COLLEGE OF ENGINEERING AND TECHNOLOGY(COETEC)

SCHOOL OF ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING

B.Sc. TELECOMMUNICATION AND INFORMATION ENGINEERING

UNIT: DISTRIBUTED COMPUTING & APPLICATION

UNIT CODE: ICS 2403

GROUP 1

NO	NAME	REG NO
1	BRIDGETTE NTHUKA	ENE221-0097/2021
2	STANLEY OMONDI	ENE221-0140/2021
3	VICTOR OSIOLO	ENE221-0131/2021
4	CYNTHIA NJAMBI	ENE221-0117/2021
5	IRENE YEGON	ENE221-0128/2021
6	PEACE ISSA	ENE221-0132/2021

ALGORITHMS FOR DISTRIBUTED PROCESSING SYSTEMS

Introduction

Distributed processing systems let a number of linked computers, or nodes, work together to do large-scale or complicated calculations. These systems are very important in today's computing environments, such as cloud platforms, telecommunication networks, and large distributed databases. To function efficiently and reliably, they require specialized algorithms that control coordination, communication, data sharing, failure recovery, and job distribution among several devices.

This paper comprehensively analyzes the principal distributed algorithms and their significance in the creation of high-performance, fault-tolerant, and scalable systems. The primary objectives of the study are to comprehend the theoretical foundations of significant algorithm categories and evaluate their applicability to practical engineering contexts.

Python is used to create and test different algorithms to show how they operate when they are spread out. The simulations look at message passing, latency, node coordination, and overall performance in different situations.

Overview of Distributed Algorithms

There are several broad categories of distributed algorithms. Each category focuses on overcoming a fundamental challenge that arises when multiple machines collaborate.

1. **Coordination and Synchronization Algorithms.**

   Like a team working on a shared doc. In distributed systems, if two nodes try to update the same piece of data at the same time, the data can become corrupted (race condition). These ensure only one process uses a shared resource at a time.

   Synchronization- establish an agreed upon order

   Coordination- manage and control how nodes access shared resources.

   Algorithms are enabled so that different processes or threads operate simultaneously to avoid race conditions, deadlocks, and inconsistencies. They usually involve distributed locks, semaphores and distributed clocks.

   - Distributed Locks: Distributed locks are mechanisms used to coordinate access to shared resources across multiple nodes in a distributed system.
   - Semaphores: Semaphores are another synchronization primitive used to control access to shared resources, particularly in concurrent programming. They can be used to limit the number of concurrent accesses to a resource or to signal events between processes.
   - Distributed Clocks: Distributed clocks are used to maintain synchronized timestamps across multiple nodes in a distributed system.

Algorithms address issues such as:

- Clock Synchronization: Ensures that all nodes in a distributed system have a consistent view of time. Algorithms like the Network Time Protocol (NTP) and the Berkeley Algorithm are commonly used.
- Mutual Exclusion: Prevents multiple processes from accessing a shared resource simultaneously. Examples include Ricart-Agrawala and Lamport's algorithms.
- Leader Election: Identifies a single node as the leader to coordinate tasks. The Bully Algorithm and the Ring Algorithm are popular choices.

Examples:

a. Leader election Algorithms
   Choosing the node with the highest ID or priority to coordinate the work.
   1. Bully algorithm
      Election: X sends an ELECTION message to processes with a higher ID
      Higher ID responds: if a higher ID process exists, it send an OK message and X backs off
      No response: if no other higher ID responds after a timeout, X becomes the leader announcing itself by sending a COORDINATOR message.
   2. Ring Algorithm
      Works well where nodes are logically connected in a circle/ring
      Initiation: X creates an ELECTION message with its ID and sends it to its neighbor in the ring.
      Circulation: when a node receives the message, it adds its own ID to the list and forwards it to the next neighbor until it gets back to X
      Selection: X examines list of IDs and chooses the highest ID as the new leader and sends a final COORDINATOR message with the winning ID to all nodes around the ring.

b. Two-Phase Commit protocol (2PC)
   Its primary purpose is to ensure atomicity (the "all-or-nothing" rule) for a transaction that spans multiple, independent machines (nodes). It forces all participating nodes to either commit the transaction or abort (roll back) it together, preventing partial, inconsistent updates e.g., moving money between accounts.
   Involves:
   - Coordinator/Boss- starts process and makes final decision
   - Participants/Workers- other computers that must perform a small part of the transaction.

