

Exp 1 : Build a program for client/server using RPC/RMI

Client/Server Using RPC/RMI: Short Theory

Client/Server using RPC (Remote Procedure Call) and RMI (Remote Method Invocation) is a model for distributed computing where a client makes requests to a server, which processes them and returns the results. RPC and RMI are techniques used to enable communication between processes located on different machines in a network.

Remote Procedure Call (RPC):

- **RPC** allows a client to execute a procedure (or function) on a remote server as if it were a local call. The client sends a request (arguments) to the server, which processes the request and returns a response.
- **Key Features:**
 - **Transparency:** The client is unaware that the procedure is being executed remotely.
 - **Simplicity:** The client interacts with the server using local procedure call syntax, making it easier to use than other network communication methods.
 - **Transport Independence:** RPC abstracts communication, so different transport protocols (e.g., HTTP, TCP) can be used behind the scenes.

Example:

- A client application may use an RPC to request data from a remote server (e.g., "getUserInfo(userId)"), and the server executes this request and sends back the result.
-

Remote Method Invocation (RMI):

- **RMI** is similar to RPC but is specific to Java applications. It allows a Java object to invoke methods on another Java object located on a remote machine.
- **Key Features:**
 - **Object-Oriented:** Unlike RPC, RMI allows passing objects between the client and server. The client can call methods on remote objects directly.
 - **Java-Specific:** RMI is integrated with Java, allowing seamless communication between Java applications across different systems.
 - **Stubs and Skeletons:** RMI uses stubs (client-side proxy) and skeletons (server-side proxy) to facilitate communication between remote objects.

Example:

- A client in a Java program calls a method on a remote object, like `remoteObject.processRequest(data)`, where the method is executed on the server and the result is returned.
-

Why Use RPC/RMI for Client/Server Communication?

1. **Decoupling:** Clients and servers can be independently developed and maintained. The client only needs to know the interface of the server (the procedures or methods), not its internal implementation.
 2. **Simplified Communication:** Both RPC and RMI abstract the complexities of network communication, making remote calls as simple as local function calls.
 3. **Distributed Systems:** These mechanisms are crucial for building distributed systems where resources, computation, and data are spread across multiple machines.
-

Conclusion:

Client/Server communication using RPC and RMI enables distributed applications by allowing remote function/method calls over a network. While RPC is language-agnostic and can be used in different environments, RMI is tailored for Java, supporting remote object invocations in a seamless manner. These techniques simplify the development of distributed systems, ensuring that clients and servers can interact efficiently and transparently.

1.1

Example.proto

```
syntax = "proto3";
package example;
service RemoteService {
  rpc RemoteMethod (Request) returns (Response);
}
message Request {
  string message = 1;
}
message Response {
  string message = 1;
}
```

1.2

```
#Server.py
import grpc
import example_pb2
import example_pb2_grpc
from concurrent import futures

class RemoteServiceServicer(example_pb2_grpc.RemoteServiceServicer):
    def RemoteMethod(self, request, context):
        print(f"Server received: {request.message}")
        return example_pb2.Response(message=f"Received: {request.message}")

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    example_pb2_grpc.add_RemoteServiceServicer_to_server(RemoteServiceServicer(),
server)
    server.add_insecure_port(":::50051")
    server.start()
    print("Server started on port 50051.")
    server.wait_for_termination()

if __name__ == "__main__":
    serve()

# Client.py

import grpc
import example_pb2
```

```
import example_pb2_grpc

def run():
    with grpc.insecure_channel("localhost:50051") as channel:
        stub = example_pb2_grpc.RemoteServiceStub(channel)
        response = stub.RemoteMethod(example_pb2.Request(message="Hello, Server!"))
        print(f"Client received: {response.message}")

if __name__ == "__main__":
    run()
```

exp 2 Demonstrate a program for Inter-process communication

Inter-Process Communication (IPC): Short Theory

Inter-Process Communication (IPC) is a mechanism that allows processes (programs in execution) to communicate and share data with each other. In a multi-processing or distributed system, processes often need to exchange information to coordinate tasks, synchronize actions, and share resources.

IPC is critical for ensuring that processes can work together efficiently, even if they are running on different machines or in different environments.

