decision-making are ensured by having one node act as the leader while others follow its lead. Raft is frequently used when creating fault-tolerant distributed systems, including file and database systems, where node agreement is essential to the accuracy and dependability of the system [1].

For the Raft method to function, nodes in a distributed system must develop a leader-follower paradigm. as seen in Figure 7. It guarantees that, despite system malfunctions or modifications, every node agrees upon a consistent state.

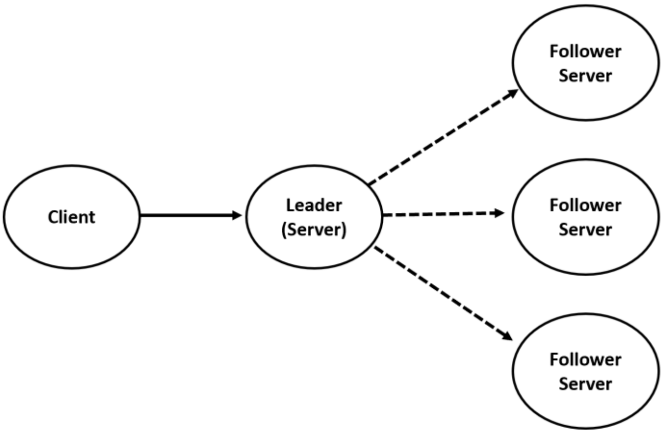


Figure 3.7: Raft Consensus Algorithm Components

1. Leader Election:

To begin, the Raft algorithm holds an election among the nodes to choose a leader. When a leader does not communicate, or when a node has a higher term number, which indicates a special period, that node becomes a candidate and calls for an election. The candidate then asks other nodes to cast their votes, and if it receives the majority, it takes over as leader for an additional period. After that, this leader oversees the replication of a log and makes sure that all the system's dispersed nodes are committed and consistent.

2. Log Replication:

In the Raft algorithm, the leader operates by first accepting orders from clients, adding these commands to its log, and then sending the log entries to followers so that they can replicate. This procedure makes sure that the dispersed nodes keep a consistent log, which helps the system to agree and coordinate.

3. Consistency and Commitment:

Only after a command is replicated by most nodes is it deemed committed. After committing, the leader notifies the followers, who then implement the directive into their state machines.

4. Heartbeats

To keep its position as the leader and make sure other nodes are responding, it periodically broadcasts heartbeats.

5. Handling Failures:

The same procedure is used to elect a new leader in the event of unavailability. To ensure consistency, the uncommitted entries in the log are removed [2].

The Paxos algorithm

A consensus technique called the Paxos algorithm is used to reach a consensus among a collection of dispersed processes, or nodes, in a distributed system. In 1989, Leslie Lamport, a computer scientist, presented it in a research paper titled "The Part-Time Parliament." In distributed systems with failures and communication delays between nodes, the Paxos method is frequently employed to guarantee consensus [1].

The main objective of the Paxos algorithm is to enable a collection of nodes to reach a consensus on a single value, regardless of the possibility of some nodes failing or of communications between nodes being lost or delayed. Its architecture considers different failure scenarios and network circumstances, and it functions in a decentralized way.

Important stages and ideas in algorithms

In the following section, Figure 8. displays the Paxos Algorithm

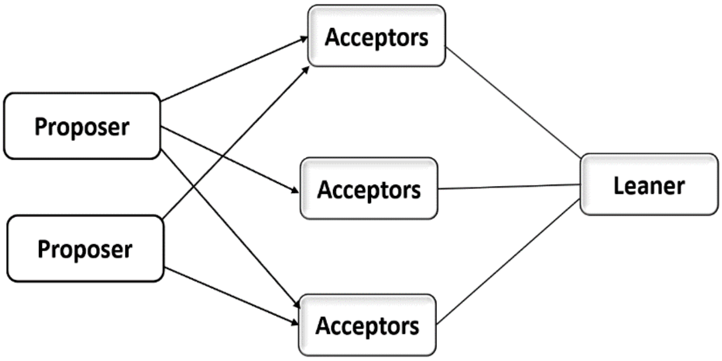


Figure 3.8: displays the Paxos Algorithm

- Proposal (Prepare) Stage:

All other nodes get a prepared message from a node that wants to suggest a value as the agreed value. The proposal number that is included in this mail serves as a special identification for this specific proposal. As a response to the prepared message, every node promises not to accept any proposals whose proposal number is less than the one in the prepare message.