   Phases:
   a. Voting
      Boss: sends a PREPARE message to participants.
      Workers check: lock any resources they need and write changes to a temporary log but they are not final.

Workers reply: If a worker is successful, they reply YES (Ready to Commit). If a worker finds any problem, they reply NO (Abort).

b. Decision

Successful commit

If the coordinator receives YES from every participant, the COMMIT decision is made and the coordinator sends COMMIT messages to all participants who make temporary changes permanent and release their locks.

Abort/Rollback

If the coordinator receives **even one NO**, the decision is ABORT which is sent as a message to participants who undo all the temporary changes and release their locks.

Limitations

I. If the coordinator fails during the commit phase, it can block the entire working of the 2PC protocol.

II. After receiving votes from participants, the network partition before providing any decision (commit/abort) by the coordinator can leave the participants in an indefinite waiting state.

2. Consensus Algorithms

Nodes in a system need to agree on a single, permanent decision, e.g. Transaction is complete, otherwise the system breaks down or becomes inconsistent.

Consensus algorithms allow the different nodes distributed throughout them to agree on a single shared value or outcome in spite of individual node failures and disagreements among them (meaning despite the situations when one of the nodes failed or there were discrepancies among them)

Examples

Paxos and Raft - Algorithms define a process, often involving voting, where a Leader proposes a value and majority of Followers must accept it after which the value is considered final. They are crucial for systems like databases or blockchains that need to guarantee consistency even during failures.

Types:

a. Raft

Ensure majority of nodes agree on sequence of operations.

Leader: In Raft, a leader handles all the client's requests and disseminates them to the followers.

Follower: All other servers are considered followers apart from the leader. The followers receive direction from the leader and operate according to those requests.

Candidate: A server may transition from a follower to a candidate. A candidate is a potential leader appointed as a part of the election among the servers in the cluster.

Phases:

a. Leader election- a node becomes a candidate, increases the term number and sends RequestVote to other nodes. Other nodes vote for the first candidate it sees in a new term provided the candidate's log is up-to-date like its own. Candidate becomes the new leader if it receives votes from a majority of nodes
b. Log replication- all client requests go to the leader. Leader adds requests to its log and sends AppendEntries messages to all followers. A log entry is committed once the entry has been successfully stored on a majority of follower nodes.

Ensures entry will be present on the future leader guaranteeing the system never rolls back a committed change.

LIMITATIONS

I. Raft protocols depend on a single leader, which can become a performance bottleneck and can become a single point of failure.

II. Insufficient log compaction: This process can be resource-intensive and can lead to performance degradation during heavy write activities or large logs.

b. Paxos

There is no single leader. The three types of roles are proposers, acceptors, and learners.

Proposers: These nodes send their proposals to other nodes in a replica group. Each proposer aims to have a majority of votes for their proposals.

Acceptors: These nodes either accept or reject the proposed value from the proposers.

Learners: The acceptors inform these nodes of the accepted value chosen via a higher majority.

<u>STEPS</u>

- Initiation: In Paxos, a node, called the proposer, initiates a proposal by sending a "prepare" message to a majority of nodes (known as acceptors) in the system.
- Voting: Upon receiving the PREPARE message, each acceptor checks if it has promised to accept proposals with higher numbers. If not, it responds with a promise and may include any previously accepted proposal.