Types of IPC Mechanisms:

1. **Message Passing:**
 - Involves processes communicating by sending and receiving messages. This can be done synchronously (blocking) or asynchronously (non-blocking).
 - **Examples:** Sockets, pipes, message queues.
 2. **Shared Memory:**
 - Allows multiple processes to access a common region of memory. This provides fast communication, but the processes need to manage synchronization to avoid data corruption.
 - **Examples:** Memory-mapped files, shared memory segments.
 3. **Remote Procedure Calls (RPC):**
 - A process on one machine can invoke a procedure on another machine as if it were a local call. RPC abstracts the complexity of network communication and makes remote calls appear similar to local function calls.
 4. **Semaphores and Mutexes:**
 - Used to manage access to shared resources in a synchronized manner, preventing race conditions. Semaphores allow signaling between processes, and mutexes lock resources to ensure that only one process accesses them at a time.
 5. **Signals:**
 - Signals are a limited form of IPC that allows one process to send a signal to another process, typically used for handling events like process termination or interruptions.
-

Why IPC is Important:

1. **Process Coordination:** IPC enables multiple processes to collaborate by sharing data and synchronizing their actions.
 2. **Data Sharing:** Allows processes to exchange data and resources, facilitating parallel processing and distributed computing.
 3. **Fault Tolerance:** IPC mechanisms can help ensure that processes can recover from failures by passing information to other processes in case of crashes.
 4. **Efficiency:** Proper use of IPC improves system efficiency by allowing processes to run concurrently and share resources.
-

Conclusion:

IPC is a crucial concept for enabling communication and collaboration between processes in modern computing systems. By using different IPC mechanisms such as message passing, shared memory, and synchronization primitives, systems can operate efficiently and handle complex distributed tasks.

write .java

```
package pkg_IPC;
```

```
import java.io.*;
```

```
public class write {
```

```
    public static void main(String[] args) {
```

```
        try {
```

```
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
```

```
            PrintWriter writer = new PrintWriter(new FileWriter("data.txt"));
```

```
            System.out.println("Enter text (type 'exit' to stop):");
```

```
            String input;
```

```
            while ((input = reader.readLine()) != null) {
```

```
                if (input.equalsIgnoreCase("exit")) break;
```

```
                writer.println(input);
```

```
                writer.flush();
```

```
            }
```

```
            reader.close();
```

```
            writer.close();
```

```
            System.out.println("Data written to data.txt");
```

```
        } catch (IOException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

read .java

```
package pkg_IPC;
```

```
import java.io.*;
```

```
public class read {
```

```
    public static void main(String[] args) {
```

```

try {
    BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
    String input;
    while ((input = reader.readLine()) != null) {
        System.out.println("Received: " + input);
    }
    reader.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

exp 3: develop a program for Group communication

Group Communication: Short Theory

Group Communication refers to the exchange of messages among a group of processes or nodes in a distributed system. It enables coordination, synchronization, and information sharing among multiple processes, allowing them to collaborate and perform tasks collectively. Group communication is essential for ensuring consistency, reliability, and fault tolerance in distributed applications.

Key Concepts in Group Communication:

1. Multicast:

- Multicast is a method of sending a message from one process to multiple processes in a group simultaneously. It helps reduce communication overhead compared to sending multiple unicast messages.

2. Reliability:

- Reliable group communication ensures that messages are delivered to all group members, even in the presence of failures or network issues. Mechanisms like **acknowledgements** or **retransmissions** are used to guarantee message delivery.

3. Atomicity:

- Atomic group communication ensures that messages sent to the group are either delivered to all members or none. This ensures consistency in cases where some processes might fail while processing the message.

4. Ordering:

- Ordering ensures that messages are delivered in the correct sequence. There are two main types of ordering:
 - **Total Order:** All processes receive the messages in the same order.
 - **causal Order**:** Messages are delivered respecting the causal relationships between them, ensuring that dependent messages are received in the correct order.

5. Fault Tolerance:

- Group communication protocols often include fault-tolerant mechanisms to handle process failures and ensure that the group can continue functioning even when some members fail.

Why Group Communication is Important:

1. **Coordination:** Group communication enables coordinated actions among distributed processes, which is essential for tasks like resource allocation, task distribution, or synchronization.
 2. **Consistency:** It ensures that all processes have a consistent view of the data or state of the system.
 3. **Reliability:** Ensures that communication within the group is reliable, even in the presence of network partitions or node failures.
-

Conclusion:

Group communication is vital for effective coordination and communication in distributed systems. By using multicast, ensuring reliability, and managing message ordering, it helps maintain consistency and fault tolerance, which are key to building robust distributed applications.