- Accept Stage:

After a node obtains commitments from most other nodes, it can send an accept message with the proposal number and proposed value to all other nodes. The suggested value will only be accepted by each node that gets this accept message if it has not previously committed to accepting a proposal with a greater number.

- Learn Stage:

A node can send a commit message with the agreed-upon value to all nodes after receiving accept messages from the majority of nodes. This agreed-upon value will then be updated in the states of the other nodes.

The Paxos algorithm, well-known for its capacity to reach consensus in distributed systems, finds use in several contexts where obtaining node agreement is essential to the accuracy and dependability of the system, such as in Ensuring that file operations and metadata changes are agreed upon uniformly throughout the system, distributed file systems use Paxos to coordinate activities among dispersed nodes. In addition, agreement algorithms equivalent to Paxos are employed by certain blockchain implementations and distributed ledger technologies to reach an agreement on the sequence and legitimacy of transactions throughout the network [2].

ANTI COLONY ALGORITHM

Ant Colony Optimization (ACO) draws inspiration from the foraging behavior of real ants. In nature, ants collectively find the shortest path between their nest and a food source by depositing and following a chemical substance called pheromone. This decentralized approach enables ants to discover and optimize paths over time. ACO translates this behavior into an algorithmic framework for solving optimization problems[23].

Synchronization in distributed systems often requires reaching a consensus among nodes. ACO, being an optimization algorithm, doesn't inherently provide mechanisms for reaching a consensus.

ACO is designed for relatively static optimization problems. Distributed systems, especially in real-world scenarios, often face dynamic changes, failures, and varying workloads, which may not be well-handled by ACO.

ACO is a powerful algorithm for solving optimization problems but is not tailored for synchronization in distributed systems. Synchronization in distributed systems typically involves protocols like Two-Phase Commit, Paxos, Raft, or other consensus algorithms designed to handle the complexities of distributed environments, failure scenarios, and the need for strong consistency guarantees.

4. RESULTS AND DISCUSSIONS

Simulation model:

The simulation employs a system conforming of six bumps, with each knot containing two active coffers and three unresistant coffers. Due to the simulation's demanding CPU and memory conditions, a larger set of bumps and fresh coffers weren't explored. Each knot is associated with a periodic task featuring a 2000- unit period and a 400- unit calculation time, exercising all available coffers. Nonperiodic tasks, unless specified else, follow normal distributions for both calculation time and laxity. The mean calculation time and laxity for tasks are 400- and 600- time units, independently, with standard diversions of 100 and 300, independently.

Non-recurring tasks are incorporated using a Poisson process, resulting in varying workloads attributed to differences in appearance rates among bumps. The system-wide non-recurring task appearance rate (denoted as R) is introduced, reflecting the cumulative occurrence rates of non-recurring tasks across all bumps. To standardize the appearance rate, the task appearance rate is defined as the average number of occurrences per 400 time units (aligning with the mean task computation time of 400 units). This normalization ensures flexibility in adhering to the specific computation time value. Despite the selection of 600 as the nominal mean laxity value, the influence of different laxity values is also investigated.

Each irregular assignment necessitates a minimum of one operative asset and can utilize none or more compliant repositories, subject to randomly determined resource circumstances. Specifically, there is a 2/3 likelihood that an irregular assignment will request each of the two active repositories, and a 0.5 likelihood of requesting each compliant resource. Although the selection of probabilities is arbitrary, it significantly influences the system load, denoted as L.

For a given system irregular task arrival rate R, the system load (L) is computed as the mean load on the active repositories using the formula L = (2/3) (R/n) ^0.20, where n is the count of protrusions. The initial term signifies the average load on an active asset of a protrusion due to irregular tasks, while the latter term accommodates periodic tasks, which consume 20 percent of the computational power on each protrusion.

The simulated framework presupposes the execution of both the worldwide scheduler and the primary scheduler on a dedicated coprocessor expressly crafted for scheduling and additional system support duties. This coprocessor functions to shield operational tasks from external disruptions and disturbances induced by the execution of kernel modules, ensuring predictable system performance. Due to the specialized requirements of real-time systems, often entailing dedicated hardware for activities such as processing data from sensors, relying on a specialized technical coprocessor is sustainable, particularly given the novel advantages it offers.