- Proposal Phase: The proposer collects promises from a majority of acceptors. It then sends a proposal with the highest numbered proposal among the promises to the acceptors.
- Acceptance: If the acceptors receive a proposal and have not made a promise to accept a proposal with a higher number, they accept the proposal and inform the learner.
- Consensus: Once a proposal is accepted by a majority of acceptors, consensus is reached, and the value proposed by the consensus becomes the chosen value for the system

LIMITATIONS

I. The Paxos requires several rounds of communication between servers to reach a consensus. This can result in reduced performance, which can cause high latency.
II. As the number of servers increases, the communication overhead and coordination complexity also increase, which limits the protocol's scalability

3. Scheduling Algorithms

Distributed systems have many waiting tasks to run. They efficiently decide where and when a task should run to minimize overall time to complete tasks(throughput) and ensure high-priority tasks run immediately (latency).

Examples

- First come, first served- run tasks in the order they arrive
- Shortest job first-prioritize task that take the shortest time
- Reward-aware scheduling- assign tasks to nodes with most available CPU or memory.

They are traffic controllers and dispatchers deciding how best to use available computing power to finish all jobs quickly and efficiently

<u>Advanced job specific scheduling</u>

a. Co-scheduling or Gang scheduling: schedules multiples interdependent sub-tasks of a single job to run simultaneously on different nodes. This reduces latency.
b. Backfilling: scheduler allows smaller shorter jobs to “fill the gaps” in the schedule as long as they don’t delay the start time of a large job waiting for its scheduled start time.
c. Min-Min algorithm: Prioritizes the one that has the earliest estimated completion time on its best possible machine. This optimizes completion time for many small jobs.
d. Max-Min algorithm: Prioritizes the one that has the latest estimated completion time on its best possible machine. This optimizes completion time for large jobs leading to better resource utilization overall.

4. Load Balancing strategies

Ensure that work (requests, tasks, or data) is distributed evenly among nodes to minimize overload and improve performance. Effective load balancing increases productivity, lowers response times, and improves cloud services and resource use.

Main Classification include:

- Static (Deterministic) load balancing

  Decisions are made using fixed rules or precomputed data (for example, round-robin and hash partitioning). There is no feedback during runtime.

- Dynamic (Adaptive) load balancing

  Decisions are made based on current system state (CPU, queue length, latency) and are adaptable at runtime (for example, least-connections, weighted least-load).

- Distributed vs Centralized
  - ➔ Centralized: A single load balancer makes global decisions (simpler, but with a single point of failure and scaling limits).
  - ➔ Distributed: Local balancing choices are made by a large number of nodes, making them more difficult to coordinate but more scalable and fault-tolerant.
- Layered approaches

  Load balancing occurs at multiple stages of the stack, including DNS, transport/proxy (L4), application (L7), and storage/data partitioning.

<u>Common Algorithms and how they work</u>

1. Round Robin (Static)

   It is a simple and straightforward way to send requests to servers sequentially. It performs best when requests have consistent resource requirements and the servers are identical.

2. Weighted Round Robin

   Requests are distributed using a predetermined weight that accounts for each server's capability. To achieve proportional load balancing, a server with a weight of three receives three times as many requests as one with a weight of one.

3. Least Load (Dynamic)

   This dynamic approach sends incoming requests to the server with the fewest active connections. The Least Load variation looks at metrics like CPU and queue length to find the least busy server, resulting in a balanced workload.

4. Hash-based (Static/Dynamic Hybrid)

   Consistent hashing ensures that client requests are sent to the same server by calculating a hash value from a key (e.g. IP address or user ID). Consistent hashing is essential for caches, decreasing data transmission when servers are added or removed.

5. Randomized/ Power - of - Two - Choices

   Pure randomness could be inefficient. The Power-of-Two-Choices strategy significantly improves this by randomly selecting a small number of servers (for example, $k=2$) and choosing the least loaded one. This gives excellent performance with minimal overhead.

6. Request Routing/ Application - aware (L7)

   This is the most intelligent strategy since it checks the request's content (such as the URL or headers) before routing. It enables for complex routing decisions, such as maintaining session affinity or diverting traffic for A/B testing.