✅ Server.java – Multiclient Support & Clean Output

java

Copy

Edit

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
```

```
public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("🚀 Server started. Waiting for clients...");
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("✅ Client connected from: " +
                    clientSocket.getInetAddress().getHostName());

                PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
                out.println("Hi, this is Comp C41!");

                clientSocket.close(); // Closing client socket after sending message
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

/-----

✅ Client.java

java

Copy

Edit

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {

            String message = in.readLine();
            System.out.println(" 📄 Message from server: " + message);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
/-----
```

✅ Client1.java

java

Copy

Edit

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class Client1 {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {

            String message = in.readLine();
            System.out.println(" 📄 Message from server: " + message);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

output:

💡 How to Run:

Compile all:

bash

Copy

Edit

javac Server.java Client.java Client1.java

Run the server:

bash

Copy

Edit

java Server

Open two terminals and run:

bash

Copy

Edit

java Client

java Client1

Let me know if you want to enhance this with bi-directional communication or multi-threaded server support!

exp 4:Election (Bully Algorithm)

Election Algorithm: Short Theory

An **Election Algorithm** is a distributed algorithm used in a network of processes to **select a leader** (or coordinator) among them. The leader is responsible for making decisions, managing resources, or coordinating tasks in a system where there is no central authority.

Types of Election Algorithms:

1. Bully Algorithm:

- In the **Bully Algorithm**, the process with the highest ID in the system becomes the leader.
- When a process detects that the current leader has failed, it initiates an election by sending a message to processes with higher IDs. The highest ID process eventually wins the election and becomes the new leader.

2. Ring Algorithm:

- In the **Ring Algorithm**, processes are arranged in a logical ring.
- When a process detects a failure, it initiates an election by sending an **election message** around the ring. The process with the highest ID eventually wins and becomes the leader.

3. Centralized Election Algorithm:

- A single process (the coordinator) is responsible for initiating the election process. It can handle requests from other processes to select a leader based on some criterion (such as highest ID or priority).
-

Why Election Algorithms are Important:

- **Fault Tolerance:** They ensure that the system can recover and continue functioning if the current leader fails.
 - **Fairness:** Election algorithms ensure that all processes have an equal chance of becoming the leader, preventing monopolies.
 - **Coordination:** A leader is often needed to coordinate tasks, manage resources, or handle critical sections in distributed systems.
-

Conclusion:

Election algorithms play a vital role in maintaining leader-based coordination in distributed systems, ensuring that leadership is assigned fairly and robustly, even in the presence of failures.

Python Code for 3-Election (Bully Algorithm)

```
class Process:
```

```
    def __init__(self, pid):
        self.pid = pid
        self.active = True
```

```
    def __str__(self):
        return f"Process {self.pid} [{'Active' if self.active else 'Down'}]"
```

```
class BullyElection:
```

```
    def __init__(self, process_ids):
        self.processes = [Process(pid) for pid in process_ids]
        self.coordinator = max(self.processes, key=lambda p: p.pid)
```

```
    def display_status(self):
        for p in self.processes:
            print(p)
        print(f"\n👑 Current Coordinator: Process {self.coordinator.pid}\n")
```

```
    def deactivate_process(self, pid):
        for p in self.processes:
            if p.pid == pid:
                p.active = False
                print(f"⚠️ Process {pid} is now down.")
                if self.coordinator.pid == pid:
                    print("🚨 Coordinator is down! Election needed.")
                    break
```

```
    def hold_election(self, initiator_pid):
        initiator = next((p for p in self.processes if p.pid == initiator_pid), None)
        if not initiator or not initiator.active:
            print("❌ Initiator process not active.")
            return
```

```

print(f"\n🗳️ Election initiated by Process {initiator_pid}")
higher_processes = [p for p in self.processes if p.pid > initiator_pid and p.active]

if not higher_processes:
    self.coordinator = initiator
    print(f"✅ Process {initiator_pid} becomes the new coordinator!")
else:
    for p in higher_processes:
        print(f"✉️ Process {initiator_pid} sends election message to Process {p.pid}")
    max_proc = max(higher_processes, key=lambda p: p.pid)
    self.coordinator = max_proc
    print(f"👑 Process {max_proc.pid} becomes the new coordinator!")

def activate_process(self, pid):
    for p in self.processes:
        if p.pid == pid:
            p.active = True
            print(f"🔄 Process {pid} is now active.")
            break

```

Example usage

```

if __name__ == "__main__":
    system = BullyElection([1, 2, 3, 4, 5])
    system.display_status()

```

