This project is a group project, each group will be 2 to 4 students. Each group will deliver a written report in addition to the developed code by the end of the project that shows the detailed analysis, design, testing … etc. of the project and the description of the used techniques, problem description, detailed solution, limitations, sample output, references, and any additional information related to the project. Online Demo (YouTube) Video, presentation, and discussion will be conducted by the end of the project.

Introduction

This coursework is itemized into several parts to get the 35 marks associated to it.

You must use the templates provided by the instructor to prepare your work.

All assignments and projects will be handed-in electronically, while quizzes and exams are written

.

Learning Outcome to be assessed

1. Design a distributed computing model to solve a complex problem
2. Design and Implement a distributed computing model
3. Configure a working environment for distributed computing
4 Work and communicate effectively in a team

Marking Criteria

89% and above:

Your work must be of outstanding quality and fully meet the requirements of the coursework specification and learning outcomes stated. You must show independent thinking and apply this to your work showing originality and consideration of key issues. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a professional ability of drafting construction details,
- express a deep understanding of the in-hand problem definition,
- and applying, masterly, the learned knowledge in the proposed solution.

76% - 89%:

Your work must be of good quality and meet the requirements of the coursework specification and learning outcomes stated. You must demonstrate some originality in your work and show this by applying new learning to the key issues of the coursework. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a Good ability of drafting construction details,
- express a very Good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

67% - 76%:

Your work must be comprehensive and meet all of the requirements stated by the coursework specification

and learning outcomes. You must show a good understanding of the key concepts and be able to apply them to solve the problem set by the coursework. There must be enough depth to your work to provide evidence of wider reading. In addition, your proposed solution should:

- illustrate a moderate ability of drafting construction details,
- express a good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

60% - 67%:

Your work must be of a standard that meets the requirements stated by the coursework specification and learning outcomes. You must show a reasonable level of understanding of the key concepts and principles and you must have applied this knowledge to the coursework problem. There should be some evidence of wider reading. In addition, your proposed solution should:

- illustrate a fair ability of drafting construction details,
- express a fair understanding of the in-hand problem definition,
- and applying some of the learned knowledge, correctly, in the proposed solution.

Below 60%:

Your work is of poor quality and does not meet the requirements stated by the coursework specification and learning outcomes. There is a lack of understanding of key concepts and knowledge and no evidence of wider reading. In addition, your proposed solution would be:

- Illustrate an inability of drafting construction details,
- Failed to define the parameters, limitations, and offerings of the in-hand problem,
- Failed to apply correctly the learned knowledge for proposing a valid solution.

Academic Misconduct

The University defines Academic Misconduct as 'any case of deliberate, premeditated cheating, collusion, plagiarism or falsification of information, in an attempt to deceive and gain an unfair advantage in assessment'. This includes attempting to gain marks as part of a team without making a contribution. The department takes Academic Misconduct very seriously and any suspected cases will be investigated through the University's standard policy. If you are found guilty, you may be expelled from the University with no award.

It is your responsibility to ensure that you understand what constitutes Academic Misconduct and to ensure that you do not break the rules. If you are unclear about what is required, please ask.

Ain Shams University
Faculty of Engineering
CSE354: Distributed Computing – 2nd Semester 2024/2025

Title: Distributed Web Crawling and Indexing System using Cloud Computing

Description: This project aims to develop a distributed web crawling and indexing system using cloud computing technologies. The system will be implemented in Python, leveraging cloud-based virtual machines for distributed computing. The application will use distributed task queues and storage to crawl websites and build a searchable index of web pages. A key focus will be on fault tolerance to ensure continuous crawling and indexing even if parts of the system fail.