Design Considerations and Alternatives

- Intelligence versus simplicity. Adaptive algorithms use more measurements but can fluctuate if unstable, whereas simple algorithms are easier to build and more predictable.
- Stateless services are easier to balance than stateful services (session-affinity), which rely on sticky sessions or consistent hashing.
- Decision delay: While centralized global decisions may be the best solution, they introduce latency and a single point of failure.
- Distributed load balancing is more scalable, but it necessitates a constant viewpoint or ultimate consistency.
- Overhead includes CPU and network costs associated with continuous metric gathering.

Performance indicators to measure

a. Operational efficiency - Requests per second
b. Response/ Latency time - avg, p95, p99
c. Resource utilization - CPU, memory, network per node
d. Load imbalance - standard deviation per - lode load
e. Request drop rate/ error rate
f. Rebalancing cost - amount of data or state transferred when nodes join or depart

Examples in Telecommunication and Cloud

- Load balancers handling SIP signaling or RTP proxying for VoIP.

  SIP Signaling: Uses Hash-based or Application-aware routing to ensure all parts of one call stay on the same server (session affinity), preventing drops.

  RTP Proxying: Uses simple methods like Round Robin or Least Connections to distribute the heavy media traffic for quality voice and low delay.

- CDN edge caches using consistent hashing to assign content.

  Consistent Hashing: Used in Content Delivery Networks (CDNs) to assign content to local cache servers. It ensures that when a server changes, minimal content has to be moved, keeping the system stable and fast.

- Microservices behind an API gateway using least-connections or L7 routing.

  Least Connections: The API Gateway uses this to send new requests to the least busy service instance, minimizing user latency.

  L7 Routing: Uses Application-aware routing to check the request (e.g., URL path) and direct it precisely to the correct, specialized microservice (e.g., the User Service).

5. Fault Tolerance Techniques

These techniques allow a system to function normally even when some of its components fail. High availability is critical in distributed information and telecommunication systems.

Key Methods

1. Replication
   - Active (master-master / multi-leader): several replicas accept writes but require strong dispute resolution or consensus.
   - Primary-backup (master-slave): one primary accepts write requests and copies them to backups. Simpler, but leaders must be elected for failover.

   Examples include databases, stateful services, and SIP registrar clusters.

2. Checkpointing and Rollback Recovery

   Periodically save the execution state (checkpoint). Restore the most recent checkpoint if the previous one fails. Used for lengthy computations and streaming pipelines.

3. Heartbeats and Failure Detection

   Nodes exchange regular heartbeats. Missing heartbeats cause failover or reconfiguration.

4. Leader Election

   When one coordinator fails, algorithms (Bully, Raft-style election) are used to select another. Leader elections are frequently paired with consensus procedures.

5. Consensus Protocols

   Paxos and Raft are used to ensure that nodes maintain consistent replicated state despite failures and message delays.

6. Quorum-based operations

   Before operation to be declared committed, a majority (or other quorum) of replicas must recognize it. Balancing availability and consistency.

7. Redundancy and Diversity

   To reduce common-mode failures, use hardware redundancy (several NICs, power supplies) and software diversity (various versions/configs).

8. Partition Tolerance Strategies

   Partition detection and degraded operating modes. Systems must choose whether to stay available despite potential inconsistency or to preserve consistency by denying operations.

9. Graceful Degradation & Circuit Breakers

   When downstream services fail, throttle or refuse requests promptly to avoid cascading failures. Circuit breakers enable systems to recover.

CAP concerns and tradeoffs

Since Partition Tolerance (P) is a necessity in a distributed system, the trade-off is between Consistency (C) and Availability (A).

- Availability vs. Consistency: During partitioning, systems must choose whether to reject operations in order to maintain consistency or to serve requests that may be available but inconsistent.
- Durability vs. performance: Increased replication improves durability while increasing write latency.
- Complexity: Implementing and debugging strong fault tolerance (consensus, multi-leader replication) is challenging.

Metrics for determining fault tolerance

A variety of key metrics are used to assess how effectively fault tolerance works in distributed systems. The Mean Time Between Failures (MTBF), which indicates how long the system typically functions without difficulties, and the overall percentage of availability (uptime) are used to assess reliability. The Mean Time to Recovery (MTTR) statistic assesses the effectiveness of the recovery procedure following a failure.