The existence of the coprocessor doesn't eliminate the time delay linked to ensuring a task. These delays, proportional to the magnitude of the task set recycled by the assurance routine (20), are duly taken into account in the simulation analyses. The proportionality constant is established at 2, signifying that if the maximum number of tasks recycled by the assurance routine at any given moment is 5 (as in our simulation scripts), a task experiences a worst-case assurance overhead slightly surpassing 10 percent of the average computation time of a task.

Communication among nodes is facilitated through a hub communication network, where each link corresponds to a specific node. One terminus of a link is linked to the associated node, while all the other ends of the links are interconnected. Messages sent by a node to another traverse the sender's link first and then the receiver's link. The messages move through links in a first-come, first-served manner, allowing only one communication to occupy a link at any given time. However, subsequent messages must wait until the link becomes available if a link is currently in use. The overall time required for transmitting a message from one node to another without any delays is referred to as the conflict-free communication duration (MD). Since a message traverses two links, the time taken for a message to traverse a single link is half of its communication duration.

Latencies in communication are also taken into account when relocating tasks in real-time systems. Given that the set of tasks executing in the system is pre-known, it is viable to save the task code at one or more nodes, specifically those equipped with the necessary resources. When a node wishes for another to execute a task, it must furnish the other node with details identifying the task and pertinent inputs. The time delay in transmitting a task to another node, for simulation study purposes, equals the message delay plus 10 percent of the computation time of the task.

Top of Form

To measure variations in node workloads within a distributed scheduling algorithm, the load distribution factor (designated as B) is introduced to signify nonperiodic task arrival rates across the six nodes. Assuming an identical sequence of arrival rates, if the load distribution factor is B and the initial node experiences an average arrival rate of N tasks per unit time, then the ith node's arrival rate is NB(i-1). For example, with B = 0.5 and N = 2, nodes 1 through 6 would have arrival rates of 2, 1, 0.5, 0.25, 0.125, and 0.0625, respectively. The equilibrium rate ranges between 0 and 1, with a higher value indicating a more evenly distributed workload among nodes.

The effectiveness of distributed scheduling algorithms is influenced by adjustable parameters, and the values assigned to these parameters must have a direct effect on the performance of the system. Although presentation of simulation studies is constrained by space, the findings suggest that the examined algorithms demonstrate stability. Here, stability denotes that there is minimal percentage variation in system performance resulting from a specific modification in parameter values. Essentially, this means that the system's performance remains relatively constant even when parameters deviate moderately from their "optimal" values, which are the values that yield optimal system performance for a specific application. In the In the simulation experiments, the adjustable parameters are configured as follows:

Top of Form an "optimal" window length (WL) is determined to be between 30 and 70 times the mean task interarrival time, with 50 times the mean task interarrival time chosen as the window length. Periodically, surplus data is transmitted to other nodes, and the duration of this transmission is also set to 50 times the average job interarrival duration. The limited number of nodes (6 nodes) ensures sufficient information dissemination to every node in the network. The simulation model incorporates The traffic of messages arising from the transmission of excess information, along with all other messages generated during the scheduling process, is considered. Systemwide Guarantee Surplus (SGS) is established at 2.0, Focused Addressing Surplus (FAS) at 1.2, Minimum Bid (MB) at 1.0, and High Bid (HB) at 2.0. Each data point in the simulation outcomes, presented in the subsequent sections, reflects the average of three runs, each lasting (at least) 3000 times the mean task interarrival time. The 95 percent confidence intervals for these data points are within 2 percent. Before delving into the simulation results, let's clarify the subsequent terms to characterize the system's performance. As the primary objective of any hard real-time system is to meet task deadlines, the guaranteed ratio emerges as a significant metric for assessing scheduling algorithm effectiveness.

Top of Form

Table 4.1 show the results of the CPU scheduling algorithms on variant cases:

Table 4.1: Results of the Four CPU Scheduling Algorithms

High algorithm	DATASET	Performance
EDF algorithm’s give the high performance	three processes in three different cases with random burst.	Improve the efficiency of EDF algorithm by changing the idea of fixed time quantum to dynamic calculated automatically without the interference of user.
EDF algorithm better efficient	Five workload system file systems, disk systems, and user preferences	It outperforms disk schedulers by as much as 15.8 precent while consuming less than 3 percent-5 precent of another algorithms.

EDF algorithm give better efficient	5 scheduling approaches, one-step scheduling, predicted mean interval scheduling, conservative scheduling, history mean scheduling, history of conservative scheduling.	EDF algorithm able to achieve efficient execution of data-parallel applications even in heterogeneous and dynamic environments.