Features and Specifications:

- **Distributed Crawling:** The system should distribute web crawling tasks across multiple virtual machines in the cloud to efficiently crawl a large number of websites.
- **Web Page Indexing:** Implement an indexing mechanism to process crawled web pages and create a searchable index. This could involve extracting text content, keywords, and metadata.
- **Scalability:** The system should be scalable, allowing for the addition of more virtual machines as the crawl scope or indexing workload increases.
- **Fault Tolerance:** The system must be resilient to failures. This includes:
  - **Crawler Node Failure:** Ability to reassign crawling tasks from failed crawler nodes to operational ones.
  - **Indexer Node Failure:** Mechanisms to ensure index integrity and continued indexing even if indexer nodes fail.
  - **Data Persistence:** Durable storage of crawled data and index to prevent data loss in case of failures.
- **Politeness and Respect for robots.txt:** The crawler must respect robots.txt directives to avoid overloading servers and adhere to website crawling policies.
- **Basic Search Functionality:** Implement a simple search interface to query the indexed content.

User Stories:

- As a user, I want to initiate a web crawl by providing a list of seed URLs.
- As a user, I want to specify crawling parameters such as depth limit or domain restrictions.
- As a user, I want the system to continue crawling even if some crawler nodes fail.
- As a user, I want to search for keywords and retrieve relevant web pages from the indexed content.
- As a user, I want to monitor the progress of the web crawling and indexing process.

System Architecture:

- **Client (User Interface):**
  - A web interface or command-line tool to:
    - Start crawls with seed URLs and parameters.
    - Submit search queries.
    - Monitor crawling/indexing progress.
- **Master Node (Controller/Scheduler):**
  - Manages the overall crawl process.
  - **Task Distribution:** Divides the list of URLs to crawl into smaller tasks.
  - **Task Scheduling:** Assigns crawling tasks to available crawler nodes.
  - **Worker Management:** Manages crawler and indexer nodes.
  - **Fault Detection:** Monitors health of worker nodes and reassigns tasks upon failure.
  - **Index Management:** Coordinates index building and merging (if using distributed indexing).
- **Crawler Nodes (Crawling VMs):**
  - Multiple VMs responsible for fetching web pages.
  - **URL Fetching:** Downloads web pages from assigned URLs.
  - **Parsing and URL Extraction:** Parses HTML content, extracts text and links to new URLs.
  - **Politeness:** Respects robots.txt and implements crawl delays.
  - **Data Storage (Temporary):** Temporarily stores crawled content before sending to indexers.
  - **Status Reporting:** Reports crawling status and extracted URLs back to the master node.
- **Indexer Nodes (Indexing VMs):**
  - Multiple VMs responsible for building the search index.
  - **Data Ingestion:** Receives crawled web page content from crawler nodes.
  - **Indexing:** Processes content to create an index (e.g., inverted index).
  - **Index Storage (Persistent):** Stores the index in a distributed and fault-tolerant manner.
  - **Search Query Handling:** Handles search queries against the index (or delegates to a search service).
- **Distributed Task Queue (Message Queue):**
  - Cloud-based message queue service (e.g., AWS SQS, GCP Pub/Sub, Azure Queue Storage).
  - **Task Queuing:** Master node uses queues to distribute crawling and indexing tasks to worker nodes.
  - **Communication:** Worker nodes can use queues to report status and send data to other components.
- **Cloud Storage (Persistent Storage):**
  - Cloud storage service (e.g., AWS S3, GCP Cloud Storage, Azure Blob Storage).
  - **Seed URL Storage:** Stores initial seed URLs.
  - **Crawled Data Storage (Raw and Processed):** Stores raw HTML content and processed text.
  - **Index Storage:** Stores the searchable index (consider distributed database or search service for larger indexes).

Fault Tolerance Mechanisms to Implement:

- **Crawler Node Failure:**
  - **Heartbeat Monitoring:** Master node periodically checks the health of crawler nodes.
  - **Task Timeout:** If a crawler node doesn't report back within a timeout period, the master assumes failure.
  - **Task Re-queueing:** Master node re-queues URLs assigned to failed crawler nodes for

reassignment to other available crawlers.

- **Indexer Node Failure:**
  - **Data Replication:** Replicate index data across multiple indexer nodes for redundancy.
  - **Distributed Indexing:** Partition the index across multiple indexer nodes. If one node fails, only a part of the index might be temporarily unavailable, but the system can still function.
  - **Index Reconstruction/Recovery:** Implement mechanisms to rebuild or recover parts of the index if indexer nodes fail and data is lost (e.g., from logs or backups).
- **Message Queue Reliability:** Rely on the fault-tolerant nature of cloud-based message queue services (they typically handle message persistence and delivery even if components fail).
- **Persistent Storage:** Use durable cloud storage for crawled data and the index to prevent data loss.