Finally, the likelihood of data loss, the amount of lost or rolled-back work, and the recovery time are utilized to assess the fault tolerance mechanism's genuine effectiveness.

Real World Examples

- Replicated Service Registries (etcd/consul)

  These tools function as a dependable, shared phonebook for all telecom cloud services; they are replicated across several nodes so that, in the case of a node failure, services may still quickly locate and communicate with one another (service discovery), ensuring operational continuity.

- Active-Active Data Centers for Carrier-Grade Availability

  This method entails running identical, live services in two or more separate data centers. Zero downtime is required for carrier-grade (extremely high-reliability) communications services, and if one data center fails (due to a natural disaster, for example), the other is prepared to take over all traffic at once.

- Checkpointing in Long-Running Billing/Charging Batch Jobs

  Billing and billing activities can take several hours or days to complete. Checkpointing is used on a regular basis to save progress in certain jobs. If the system crashes in the middle of a job, it might resume from the last saved state (checkpoint). This saves time while ensuring that critical financial records are correct.

6. Distributed Search Retrieval Algorithms

Utilized in systems where data is spread among multiple nodes. These algorithms make it easier to search for and retrieve data. This is critical for systems like large-scale search engines, P2P networks, and telecom clouds.

Main Approaches

The core approaches to distributed search and retrieval include indexing data and routing requests throughout the network.

1. Centralized Index / Directory: This method employs a single, authoritative index to map all data keys to the specific nodes where the data is stored. It allows for quick lookups, but has a single point of failure and scalability restrictions.
2. Distributed Hash Tables (DHTs): DHTs create a peer-to-peer overlay network by mapping keys to one or more nodes via consistent hashing. DHTs like Chord and Kademlia provide decentralized lookup efficiency, frequently requiring only O(log N) network hops.
3. Gossip Protocols / Epidemic Protocols: This technique involves nodes periodically and randomly exchanging state information with their peers.

Information spreads probabilistically over time (anti-entropy), making it effective for disseminating membership data and achieving eventual consistency.

4. Query Flooding / Broadcast: The most basic approach, in which a request is sent blindly to all available nodes until the desired data is found. While simple to implement, it is not scalable and is typically limited to very small or rare-data networks.
5. Federated / Hierarchical Search: This necessitates organizing the index in a hierarchy. Queries are routed through this structure to reduce search scope and improve lookup latency.
6. Inverted Index & Distributed Indexing (Search Engines): This is important for text search since it involves building and spreading inverted indices (where words map to document locations) among multiple nodes (sharding). Queries are then routed to appropriate shards, with the results pooled and rated.

Key Algorithms

1. Nodes and data keys are mapped to a circular identifier space using **Chord (DHT)**. It offers efficient lookups in O(Log N) network hops by utilizing a finger table for shortcuts.
2. **Kademlia (DHT)** calculates the distance between keys and nodes using the XOR metric. It is robust and popular in peer-to-peer applications because it does concurrent, iterative lookups.
3. **Anti-entropy and gossip**: Nodes interact with one another at random regarding their status. As a result of this process, all nodes eventually agree on the same system view, providing significant resistance to churn and failure.

Trade-offs and Considerations

1. Latency vs. Bandwidth: Central indices require more core bandwidth, whereas DHT lookups involve more hops (higher latency).
2. Consistency vs. Availability: Search indices may prefer consistency (CP) by updating synchronously (slower writes) or availability (AP) by updating lazily (faster writes).
3. Churn Handling: To properly deal with frequent node joins and departures (churn), P2P systems must include protocols such as replication and stabilization.
4. Result Quality: Complex ranking logic is required to ensure that results from multiple shards are relevant and fresh.

Performance Metrics

Key metrics include: lookup latency (average and tail), number of messages/hops per lookup, success chance under failure (churn), and index freshness (staleness).