```

system.deactivate_process(5) # Simulate coordinator crash
system.hold_election(2)      # Process 2 initiates election
system.display_status()

```

/-----

🔍 Sample Output:

arduino

Copy

Edit

Process 1 [Active]

Process 2 [Active]

Process 3 [Active]

Process 4 [Active]

Process 5 [Active]

👑 Current Coordinator: Process 5

⚠️ Process 5 is now down.

🔔 Coordinator is down! Election needed.

🗳️ Election initiated by Process 2

✉️ Process 2 sends election message to Process 3

✉ Process 2 sends election message to Process 4

👑 Process 4 becomes the new coordinator!

4.2ring algorithm

🧠 What is Ring Election Algorithm?

Processes are arranged in a logical ring structure.

Election starts from any process and passes a message around the ring.

Each process adds its ID to the election message.

When the message returns to the initiator, the process with the highest ID becomes the coordinator.

🐍 Python Code: Ring Election Algorithm

```
class RingProcess:
    def __init__(self, pid):
        self.pid = pid
        self.active = True

    def __str__(self):
        return f"Process {self.pid} [{'Active' if self.active else 'Down'}]"

class RingElection:
    def __init__(self, process_ids):
        self.processes = [RingProcess(pid) for pid in process_ids]
        self.n = len(self.processes)
        self.coordinator = max(self.processes, key=lambda p: p.pid)

    def display_status(self):
        for p in self.processes:
            print(p)
        print(f"\n👑 Current Coordinator: Process {self.coordinator.pid}\n")

    def deactivate_process(self, pid):
        for p in self.processes:
            if p.pid == pid:
                p.active = False
                print(f"⚠ Process {pid} is now down.")
                if self.coordinator.pid == pid:
                    print("🚨 Coordinator is down! Election required.")
                    break

    def activate_process(self, pid):
        for p in self.processes:
```

```

    if p.pid == pid:
        p.active = True
        print(f"🔄 Process {pid} is now active.")
        break

def hold_election(self, initiator_pid):
    print(f"\n✅ Ring Election initiated by Process {initiator_pid}")

    if not any(p.pid == initiator_pid and p.active for p in self.processes):
        print("❌ Initiator is not active.")
        return

    index = next(i for i, p in enumerate(self.processes) if p.pid == initiator_pid)
    msg = []


    for i in range(self.n):
        curr_index = (index + i) % self.n
        curr_proc = self.processes[curr_index]
        if curr_proc.active:
            msg.append(curr_proc.pid)
            print(f"🔄 Passing through Process {curr_proc.pid}")

    new_coordinator_pid = max(msg)
    self.coordinator = next(p for p in self.processes if p.pid == new_coordinator_pid)
    print(f"👑 New Coordinator is Process {self.coordinator.pid}")

# Example Usage
if __name__ == "__main__":
    system = RingElection([1, 2, 3, 4, 5])
    system.display_status()

    system.deactivate_process(5)
    system.hold_election(2) # Process 2 initiates the election
    system.display_status()

```

/-----
 Sample Output:

```

pgsql
Copy
Edit
Process 1 [Active]
Process 2 [Active]
Process 3 [Active]
Process 4 [Active]
Process 5 [Active]

```

👑 Current Coordinator: Process 5

⚠️ Process 5 is now down.

🔥 Coordinator is down! Election required.

🗑️ Ring Election initiated by Process 2

🔄 Passing through Process 2

🔄 Passing through Process 3

🔄 Passing through Process 4

🔄 Passing through Process 1

👑 New Coordinator is Process 4

Want me to show a graphical representation of the ring or simulate it using threads and sockets? Let me know!

exp 5 : Develop a program for clock synchronization algorithms

Clock Synchronization Algorithms: Short Theory

Clock Synchronization is crucial in distributed systems where multiple processes or nodes, often on different machines, need to maintain a consistent view of time. The goal is to ensure that all nodes in the system have synchronized clocks, allowing them to operate in a coordinated and predictable manner.

Types of Clock Synchronization Algorithms:

1. Network Time Protocol (NTP):

- NTP is one of the most widely used clock synchronization protocols. It uses a hierarchical system of time sources (servers) to synchronize clocks across a network.
- NTP typically operates on the client-server model, where the client synchronizes its clock by querying a time server.

2. Berkeley Algorithm:

- This is a **centralized** algorithm where one process (usually the master) acts as the coordinator to synchronize the clocks of all other nodes in the system.
- The master process sends its clock time to all other nodes, and they adjust their clocks based on the difference with the master.

3. Cristian's Algorithm:

- In this algorithm, a client synchronizes its clock by requesting the current time from a time server.
- The server sends the current time, and the client adjusts its clock based on the round-trip delay of the request.

4. Lamport's Logical Clocks:

- This is not an actual physical clock synchronization algorithm but ensures **logical consistency** of events in distributed systems.
 - Every process maintains a logical clock that is updated on every event. The clocks are synchronized based on causal relationships between events rather than real-time.
-

Why Clock Synchronization is Important:

- **Consistency:** Ensures that all processes have a common time reference.
 - **Event Ordering:** Helps in ordering events in distributed systems and ensuring the correct sequence of operations.
 - **Coordination:** Essential for algorithms that require coordination between nodes, such as mutual exclusion or distributed transactions.
-

Conclusion:

Clock synchronization is essential in distributed systems to ensure that all processes have a consistent view of time, allowing the system to function correctly. Algorithms like NTP, Berkeley, and Cristian's help achieve synchronization in different ways, balancing between accuracy and complexity.



Berkeley Algorithm Overview:

One node acts as the master (server).

Master polls all slave clocks, collects their times.

Calculates the average time (ignoring faulty ones).

Sends time adjustments to slaves.



Python Code: Clock Synchronization (Berkeley Algorithm)

python

Copy

Edit