Technology Suggestions:

- **Programming Language:** Python (again, excellent libraries for web crawling and data processing)
- **Web Crawling Libraries:** Scrapy, Beautiful Soup, requests
- **Indexing Libraries/Search Engines:** Whoosh (for smaller projects, Python-based), Elasticsearch, Solr (for more scalable solutions, consider cloud-managed services)
- **Distributed Task Queue:** Celery (with Redis or RabbitMQ as brokers - can be deployed in the cloud), or cloud-native services like AWS SQS, GCP Pub/Sub, Azure Queue Storage.
- **Cloud Platform:** AWS, GCP, or Azure (choose one based on familiarity and cost considerations).
- **Database (for storing crawl metadata, URLs, etc.):** Cloud-managed databases like AWS RDS, GCP Cloud SQL, Azure SQL Database, or NoSQL options like Cassandra or MongoDB for scalability.

Implementation Steps:

1. **Cloud Platform Setup.**
2. **VM Configuration (for Crawler, Indexer, Master Nodes).**
3. **Master Node Implementation (Python):** Task scheduling, worker management, fault detection, etc.
4. **Crawler Node Implementation (Python):** Web page fetching, parsing, URL extraction, politeness logic.
5. **Indexer Node Implementation (Python):** Data ingestion, indexing, search functionality (basic).
6. **Client Interface Development.**
7. **Distributed Task Queue Integration.**
8. **Cloud Storage and Database Integration.**
9. **Fault Tolerance Implementation (monitoring, task re-queueing, data replication).**
10. **Testing and Evaluation (functional, scalability, fault tolerance, crawl quality).**
11. **Documentation.**

Testing and Evaluation (Focus on Fault Tolerance):

- **Crawler Failure Simulation:** Kill crawler VMs during a crawl and verify that crawling continues and tasks are reassigned. Check for data loss.
- **Indexer Failure Simulation:** Simulate indexer node failures during indexing and search operations. Verify index availability and recovery mechanisms.
- **Data Persistence Testing:** Test scenarios where storage or database components might fail and ensure data integrity and recoverability.
- **Crawl Quality Metrics:** Measure crawl coverage, politeness, and identify any issues with respecting robots.txt.

Project Phases

Phase 1: Project Inception and Core Architecture (Weeks 1-2)

- **Focus:** Establish project foundations, define core architecture, set up cloud environment, and plan in detail.

  - **Activities:**

    1. **Team Formation and Roles:** Define roles within the team (e.g., Architect, Crawler Lead, Indexer Lead, Cloud Infrastructure Lead, Tester/Documentation Lead).
    2. **Requirement Refinement:** Re-read and clarify all project requirements and specifications. Ensure everyone understands the scope, features, and user stories.
    3. **Technology Deep Dive:** Finalize technology choices (cloud provider, libraries for crawling, indexing, task queue, storage, database). Document justifications for each selection.
    4. **System Architecture Design (Detailed):**
       - Create detailed diagrams for system components (Client, Master, Crawler, Indexer, Task Queue, Storage) and their interactions.
       - Define data flow between components.
       - Design API interfaces (even if internal) for communication.
       - Plan for data structures and storage schemas.
       - Outline initial fault tolerance strategies (focus on crawler node failure in this phase).
    5. **Detailed Project Planning:**
       - Break down each phase into smaller, manageable tasks.
       - Assign tasks to team members based on roles and skills.
       - Create a detailed timeline using Gantt charts or similar tools, allocating time for each task within the 8 weeks.
       - Set milestones for each phase.
    6. **Cloud Environment Setup (Basic):**
       - Set up accounts with the chosen cloud provider.
       - Configure basic cloud resources (initial VMs for Master, Crawler, Indexer for testing, Task Queue service, Storage service).
       - Establish basic network configurations for communication between VMs.
    7. **Initial Code Repository Setup:**
       - Create a Git repository for version control.
       - Set up basic project directory structure.
       - Implement basic "skeleton" code for Master, Crawler, and Indexer nodes (classes, function stubs) to outline the architecture.
    8. **Phase 1 Deliverables:**
       - **Phase 1 Report:** Answer Phase 1 Deliverable Questions (What is the objective? Technologies? Architecture?).
       - **System Architecture Document (Version 1):** Detailed diagrams, component descriptions, data flow, initial fault tolerance plan.
       - **Detailed Project Plan (Gantt Chart/Timeline):** Task breakdown, assignments, timeline.