Real-world examples

These algorithms are fundamental to:

- Chord and Kademlia derivatives in decentralized P2P applications.

- Large search engines utilize sharded inverted indices and query routers.
- Distributed directories are used in telecommunications to quickly map a subscriber to their present network node.

7. Data partitioning and sharding

Theoretical Aspects

Data partitioning and sharding are foundational techniques for achieving horizontal scalability in distributed databases and storage systems. The core principle involves dividing a large dataset into smaller, more manageable segments called partitions or shards, which are then distributed across multiple nodes in a cluster.

- Sharding vs. Partitioning: While often used interchangeably, a key distinction exists. Sharding is a specific type of partitioning that involves distributing data across multiple independent database instances, often on different servers, to spread load. In contrast, general partitioning can also refer to dividing data within a single database instance for organizational purposes.
- Core Objectives: The primary goals are scalability, performance, and availability. By distributing data, the system can handle read and write operations that exceed the capacity of a single machine. It also enhances fault tolerance; the failure of one shard does not render the entire dataset unavailable.
- Partitioning Strategies: The method of splitting the data is critical and is typically governed by a shard key.
    - ❖ Horizontal Partitioning (Sharding): This is the most common strategy, where rows of a table are distributed across different shards based on the shard key.
    - ❖ Vertical Partitioning: This strategy splits a table by its columns. For instance, frequently accessed columns might be stored in one partition, while rarely accessed, large-text or BLOB columns are stored in another.
- Sharding Architectures: The algorithm that determines which shard receives a piece of data is vital for performance and load distribution.
    - ❖ Key-Based (Hash-Based) Sharding: A shard key (e.g., user_id) is passed through a consistent hash function. The output hash determines the specific shard. This ensures a uniform data distribution, preventing "hotspots," but can make resharding, adding or removing shards, complex and expensive.
    - ❖ Range-Based Sharding: Data is partitioned based on a range of values of the shard key (e.g., users A-D on shard 1, E-H on shard 2). This is efficient for range queries but is highly susceptible to hotspots if the shard key is not chosen carefully.
    - ❖ Directory-Based Sharding: A flexible lookup service (a "directory") maintains a mapping of which shard key corresponds to which shard. While offering maximum flexibility, the directory itself can become a single point of failure and a performance bottleneck.

Practical Applications and Relevance

1. Large-Scale Web Applications: Social media platforms and e-commerce sites shard their user data across thousands of database servers to handle millions of concurrent interactions.
2. Modern Telecommunication Networks: In 5G/6G networks, subscriber data is stored in a distributed Unified Data Repository (UDR). Sharding is critical here, where data is partitioned by a Subscription Permanent Identifier (SUPI), allowing the network to handle authentication, billing, and policy enforcement for millions of subscribers simultaneously with low latency.
3. Large-Scale Cloud Computing Environments: All major cloud providers offer managed database services (e.g., Amazon DynamoDB, Azure Cosmos DB) that abstract away the complexity of sharding. They automatically handle partitioning, replication, and scaling based on configured throughput, making sharding a commodity service for cloud-native applications.

8. Large-scale data processing algorithms

Theoretical Aspects: Beyond MapReduce

While MapReduce was a groundbreaking model for parallel data processing, the field has evolved significantly. The core theoretical challenge is to perform computations over massive datasets in a parallel, distributed manner while managing communication, coordination, and fault tolerance.

- The MapReduce Paradigm: This model simplifies distributed programming by breaking it into two user-defined functions: Map and Reduce. The Map function processes a key-value pair to generate a set of intermediate key-value pairs. The Reduce function merges all intermediate values associated with the same intermediate key. The underlying execution framework handles parallelization, shuffling & sorting (the network-intensive phase of grouping data by key), and fault tolerance.
- Evolution to Modern Distributed Algorithms: Current research focuses on a broader class of distributed algorithms for optimization and resource allocation. These are often designed for multi-agent systems, where multiple nodes must collaboratively solve a problem without a central coordinator.
- Distributed Resource Allocation (DRA): This is a significant area of research at the intersection of optimization and multi-agent systems. DRA algorithms are designed to efficiently distribute limited resources (CPU, memory, bandwidth) among multiple entities. They are characterized by key properties:
    - ❖ Scalability: Handling large-scale systems with numerous entities.
    - ❖ Fault-tolerance and Robustness: Continuing to function despite node failures or network partitions.
    - ❖ Privacy and Security: Allowing agents to make decisions based on private or limited shared information.