```
import random
```

```
class Node:
```

```
    def __init__(self, node_id, clock_time):
        self.node_id = node_id
        self.clock_time = clock_time # In seconds
        self.offset = 0
```

```
    def __str__(self):
        return f"Node {self.node_id}: Clock = {self.clock_time + self.offset:.2f} sec"
```

```
class BerkeleyClockSync:
```

```
    def __init__(self, nodes):
        self.nodes = nodes
        self.master = nodes[0] # First node is the master
```

```
    def synchronize(self):
        print("🕒 Clock Times Before Synchronization:")
        for node in self.nodes:
            print(node)
```

```
        total_time = 0
```

```

active_nodes = 0
offsets = []

# Master collects time differences
for node in self.nodes:
    if node != self.master:
        delay = random.uniform(0.1, 0.5) # Simulate network delay
        difference = node.clock_time - self.master.clock_time - delay
        offsets.append((node, difference))
        total_time += node.clock_time
        active_nodes += 1

# Include master's time too
total_time += self.master.clock_time
active_nodes += 1
average_time = total_time / active_nodes

# Set master offset
self.master.offset = average_time - self.master.clock_time

# Adjust other nodes
for node, diff in offsets:
    node.offset = average_time - node.clock_time

print("\n✅ Clock Times After Synchronization:")
for node in self.nodes:
    print(node)

```

Example Usage

```
if __name__ == "__main__":
```

```


    nodes = [
        Node(1, 10.5),
        Node(2, 12.8),
        Node(3, 9.6),
        Node(4, 11.0)
    ]

```

```

sync = BerkeleyClockSync(nodes)
sync.synchronize()

```

 Output Sample:

mathematica

Copy

Edit

 Clock Times Before Synchronization:

Node 1: Clock = 10.50 sec

Node 2: Clock = 12.80 sec

Node 3: Clock = 9.60 sec

Node 4: Clock = 11.00 sec

✅ Clock Times After Synchronization:

Node 1: Clock = 10.98 sec

Node 2: Clock = 10.98 sec

Node 3: Clock = 10.98 sec

Node 4: Clock = 10.98 sec

🔧 Notes:

The current version simulates delay and synchronization logic.

You can add actual socket communication for a real distributed version.

For fun, try giving random clocks each time for realism.

exp 6: Token Based Algorithm

implement the **Token-Based Algorithm** — it's commonly used for **mutual exclusion** in **distributed systems**.

💡 **What is the Token-Based Algorithm?**

- Only the **node holding the token** is allowed to enter the **critical section** (e.g., access a shared file).
- The token is **passed** in a logical ring.
- If a node wants to enter the critical section, it **waits** for the token.

🐍 Python Code: Token-Based Algorithm (Ring Topology)