    - **Technology Selection Document:** Justification for chosen technologies.
    - **Initial Code Repository (Skeleton Code):** Basic structure and component outlines.
- **Software Engineering Process Element:** *Inception and Elaboration* phases of a simplified iterative process. Focus on understanding requirements and planning the architecture.

Phase 2: Core Crawling and Basic Indexing Functionality (Weeks 3-4)

- **Focus:** Implement the fundamental crawling process in a distributed manner, and establish a basic, functional (but not necessarily scalable or robust yet) indexing mechanism.

- **Activities:**

    1. **Crawler Node Implementation (Core Crawling):**
        - Implement web page fetching (using `requests` or `Scrapy`).
        - Implement basic HTML parsing and text extraction (using `Beautiful Soup`).
        - Implement URL extraction from web pages (basic link finding).
        - Implement *basic* politeness measures (crawl delay - simple).
        - Implement communication to report crawled content and extracted URLs back to the Master.
    2. **Master Node Implementation (Task Distribution and Worker Management):**
        - Implement task queue integration (e.g., using Celery or cloud queue service).
        - Develop logic for dividing seed URLs into crawling tasks.
        - Implement task assignment to Crawler Nodes via the task queue.
        - Implement basic Crawler Node lifecycle management (start, stop, basic monitoring - ping/heartbeat).
    3. **Basic Indexer Node Implementation (Single Node, Simple Index):**
        - Implement a *single* Indexer Node for simplicity initially.
        - Implement data ingestion from Crawler Nodes (via queue or direct communication - simpler method initially).
        - Implement a *very basic* in-memory index or file-based index (e.g., Python dictionary, or a basic Whoosh index if time allows) for storing keywords and associating them with URLs.
        - Implement a *very simple* keyword search functionality against the index (exact match).
    4. **Basic Communication and Workflow Integration:**
        - Establish communication flow: Master -> Task Queue -> Crawlers, Crawlers -> Indexer (initially direct or via queue), Client (for basic testing and monitoring) -> Master.
        - Integrate components to create a functional, albeit basic, crawling and indexing pipeline.
    5. **Initial Testing and Debugging:**
        - Test the basic crawling process with a small set of seed URLs.
        - Test basic indexing functionality and search.
        - Debug core crawling and indexing workflows.

6. **Phase 2 Deliverables:**
    - **Phase 2 Report:** Answer Phase 2 Deliverable Questions (Worker thread, basic operations, cloud setup).
    - **Functional Distributed Crawler (Basic Politeness):** Crawls websites in a distributed manner.
    - **Basic Indexer (Single Node, Simple Index):** Indexes crawled content and provides basic search.
    - **Initial Integration Test Results:** Report on basic functionality testing.
    - **Updated System Architecture Document (Version 2):** Reflecting any changes during implementation.

- **Software Engineering Process Element:** *Construction - Iteration 1*. Focus on building the core functionality and integrating basic components.

Phase 3: Enhanced Indexing, Fault Tolerance, and Monitoring (Weeks 5-6)

- **Focus:** Improve indexing capabilities for better search, implement fault tolerance for Crawler Nodes (and start considering Indexer fault tolerance), and add basic monitoring.