- Algorithmic Approaches: A 2025 survey classifies DRA solutions into linear, nonlinear, primal-based, and dual-formulation-based approaches, analyzing them for feasibility, convergence rate, and network reliability.

Practical Applications and Relevance

- Log Analysis and Inverted Index Construction: The canonical use case for MapReduce, processing terabytes of web server logs or building the index for search engines.
- Smart Grids and Energy Management: DRA algorithms are used for distributed economic dispatch, where various energy resources (solar, wind, batteries) collaboratively decide on optimal generation and consumption schedules to meet demand while minimizing costs, all without a centralized controller.
- Telecommunication Networks: Telecom operators use frameworks like Apache Spark (a successor to MapReduce) for network analytics. They process millions of Call Detail Records (CDRs) to identify congestion, analyze call drops, and detect fraudulent activities like SIM box fraud.
- Cloud Computing and Edge Environments: Modern frameworks have evolved to be more efficient. Apache Spark uses in-memory data structures for faster iterative processing. Furthermore, Machine Learning-based cloud resource allocation is now a dominant trend. A 2025 review highlights the use of Deep Reinforcement Learning (DRL), Neural Networks, and Multi-Agent systems to dynamically allocate CPU, memory, and storage, significantly outperforming traditional heuristic methods in metrics like cost optimization and energy efficiency.

Python Demonstrations

We worked on three implementations for our demonstrations. All the Python files for our work have been provided on GitHub as instructed. Here, we have brief explanations of what the implementations are about.

1. **Fault Tolerance Technique**

This simulation demonstrates fault tolerance inside the BTS hardware.

BTS moves between OK, DEGRADED, and DOWN.

DEGRADED mode represents:

- TRX failure
- Power Amplifier failure
- High VSWR
- Cooling failure
- Reduced transmit power

If a BTS fails, then the neighbouring BTSs pick up users.

Users are not instantly dropped when a BTS fails; they shift to neighbours depending on capacity.

Metrics Produced:

- How many users were recovered by neighbors
- How many dropped due to insufficient redundancy
- Active load per BTS per cycle
- BTS health state transitions (OK - DEGRADED - DOWN - RECOVERY)

2. **Load Balancing Strategies**

This program simulates how traffic is distributed across multiple telecom servers using a weighted load-balancing algorithm, aiming to mimic how real telecom systems, such as core network nodes, share incoming load to avoid overload and maintain low latency.

Each node represents a telecom server with a capacity that indicates how much traffic it can normally handle, a weight that determines how much traffic it should receive compared to others, a current load showing how busy the node is at the moment, a latency history tracking delays experienced by traffic, and a dropped counter for traffic rejected due to overload

The program generates 30 random traffic events, each between 10 and 40 units, and for each traffic event;

- The load balancer picks a node based on weights
- Then the chosen node tries to handle the traffic
- Then the program prints which node handled it
- Documents the latency created and whether it was dropped due to overload.

3. **Data Partitioning and Sharding**

   The sample demonstrated is a simulation of how telecom subscriber data is stored in a distributed database.

   It demonstrates:

   1. Creating multiple shards
   2. Generating Telecom Subscriber data.
   3. Distributing subscribers across shards using hashing.
   4. Inserting records into the correct shard.
   5. Printing the shard distribution.
   6. Looking up subscriber records.

This is useful because it is used to handle millions of subscribers efficiently.

Classification of Modern Distributed Algorithms

The table below categorizes key distributed algorithm types, their primary goals, and illustrative techniques, synthesizing information from multiple contemporary sources.