On the above simulation model, the results in Table 4.1 show that the EDF algorithm gives more efficient results compared to other algorithms in different cases, and Figure 8 summarizes the results as follows:

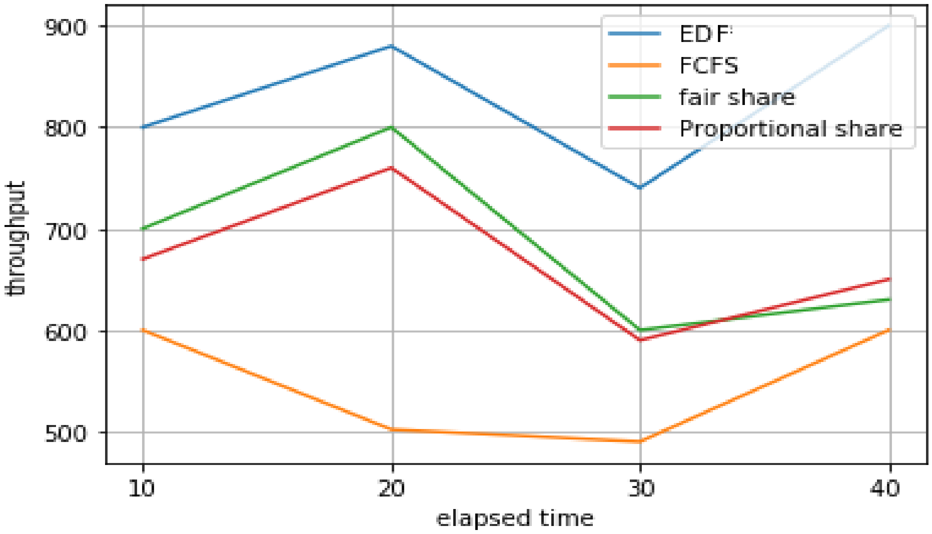
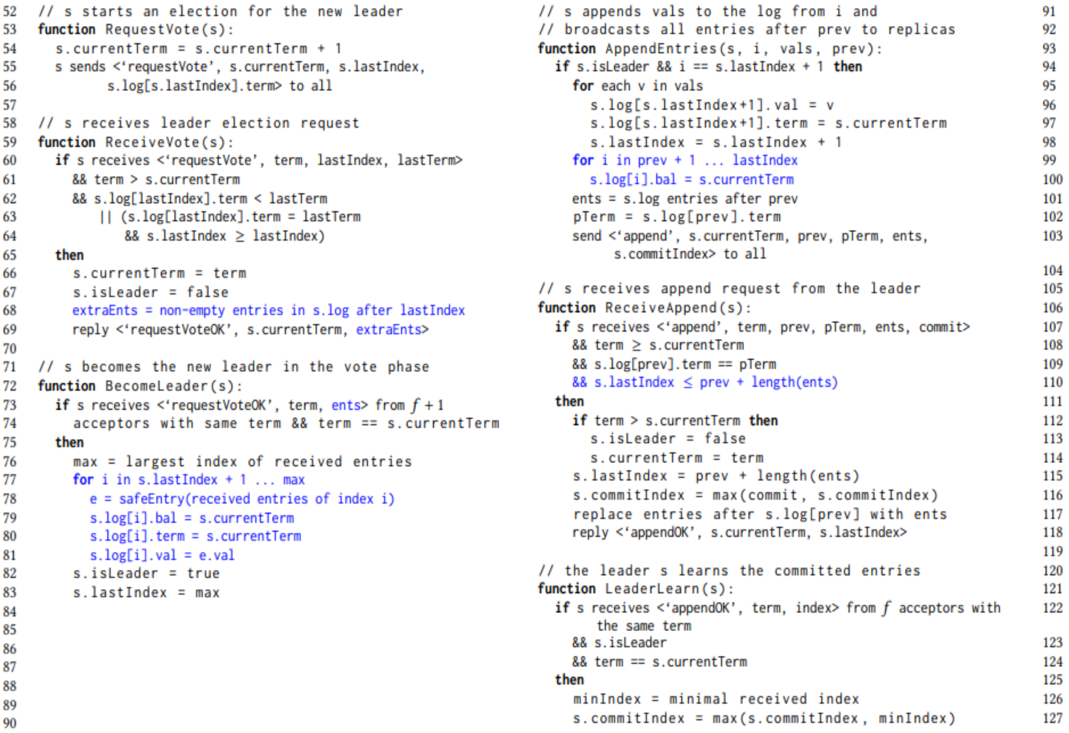