- **Activities:**

1. **Enhanced Indexer Node Implementation (Improved Index & Search):**
    - Replace the very basic index with a more robust indexing library like Whoosh (single node still, or explore a very basic Elasticsearch setup if ambitious and time allows).
    - Improve content processing for indexing: more robust text extraction, basic keyword analysis or stemming.
    - Enhance search functionality: implement slightly more advanced search (e.g., boolean operators, phrase search - depending on chosen index library).
2. **Fault Tolerance Implementation (Crawler Nodes):**
    - Implement heartbeat monitoring in the Master Node to track Crawler Node health.
    - Implement Task Timeout in Master Node: if a Crawler doesn't respond within a timeout, assume failure.
    - Implement Task Re-queueing in Master Node: re-assign URLs from failed crawlers to available ones.
    - **Basic Monitoring Implementation:**
    - Implement basic logging in Master, Crawler, and Indexer Nodes (more detailed logging than Phase 2).
    - Create simple monitoring dashboards or scripts to track:
        - Number of URLs crawled.
        - Number of URLs indexed.
        - Crawler Node status (active, failed).
        - Error rates.
3. **Data Persistence for Crawled Data:**
    - Implement storage of crawled content (raw HTML and/or processed text) in cloud storage (e.g., AWS S3). This is for data durability and potential re-indexing later.
4. **Client Interface Development (Basic Search Interface):**

- Develop a *very basic* web interface or command-line client for users to:
  - Submit search queries.
  - Potentially view basic crawling/indexing status.

5. **Testing (Fault Tolerance and Scalability (Basic)):**
   - Simulate Crawler Node failures during a crawl and verify task re-queueing and continued crawling.
   - Test the enhanced indexing and search functionality.
   - Perform basic "scalability" testing by running with more Crawler Nodes and observe if crawl rate increases (basic observation, not rigorous benchmarking yet).
6. **Phase 3 Deliverables:**
   - **Phase 3 Report:** Answer Phase 3 Deliverable Questions (Advanced operations, distributed processing, scalability/fault tolerance).
   - **Enhanced Indexer (Whoosh or similar):** More robust index and search.
   - **Crawler Node Fault Tolerance:** Heartbeat, timeout, task re-queueing implemented.
   - **Basic Monitoring System:** Logging and basic dashboards/scripts.
   - **Basic Search Interface (Client):** For querying the index.
   - **Updated Test Results (Fault Tolerance and Basic Scalability):** Report on testing fault tolerance and basic scalability observations.
   - **Updated System Architecture Document (Version 3):** Reflecting fault tolerance and monitoring components.

- **Software Engineering Process Element:** *Construction - Iteration 2* and *Transition - Preparation for Release*. Focus on enhancing features, adding robustness, and preparing for user interaction and testing.

Phase 4: System Refinement, Testing, Documentation, and Deployment (Weeks 7-8)

- **Focus:** Final system testing, bug fixing, performance tuning (if time permits), comprehensive documentation, and preparing for demonstration and final deliverables.

- **Activities:**

1. **Thorough System Testing (Functional, Fault Tolerance, Scalability):**
   - **Functional Testing:** Comprehensive testing of all user stories and features (crawling, indexing, search, politeness).
   - **Rigorous Fault Tolerance Testing:** Simulate various failure scenarios (Crawler Node failures, Indexer Node *simulated* failures if you have a distributed index – otherwise focus on Crawler failures and data persistence). Verify recovery mechanisms work as expected.
   - **Scalability Testing (More Detailed):** Conduct more systematic scalability testing by varying the number of Crawler Nodes and measuring crawl rate and indexing throughput. Document scalability characteristics.
   - **Crawl Quality Evaluation:** Assess crawl quality – coverage, politeness (verify robots.txt respect), identify any issues.
2. **Bug Fixing and Refinement:** Address bugs identified during testing. Refine the system based on testing feedback. Improve error handling and logging.