Algorithm Type	Primary Goal	Key Techniques & Examples
Consensus Algorithms	Achieve agreement on a value/state among distributed nodes despite failures.	Paxos, Raft, and their variants.
Distributed Optimization	Solve optimization problems where data/control is spread across multiple agents.	Primal/Dual decomposition, gradient-based methods, aggregative game theory.
Distributed Graph Algorithms	Solve graph problems (e.g., pathfinding, cuts) in a network where each node only knows its neighbors.	Local computations and message passing; e.g., algorithms for Locally Optimal Cut.
Multi-Agent Learning	Enable multiple agents to learn optimal decisions in a shared, dynamic environment.	Multi-Agent Reinforcement Learning (MARL), Federated Learning.

Relevance to Modern Telecommunications Networks

Distributed algorithms are not just supporting but actively transforming telecommunication networks, enabling them to become more intelligent, autonomous, and efficient.

- Agentic AI for Network Operations: The emergence of Agentic AI marks a shift towards autonomous, goal-driven software agents that can perceive the network environment, make decisions, and act to accomplish complex goals. For telecom, this means networks can self-optimize, predict and preempt faults, and dynamically manage traffic and spectrum resources in real-time, moving towards Autonomous Networks Level 4.

- Integrated Sensing and Communication (ISAC): This is a paradigm shift that merges connectivity with sensing capabilities, allowing mobile infrastructure (like base stations) to serve as a distributed sensing platform. This could enable motion detection, environmental mapping, and enhanced interference monitoring from the same physical layer that carries communications traffic, opening up new services for 6G.
- Network API Economy: Mobile networks are evolving into platforms through standardized Network APIs. These APIs expose network capabilities (like latency, speed, and location verification) to developers, enabling innovative, context-aware applications in sectors like finance (for fraud reduction) and logistics (for augmented reality collaboration).
- AI-Driven Security and Efficiency: Startups and researchers are applying advanced distributed AI to solve pressing telecom issues. This includes deepfake detection in voice traffic, optimizing AI itself for energy efficiency at the network edge, and using AI for robust threat detection in encrypted network environments.

Distributed Algorithms in Cloud & Information Systems

In cloud computing and distributed information systems, the focus is on intelligent, adaptive, and highly consistent resource management.

- Adaptive Distributed Optimal Resource Allocation: Recent research focuses on algorithms that dynamically adjust resource allocation in real-time based on network conditions and user demands. For example, a 2025 study combined a Q-learning algorithm with an extended Paxos consensus protocol to ensure global consistency in Wide-Area Network (WAN) environments, achieving a 97% average resource utilization rate and significantly faster response times for cloud resource scheduling.
- The Rise of AI/ML-Based Allocation: Traditional heuristic approaches are being replaced by machine learning techniques. A 2025 comparative review found that hybrid architectures combining multiple AI/ML techniques (e.g., Deep Reinforcement Learning with Genetic Algorithms) consistently outperform single-method approaches. These are particularly effective in the dynamic, heterogeneous environments of multi-cloud and edge computing infrastructures.
- Algorithmic Aspects in Academia: The importance of this field is underscored by its prominence in graduate-level courses at leading institutions like MIT, which cover fundamental aspects like "fault-tolerance, asynchrony, bandwidth limitations and congestion" in distributed algorithms.

Future directions are already taking shape, driven by trends such as:

- **Agentic AI** transforming network operations and service innovation.
- **Quantum distributed computing** principles beginning to influence cryptography and optimization
- The integration of **AI-driven distributed systems** for improved decision-making and resource allocation.
  The theoretical foundations of partitioning, sharding, and distributed computation, combined with these advanced algorithmic approaches, ensure that distributed systems will continue to meet the escalating demands of modern telecommunications, cloud computing, and global-scale information systems.

Conclusion

Distributed processing algorithms are the bedrock of our scalable digital infrastructure. The journey from the foundational MapReduce model to sophisticated Distributed Resource Allocation and Multi-Agent AI systems highlights a continuous evolution towards greater autonomy, intelligence, and efficiency.