Figure 4.1: CPU SCHEDULING RESULTS

In synchronization algorithms, Raft ensures that if two logs include the same operation, they will possess identical indices and terms. This implies that each operation receives a distinct index. In the context of Paxos, when multiple servers become candidates simultaneously, the candidate with the higher term emerges as the winner in the election. Conversely, in Raft, the simultaneous candidacy of multiple servers may result in a vote split as they share the same term, preventing either from winning the election. Raft addresses this issue by having followers wait an additional period, randomly chosen from a uniform distribution, after the election timeout. Consequently, Raft is anticipated to exhibit both a slower pace and greater variability in the time required to elect a leader.



```
// s starts an election for the new leader
function RequestVote(s):
  s.currentTerm = s.currentTerm + 1
  s sends <'requestVote', s.currentTerm, s.lastIndex,
        s.log[s.lastIndex].term> to all

// s receives leader election request
function ReceiveVote(s):
  if s receives <'requestVote', term, lastIndex, lastTerm>
    && term > s.currentTerm
    && s.log[lastIndex].term < lastTerm
       || (s.log[lastIndex].term = lastTerm
          && s.lastIndex ≥ lastIndex)
  then
    s.currentTerm = term
    s.isLeader = false
    extraEnts = non-empty entries in s.log after lastIndex
    reply <'requestVoteOK', s.currentTerm, extraEnts>

// s becomes the new leader in the vote phase
function BecomeLeader(s):
  if s receives <'requestVoteOK', term, ents> from f + 1
    acceptors with same term && term == s.currentTerm
  then
    max = largest index of received entries
    for i in s.lastIndex + 1 ... max
      e = safeEntry(received entries of index i)
      s.log[i].bal = s.currentTerm
      s.log[i].term = s.currentTerm
      s.log[i].val = e.val
    s.isLeader = true
    s.lastIndex = max
```

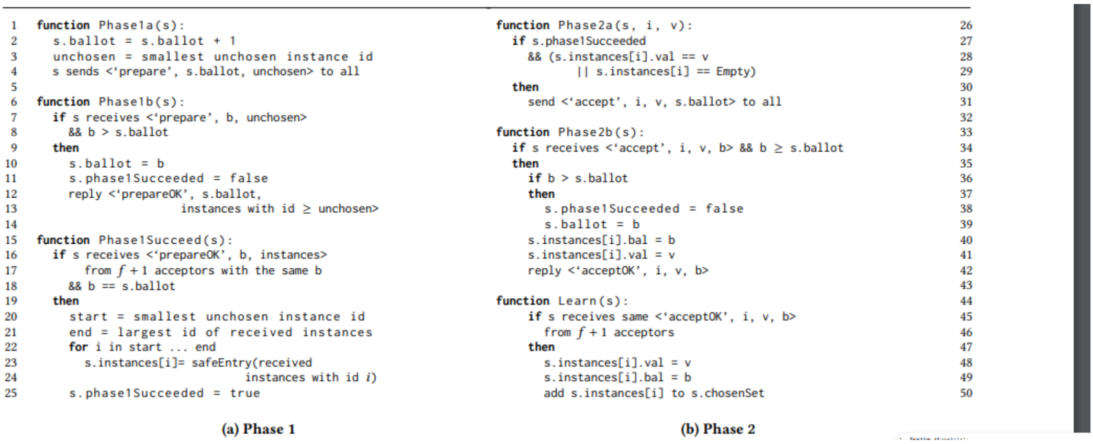
(a) Phase 1

```
// s appends vals to the log from i and
// broadcasts all entries after prev to replicas
function AppendEntries(s, i, vals, prev):
  if s.isLeader && i == s.lastIndex + 1 then
    for each v in vals
      s.log[s.lastIndex+1].val = v
      s.log[s.lastIndex+1].term = s.currentTerm
      s.lastIndex = s.lastIndex + 1
    for i in prev + 1 ... lastIndex
      s.log[i].bal = s.currentTerm
    ents = s.log entries after prev
    pTerm = s.log[prev].term
    send <'append', s.currentTerm, prev, pTerm, ents,
         s.commitIndex> to all

// s receives append request from the leader
function ReceiveAppend(s):
  if s receives <'append', term, prev, pTerm, ents, commit>
    && term ≥ s.currentTerm
    && s.log[prev].term == pTerm
    && s.lastIndex ≤ prev + length(ents)
  then
    if term > s.currentTerm then
      s.isLeader = false
      s.currentTerm = term
    s.lastIndex = prev + length(ents)
    s.commitIndex = max(commit, s.commitIndex)
    replace entries after s.log[prev] with ents
    reply <'appendOK', s.currentTerm, s.lastIndex>

// the leader s learns the committed entries
function LeaderLearn(s):
  if s receives <'appendOK', term, index> from f acceptors with
       the same term
    && s.isLeader
    && term == s.currentTerm
  then
    minIndex = minimal received index
    s.commitIndex = max(s.commitIndex, minIndex)
```