```
```python
import time

class Node:
 def __init__(self, node_id):
 self.node_id = node_id
 self.has_token = False
 self.request_CS = False # Wants to enter critical section

 def __str__(self):
 status = "has token" if self.has_token else "no token"
 return f"Node {self.node_id} [{status}]"

 def enter_critical_section(self):
 print(f"🔒 Node {self.node_id} is entering the critical section...")
 time.sleep(1) # Simulate time spent in critical section
 print(f"🔓 Node {self.node_id} has exited the critical section.")
```



```

class TokenRing:
 def __init__(self, total_nodes):
 self.nodes = [Node(i) for i in range(total_nodes)]
 self.token_index = 0
 self.nodes[self.token_index].has_token = True # Token starts with node 0

 def request_critical_section(self, node_id):
 self.nodes[node_id].request_CS = True

 def pass_token(self):
 current_node = self.nodes[self.token_index]
 if current_node.request_CS:
 current_node.enter_critical_section()
 current_node.request_CS = False

 # Pass token to next node
 current_node.has_token = False
 self.token_index = (self.token_index + 1) % len(self.nodes)
 next_node = self.nodes[self.token_index]
 next_node.has_token = True
 print(f"🔄 Token passed to Node {next_node.node_id}")

 def show_status(self):
 for node in self.nodes:
 print(node)
 print()

Example Usage
if __name__ == "__main__":
 ring = TokenRing(5)
 ring.show_status()

 # Request CS from Node 2 and Node 4
 ring.request_critical_section(2)
 ring.request_critical_section(4)

 # Run the ring for a few rounds
 for _ in range(10):
 ring.pass_token()
 time.sleep(0.5)

```

---










#### **Sample Output:**

```

Node 0 [has token]
Node 1 [no token]
Node 2 [no token]
Node 3 [no token]

```

Node 4 [no token]

-  Token passed to Node 1
-  Token passed to Node 2
-  Node 2 is entering the critical section...
-  Node 2 has exited the critical section.
-  Token passed to Node 3
-  Token passed to Node 4
-  Node 4 is entering the critical section...
-  Node 4 has exited the critical section.
-  Token passed to Node 0




...

---

#### Features:

- Easy to simulate mutual exclusion.
  - Simple round-robin logic.
  - Only **one process accesses** critical section at a time (due to the token).
- 

Let me know if you want:

- GUI simulation 
- Multi-threaded version using threading 
- Token loss detection & recovery 

## ## 6.1 Token-Based Algorithm (Message Passing)

```python

ure! Here's another version of the Token-Based Algorithm — this time, implemented using message passing for mutual exclusion. This is a simplified version of how processes can use tokens to ensure only one process enters the critical section at a time.

Token-Based Algorithm (Message Passing)

In this approach:

Each node sends a request message to the next node to request the token.

Only one node can hold the token at a time and enter the critical section.

Python Code: Token-Based Algorithm with Message Passing

python

Copy

Edit

```
import random
```

```
import time
```

```
class Node:
```

```
    def __init__(self, node_id, ring):
```

```

        self.node_id = node_id
        self.ring = ring
        self.token = False # Initially, no token
        self.request_CS = False # Flag to request critical section
        self.in_critical_section = False # Flag for critical section status

    def __str__(self):
        return f"Node {self.node_id} [{'Token' if self.token else 'No Token'}]"

    def request_critical_section(self):
        """Request for the critical section."""
        print(f"Node {self.node_id} is requesting the critical section.")
        self.request_CS = True

    def enter_critical_section(self):
        """Enter the critical section."""
        if self.token and self.request_CS:
            self.in_critical_section = True
            print(f"Node {self.node_id} is entering the critical section.")
            time.sleep(1) # Simulating work in critical section
            print(f"Node {self.node_id} has exited the critical section.")
            self.token = False # Pass token after leaving critical section

    def pass_token(self):
        """Pass token to next node."""
        if self.token and not self.in_critical_section:
            print(f"Node {self.node_id} has the token and will pass it.")
            self.token = False
            next_node = self.ring[(self.node_id + 1) % len(self.ring)] # Passing token in the ring
            next_node.token = True
            print(f"Token passed to Node {next_node.node_id}")
            return next_node
        return self

    def receive_token(self):
        """Receive token if available."""
        if self.request_CS and not self.in_critical_section:
            print(f"Node {self.node_id} has received the token.")
            self.enter_critical_section()

class TokenRing:
    def __init__(self, total_nodes):
        self.ring = [Node(i, self) for i in range(total_nodes)]
        self.token_index = 0
        self.ring[self.token_index].token = True # Initially, Node 0 holds the token

    def show_status(self):

```

```

        """Display the status of each node in the ring."""
        for node in self.ring:
            print(node)
        print()

    def start_token_pass(self):
        """Start passing the token around the ring."""
        for _ in range(10): # Run for 10 rounds
            current_node = self.ring[self.token_index]
            current_node = current_node.pass_token()
            time.sleep(1) # Delay between token passing