3. **Performance Tuning (Optional, if Time):** If time permits and performance bottlenecks are identified, perform basic performance tuning (e.g., optimize crawling speed, indexing efficiency, database queries).
4. **Security Review (Basic):** Review basic security aspects: access control to cloud resources, data handling. Document any security considerations.
5. **Comprehensive Documentation:**
    - **Final System Design Document:** Complete and update the System Architecture Document.
    - **User Manual:** Document how to use the search interface, start crawls, configure parameters.
    - **Code Documentation:** Add comments to code, generate API documentation if applicable.
    - **Deployment Guide:** Document how to deploy and run the system in the cloud environment (even if "deployment" is primarily for demonstration purposes).
6. **"Deployment" and Demonstration Preparation:**
    - Prepare scripts and instructions to easily launch all system components in the cloud environment for demonstration.
    - Prepare a compelling project demonstration showcasing all features and fault tolerance capabilities.
7. **Final Project Report:**
    - Complete Phase 4 Report Deliverable Questions (Testing results, documentation, deployment).
    - Write a comprehensive final project report summarizing the entire project: objectives, design, implementation, testing, results, lessons learned, challenges faced, and potential future work.
    - Prepare project presentation materials (slides, demo scripts).
8. **Phase 4 Deliverables:**
    - **Phase 4 Report:** Answer Phase 4 Deliverable Questions (Testing, Documentation, Deployment).
    - **Fully Tested and Functional Distributed Web Crawling and Indexing System.**
    - **Complete System Documentation:** System Design Doc, User Manual, Code Docs, Deployment Guide.
    - **Deployment/Demonstration Ready System in the Cloud.**
    - **Final Project Report and Presentation Materials.**

- **Software Engineering Process Element:** *Transition - Final Testing, Refinement, and Release* and *Deployment* (for demonstration). Focus on quality assurance, documentation, and preparing the system for its final presentation and hand-in.

Code Skeleton

1. Master Node (master_node.py)

```
from mpi4py import MPI
import time
import logging
# Import necessary libraries for task queue, database, etc. (e.g., redis,
cloud storage SDKs)

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - Master -
%(levelname)s - %(message)s')

def master_process():
    """
    Main process for the master node.
    Handles task distribution, worker management, and coordination.
    """
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    status = MPI.Status()

    logging.info(f"Master node started with rank {rank} of {size}")

    # Initialize task queue, database connections, etc.
    # ... (Implementation needed) ...

    crawler_nodes = size - 2 # Assuming master and at least one indexer node
    indexer_nodes = 1 # At least one indexer node

    if crawler_nodes <= 0 or indexer_nodes <= 0:
        logging.error("Not enough nodes to run crawler and indexer. Need at
least 3 nodes (1 master, 1 crawler, 1 indexer)")
        return

    active_crawler_nodes = list(range(1, 1 + crawler_nodes)) # Ranks for
crawler nodes (assuming rank 0 is master)
    active_indexer_nodes = list(range(1 + crawler_nodes, size)) # Ranks for
indexer nodes

    logging.info(f"Active Crawler Nodes: {active_crawler_nodes}")
    logging.info(f"Active Indexer Nodes: {active_indexer_nodes}")
```

```
    seed_urls = ["http://example.com", "http://example.org"] # Example seed
URLs - replace with actual seed URLs
    urls_to_crawl_queue = seed_urls  # Simple list as initial queue -
replace with a distributed queue

    task_count = 0
    crawler_tasks_assigned = 0


    while urls_to_crawl_queue or crawler_tasks_assigned > 0: # Continue as
long as there are URLs to crawl or tasks in progress
        # Check for completed crawler tasks and results from crawler nodes
        if crawler_tasks_assigned > 0:
            if comm.iprobe(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
status=status): # Non-blocking check for incoming messages
                message_source = status.Get_source()
                message_tag = status.Get_tag()
                message_data = comm.recv(source=message_source,
tag=message_tag)

                if message_tag == 1: # Crawler completed task and sent back
extracted URLs
                    crawler_tasks_assigned -= 1
                    new_urls = message_data # Assuming message_data is a
list of URLs
                    if new_urls:
                        urls_to_crawl_queue.extend(new_urls) # Add newly
discovered URLs to the queue
                    logging.info(f"Master received URLs from Crawler
{message_source}, URLs in queue: {len(urls_to_crawl_queue)}, Tasks assigned:
{crawler_tasks_assigned}")
                elif message_tag == 99: # Crawler node reports
status/heartbeat
                    logging.info(f"Crawler {message_source} status:
{message_data}") # Example status message
                elif message_tag == 999: # Crawler node reports error
                    logging.error(f"Crawler {message_source} reported error:
{message_data}")
                    crawler_tasks_assigned -= 1 # Decrement task count even
on error, consider re-assigning task in real implementation

        # Assign new crawling tasks if there are URLs in the queue and
available crawler nodes
        while urls_to_crawl_queue and crawler_tasks_assigned <
crawler_nodes: # Limit tasks to available crawler nodes for simplicity in
this skeleton
            url_to_crawl = urls_to_crawl_queue.pop(0) # Get URL from queue
```

```
(FIFO for simplicity)
            available_crawler_rank =
active_crawler_nodes[crawler_tasks_assigned % len(active_crawler_nodes)] #
Simple round-robin assignment
            task_id = task_count
            task_count += 1
            comm.send(url_to_crawl, dest=available_crawler_rank, tag=0) #
Tag 0 for task assignment
            crawler_tasks_assigned += 1
            logging.info(f"Master assigned task {task_id} (crawl
{url_to_crawl}) to Crawler {available_crawler_rank}, Tasks assigned:
{crawler_tasks_assigned}")
            time.sleep(0.1) # Small delay to prevent overwhelming master in
this example

        time.sleep(1) # Master node's main loop sleep - adjust as needed

    logging.info("Master node finished URL distribution. Waiting for
crawlers to complete...")
    # In a real system, you would have more sophisticated shutdown and
result aggregation logic

    print("Master Node Finished.")


if __name__ == '__main__':
    master_process()
```