(b) Phase 2

Figure 4.2: Raft

However, Raft's leader election phase is more lightweight than Paxos'. Raft only allows a candidate with an up-to-date log to become a leader and thus need not send log entries during leader election .



```
function Phase1a(s):
  s.ballot = s.ballot + 1
  unchosen = smallest unchosen instance id
  s sends <'prepare', s.ballot, unchosen> to all

function Phase1b(s):
  if s receives <'prepare', b, unchosen>
    && b > s.ballot
  then
    s.ballot = b
    s.phase1Succeeded = false
    reply <'prepareOK', s.ballot,
                  instances with id ≥ unchosen>

function Phase1Succeed(s):
  if s receives <'prepareOK', b, instances>
      from f + 1 acceptors with the same b
    && b == s.ballot
  then
    start = smallest unchosen instance id
    end = largest id of received instances
    for i in start ... end
      s.instances[i]= safeEntry(received
                          instances with id i)
    s.phase1Succeeded = true
```

(a) Phase 1

```
function Phase2a(s, i, v):
  if s.phase1Succeeded
    && (s.instances[i].val == v
          || s.instances[i] == Empty)
  then
    send <'accept', i, v, s.ballot> to all

function Phase2b(s):
  if s receives <'accept', i, v, b> && b ≥ s.ballot
  then
    if b > s.ballot
    then
      s.phase1Succeeded = false
      s.ballot = b
    s.instances[i].bal = b
    s.instances[i].val = v
    reply <'acceptOK', i, v, b>

function Learn(s):
    if s receives same <'acceptOK', i, v, b>
      from f + 1 acceptors
    then
      s.instances[i].val = v
      s.instances[i].bal = b
      add s.instances[i] to s.chosenSet
```

(b) Phase 2

Figure 4.3: Multipaxos

When the message complexity is low, the CMH algorithm guarantees the best scenario, but the 2PC and 3PC are more efficient in transactions that may involve updates to multiple nodes or databases.

In scenarios where ensuring atomicity and consistency across distributed nodes is critical, such as financial transactions or inventory management.

The CMH algorithm differs in its focus and application, emphasizing snapshot collection and global state detection, while 2PC and 3PC are tailored for transactional consistency. CMH may be preferred in scenarios where obtaining a consistent snapshot is a priority, whereas 2PC and 3PC are more suited for distributed transactional operations. The choice between them depends on the specific requirements of the distributed system, considering factors such as blocking tolerance, simplicity, and ease of implementation.

5. CONCLOUSION

In this manuscript, an evaluation model was constructed to assess various CPU scheduling algorithms and synchronization algorithms within a distributed system. A simulation model was devised with six nodes, each equipped with two active resources and three passive resources. (Had it not been for the excessive CPU time and memory demands necessary for this simulation, we would have explored a more extensive set of nodes and additional resources.) Each node accommodates one periodic task with a period of 2000 time units and a computation time of 400 units, requiring all available resources. We implemented four CPU scheduling algorithms, and the outcomes indicated that the EDF algorithm exhibited superior performance, contingent on the synchronization algorithm. The CMH algorithm places emphasis on snapshot collection, whereas 2PC and 3PC are transactional. Raft ensures that if two logs contain the same operation, they will have identical index and term values, whereas in Paxos, if multiple servers become candidates simultaneously, the candidate with the higher term will prevail in the election.