# Example Usage
if __name__ == "__main__":
    ring = TokenRing(5)
    ring.show_status()

    # Request CS for Nodes 1 and 4
    ring.ring[1].request_critical_section()
    ring.ring[4].request_critical_section()

```

```

    # Start passing the token around the ring
    ring.start_token_pass()

```

 Sample Output:

```

css
Copy
Edit
Node 0 [Token]
Node 1 [No Token]
Node 2 [No Token]
Node 3 [No Token]
Node 4 [No Token]

```

Node 1 is requesting the critical section.
Node 4 is requesting the critical section.

Node 0 has the token and will pass it.
Token passed to Node 1
Node 1 has received the token.
Node 1 is entering the critical section...
Node 1 has exited the critical section.
Node 1 has the token and will pass it.
Token passed to Node 2
...

 Key Features:

Message Passing: Tokens are passed between nodes in the ring.

Critical Section Access: A node enters the critical section only when it has the token and has requested access.

Simple Design: This version uses a direct message-passing style to simulate the process of passing the token and requesting critical section access.

Would you like to add failure handling for token loss or node crash recovery? Let me know!

Token-Based Algorithm: Theory

The **Token-Based Algorithm** is a synchronization method used in distributed systems to manage access to shared resources or critical sections (CS). The primary objective is to ensure **mutual exclusion**, meaning that only one process or node can access a shared resource at any given time. This algorithm is particularly important in distributed computing, where there is no central coordinator.

Key Concepts:

1. **Token:**
 - A **token** is a special message or signal passed around the system. The process or node that holds the token is allowed to enter the critical section and perform operations that require exclusive access to a shared resource.
 - The token is passed from one process to another in a specific order (often circular, in the case of token rings).
 - **Token Passing** ensures that only one process holds the token at any time, thus preventing multiple processes from simultaneously accessing the critical section.
 2. **Mutual Exclusion:**
 - The main goal of the token-based algorithm is to ensure **mutual exclusion**. This means that only one process is allowed to execute its critical section at a time, and others must wait until the token is passed to them.
 3. **Distributed System:**
 - In a distributed system, processes or nodes may be located on different machines or processors. The **token** helps to synchronize access to resources without needing a central authority to control the system.
 4. **Token Circulation:**
 - The token is passed in a **logical order** between the nodes. The passing of the token ensures that every process gets a turn to access the critical section.
 - Once a process finishes its task in the critical section, it passes the token to the next process in the sequence.
-

Types of Token-Based Algorithms:

1. **Token Ring Algorithm:**
 - In this algorithm, nodes are arranged in a **logical ring**. Each node has a direct link to the next node, and the token circulates around the ring.
 - When a process wants to access the critical section, it must first wait until it receives the token. After finishing its task, the process passes the token to the next node.
2. **Token Bus Algorithm:**

- Similar to the token ring algorithm, but instead of a ring, processes are connected in a **bus structure**. The token is passed along the bus, and only the process that holds the token can access the critical section.
3. **Centralized Token Algorithm:**
- A single node (often called the **coordinator**) is responsible for distributing the token to other nodes. The coordinator ensures that the token is passed around efficiently and that no two processes access the critical section simultaneously.
 - This approach is typically easier to implement but may create a single point of failure if the coordinator node crashes.
-

Working of the Token-Based Algorithm:

1. **Initialization:**
 - A token is created and given to one of the nodes (or processes) in the system.
 2. **Request for Critical Section:**
 - When a process (node) wants to enter the critical section, it waits for the token. It cannot proceed until it holds the token.
 3. **Entering Critical Section:**
 - Once a process receives the token, it enters the critical section and performs its work (e.g., updating a shared resource).
 4. **Exiting Critical Section:**
 - After completing its work, the process passes the token to the next node in the sequence.
 - Passing the token ensures that only one process can access the critical section at any time.
 5. **Token Passing:**
 - The token circulates around the system in a predefined order. The order of token passing can be determined by a **ring topology**, **bus topology**, or any other structure.
 - The token passing mechanism is designed to prevent deadlock, ensure fairness, and avoid starvation, meaning that all processes will eventually receive the token and be able to access the critical section.
-

Advantages of Token-Based Algorithm:

1. **Simplicity:**
 - The token-based algorithm is conceptually simple and easy to implement in distributed systems.