2. Crawler Node (crawler_node.py)

```
from mpi4py import MPI
import time
import logging
# Import necessary libraries for web crawling (e.g., requests,
beautifulsoup4, scrapy), parsing, etc.

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - Crawler -
%(levelname)s - %(message)s')


def crawler_process():
    """
    Process for a crawler node.
    Fetches web pages, extracts URLs, and sends results back to the master.
    """
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    logging.info(f"Crawler node started with rank {rank} of {size}")

    while True:
        status = MPI.Status()
        url_to_crawl = comm.recv(source=0, tag=0, status=status) # Receive
URL from master (tag 0)
        if not url_to_crawl: # Could be a shutdown signal (if you implement
one)
            logging.info(f"Crawler {rank} received shutdown signal.
Exiting.")
            break

        logging.info(f"Crawler {rank} received URL: {url_to_crawl}")

        try:
            # --- Web Crawling Logic ---
            # 1. Fetch web page content (using requests, scrapy, etc.)
            # 2. Parse content (using BeautifulSoup4, etc.)
            # 3. Extract new URLs from the page
            # 4. Extract relevant text content for indexing (send to indexer
later or store temporarily)

            time.sleep(2) # Simulate crawling delay
```

```
            extracted_urls = [f"http://example.com/page_from_crawler_{rank}_{i}" for i in range(2)] # Example extracted URLs - replace with actual extraction
            # extracted_content = "Extracted content from " + url_to_crawl # Example content - replace with actual content extraction

            logging.info(f"Crawler {rank} crawled {url_to_crawl}, extracted {len(extracted_urls)} URLs.")

            # --- Send extracted URLs back to master ---
            comm.send(extracted_urls, dest=0, tag=1) # Tag 1 for sending extracted URLs

            # ---  Optionally send extracted content to indexer node (or queue for indexer) ---
            # indexer_rank = 1 + (rank - 1) % (size - 2) # Example: Send to indexer in round-robin (adjust indexer ranks accordingly)
            # comm.send(extracted_content, dest=indexer_rank, tag=2) # Tag 2 for sending content to indexer

            comm.send(f"Crawler {rank} - Crawled URL: {url_to_crawl}", dest=0, tag=99) # Send status update (tag 99)

        except Exception as e:
            logging.error(f"Crawler {rank} error crawling {url_to_crawl}: {e}")
            comm.send(f"Error crawling {url_to_crawl}: {e}", dest=0, tag=999) # Report error to master (tag 999)


if __name__ == '__main__':
    crawler_process()
```