References

1. Ali, S., Alshahrani, R., Hadadi, A., Alghamdi, T., Almuhsin, F., & Sharawy, E. E. (2021). A Review on the CPU Scheduling Algorithms: Comparative Study. Int. J. Comput. Sci. Netw. Secur, 21(1), 19-26.
2. Grammenos, A., Charalambous, T., & Kalyvianaki, E. (2023). CPU scheduling in data centers using asynchronous finite-time distributed coordination mechanisms. IEEE Transactions on Network Science and Engineering.
3. Bittencourt, L. F., Goldman, A., Madeira, E. R., da Fonseca, N. L., & Sakellariou, R. (2018). Scheduling in distributed systems: A cloud computing perspective. Computer science review, 30, 31-54.
4. Alsulami, A. A., Al-Haija, Q. A., Thanoon, M. I., & Mao, Q. (2019, April). Performance evaluation of dynamic round robin algorithms for CPU scheduling. In 2019 SoutheastCon (pp. 1-5). IEEE.
5. Shukur, H., Zeebaree, S., Zebari, R., Zeebaree, D., Ahmed, O., & Salih, A. (2020). Cloud computing virtualization of resources allocation for distributed systems. Journal of Applied Science and Technology Trends, 1(3), 98-105.
6. Yiğitler, Hüseyin, Behnam Badihi, and Riku Jäntti. "Overview of time synchronization for

IoT deployments: Clock discipline algorithms and protocols." Sensors 20.20 (2020): 5928.

7. Mijuskovic, A., Chiumento, A., Bemthuis, R., Aldea, A., & Havinga, P. (2021). Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification. Sensors, 21(5), 1832.
8. Zakaria, Y. Y., Swief, R. A., El-Amary, N. H., & Ibrahim, A. M. (2020). Optimal distributed generation allocation and sizing using genetic and ant colony algorithms. In Journal of Physics: Conference Series (Vol. 1447, No. 1, p. 012023). IOP Publishing.
9. Chao-fan, Z. H. U., Jin-wei, G. U. O., & Peng, C. A. I. (2019). Implementation and optimization of a distributed consistency algorithm based on Paxos. Journal of East China Normal University (Natural Science), 2019(5), 168.
10. Geng, J., Sivaraman, A., Prabhakar, B., & Rosenblum, M. (2022). Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. arXiv preprint arXiv:2206.03285.
11. Lamport, L. (2019). Time, clocks, and the ordering of events in a distributed system. In Concurrency: the Works of Leslie Lamport (pp. 179-196).
12. Themistokilis, Charalambous and Evangelia kalyvianaki, (2023). CPU Scheduling in Data Centers Using Asynchronous Finite-Time Distributed Coordination Mechanisms
13. Alexandra Olteanu, Florin Pop, Ciprian Dobre, Valentin Criste, (2012). A dynamic rescheduling algorithm for resource management in large scale dependable distributed systems
14. Mega Putra, Gaurav Karki, Mayank Verma, (March 2019). A Survey on Deadlock Detection Algorithms for Distributed Systems.
15. Sabir Hussain, Adeel Sajjad, and Zeeshan Javed, (2020). Deadlock Detection in Distributed System
16. Aman Kumar Pandey, Sarvesh Pandey, Udai Shanker, (2019). LIFT- A Linear Two-Phase Commit Protocol.
17. Mahmoud Y. Shamsa, Ahmed S. Abolabanb, Amr A. Abohanyb, (2022) A Review on Concurrency Control Techniques in Database Management Systems.
18. Ali A. AL-Bakhrani1, Abdulnaser A. Hagar2, Ahmed A. Hamoud3,(2020). Comparative Analysis of CPU Scheduling Algorithms: Simulation and Its Applications.
19. Ethan Bolker, (2020). ON THE PERFORMANCE IMPACT OF FAIR SHARE SCHEDULING
20. MarioG unzel, Jian-Jia Chen (2020) Suspension-Aware Earliest-Deadline-First Scheduling Analysis. 10.1109/TCAD.2020.3013095
21. Jason Nieh, Chris Vaill, Hua Zhong,(2001).Virtual-Time RounRobin: An O(1)Proportional Share Scheduler
22. Heidi Howard, Aleksey Charapko, (2021) Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos, https://doi.org/10.1145/3427796.3427815
23. Qaddara, I. A. A. R. (2022). APPLYING MACHINE LEARNING TECHNIQUES ON CYBER SECURITY DATASETS: DETECTING CYBER ATTACKS. Harbin Gongye Daxue Xuebao/Journal of Harbin Institute of Technology, 54(7), 95-110.