3. Indexer Node (indexer_node.py)

```
from mpi4py import MPI
import time
import logging
# Import necessary libraries for indexing (e.g., whoosh, elasticsearch
client), database interaction, etc.

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - Indexer -
%(levelname)s - %(message)s')


def indexer_process():
    """
    Process for an indexer node.
    Receives web page content, indexes it, and handles search queries
(basic).
    """
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    logging.info(f"Indexer node started with rank {rank} of {size}")

    # Initialize index, database connection, etc.
    # ... (Implementation needed - e.g., Whoosh index creation,
Elasticsearch connection) ...


    while True:
        status = MPI.Status()
        content_to_index = comm.recv(source=MPI.ANY_SOURCE, tag=2,
status=status) # Receive content from crawlers (tag 2)
        source_rank = status.Get_source()

        if not content_to_index: # Could be a shutdown signal
            logging.info(f"Indexer {rank} received shutdown signal.
Exiting.")
            break

        logging.info(f"Indexer {rank} received content from Crawler
{source_rank} to index.")

        try:
            # --- Indexing Logic ---
```

```
            # 1. Process the received content (e.g., text cleaning,
tokenization)
            # 2. Update the search index with the content (e.g., using
Whoosh, Elasticsearch)

            time.sleep(1) # Simulate indexing delay

            logging.info(f"Indexer {rank} indexed content from Crawler
{source_rank}.")
            comm.send(f"Indexer {rank} - Indexed content from Crawler
{source_rank}", dest=0, tag=99) # Send status update to master (tag 99)


        except Exception as e:
            logging.error(f"Indexer {rank} error indexing content from
Crawler {source_rank}: {e}")
            comm.send(f"Indexer {rank} - Error indexing: {e}", dest=0,
tag=999) # Report error to master (tag 999)


if __name__ == '__main__':
    indexer_process()
```

To Run this Skeleton (Basic MPI Example):

1. **Save:** Save each code block as master_node.py, crawler_node.py, and indexer_node.py.
2. **Install mpi4py:** pip install mpi4py
3. **Run using mpiexec or mpirun:** You'll need an MPI implementation installed (like MPICH or Open MPI). For example, to run with 4 processes (1 master, 2 crawlers, 1 indexer):

   Bash
   ```
   mpiexec -n 4 python master_node.py
   # OR (depending on your MPI setup)
   mpirun -n 4 python master_node.py
   ```

   Important: When you run with mpiexec -n 4, MPI will automatically run 4 Python processes. The MPI.COMM_WORLD communicator will have size 4, and ranks 0, 1, 2, and 3 will be assigned to the processes. The code uses rank 0 as the master, ranks 1 and 2 as crawlers, and rank 3 as the indexer in this example setup.

Key Points and Next Steps:

- **Error Handling and Logging:** Basic error handling and logging are included. Enhance these for robustness.
- **Task Queue:** The urls_to_crawl_queue in the master is a simple Python list. Replace this with a distributed queue (like Redis Queue, or cloud-based queue services) for scalability and fault tolerance.
- **Content Storage & Indexing:** Implement the actual web crawling logic in crawler_node.py (using requests, BeautifulSoup4, Scrapy), and indexing logic in indexer_node.py (using Whoosh, Elasticsearch, etc.).
- **Fault Tolerance Implementation:** Expand on the basic error reporting to implement robust fault tolerance mechanisms (task re-assignment, data replication, etc.) as discussed earlier.
- **Communication Tags:** The code uses tags (0, 1, 2, 99, 999) for message differentiation. Use consistent tagging for different message types.
- **Scalability:** To scale, you would increase the number of crawler and indexer nodes when running with mpiexec -n <number_of_processes>. In a cloud environment, you would manage VMs dynamically.
- **Cloud Deployment:** Adapt the code to run on cloud VMs. You'll need to handle VM provisioning, MPI setup in the cloud, and cloud service integrations (storage, queues, databases).

Remember to install the necessary Python libraries (mpi4py, requests, beautifulsoup4, etc.) in your Python environment and within your cloud VM images. Let me know if you have specific questions about any part of this skeleton code or need help with implementation details!