2. **Decentralization:**
 - It does not require a central coordinator, especially in ring-based systems. This makes the system more fault-tolerant.
3. **Fairness:**
 - The algorithm ensures fairness as every process gets an opportunity to access the critical section in a predictable order.
4. **Deadlock-Free:**
 - As long as the token is not lost, the algorithm guarantees that deadlock will not occur, since only one process can hold the token at a time.
5. **Efficiency:**

- The token is passed from process to process, minimizing communication overhead once the initial setup is done.

Disadvantages of Token-Based Algorithm:

1. **Token Loss:**
 - If the token is lost (due to a crash or network failure), the system can become stuck. Recovery mechanisms are needed to regenerate or rediscover the token.
2. **Overhead of Passing Token:**
 - Passing the token around can introduce some overhead, especially in large systems with many processes. The time taken for the token to reach a process might increase as the system grows.
3. **Single Point of Failure:**
 - If there is a failure in the communication network or the node holding the token, the system can break down unless recovery mechanisms are implemented.
4. **Not Suitable for High-Contention Environments:**
 - In scenarios with high contention for the critical section, the token-based approach might not be as efficient as other synchronization techniques like **centralized locking** or **lamport's bakery algorithm**.

Example of Token-Based Algorithm:

Consider a distributed system with five nodes (Node 1 to Node 5) that need to access a shared resource (e.g., database). Each node must wait for the token to enter the critical section.

- The nodes are arranged in a **logical ring**. The token starts at **Node 1**.
- **Node 1** needs to access the critical section, so it keeps the token. Once finished, it passes the token to **Node 2**.
- **Node 2** proceeds with its task and passes the token to **Node 3**, and so on.
- After all nodes have processed their requests, the token completes one full circle and starts over from **Node 1**.

Conclusion:

The **Token-Based Algorithm** is a highly effective and fair method for managing mutual exclusion in distributed systems. By passing a token between nodes, the system ensures that only one process can access the critical section at any time. While it is simple and robust, special care must be taken to handle token loss and system failures, especially in large, dynamic systems.

Would you like me to implement this theory in code or explore another related topic?

exp 7 Load balancing

Load Balancing Algorithm: Overview

****Load balancing**** is a crucial technique in distributed systems to efficiently distribute workloads across multiple resources (e.g., servers, processors). The goal is to optimize resource usage, minimize response time, and avoid overloading any single resource.

There are several load balancing algorithms, and the two most popular ones are:

1. **Round Robin (RR)**: Distributes tasks to servers in a circular manner.
2. **Least Connections**: Directs traffic to the server with the least active connections.
3. **Weighted Round Robin**: An enhanced version of round-robin where each server is assigned a weight (based on its capacity).

Basic Theory Behind Load Balancing:

- **Objective**: Distribute incoming tasks (requests) efficiently among multiple servers or resources.
- **Dynamic Load Balancing**: Adapts to the system's state by considering factors like load, server capacity, etc.
- **Static Load Balancing**: Uses predetermined methods like Round Robin without considering the current load of servers.

Common Load Balancing Algorithms:

1. **Round Robin**:
 - It assigns tasks to servers in a **circular order**.
 - Simplicity: Easy to implement, but does not take into account server load, so may not always be optimal.
2. **Least Connections**:
 - It directs traffic to the **server with the least active connections**.
 - Ideal for environments where tasks take varying amounts of time to complete.
3. **Weighted Round Robin**:
 - Servers are assigned **weights** based on their capacity.
 - Servers with higher weights receive more requests.
 - Good for scenarios where some servers have more processing power than others.

Python Code: Load Balancing Using Round Robin

Here's an implementation of a simple **Round Robin Load Balancing Algorithm** in Python. This example will simulate distributing incoming requests to multiple servers.

Python Code: Round Robin Load Balancing

```
python
import time

class Server:
    def __init__(self, server_id):
        self.server_id = server_id
```



```

        self.load = 0 # Number of requests handled by the server

def __str__(self):
    return f"Server {self.server_id} [Load: {self.load}]"

def process_request(self):
    """Simulate processing a request."""
    self.load += 1
    print(f"Server {self.server_id} is processing request. Total load: {self.load}")
    time.sleep(0.5) # Simulate processing time

class LoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.index = 0

    def get_next_server(self):
        """Select the next server in round-robin fashion."""
        server = self.servers[self.index]
        self.index = (self.index + 1) % len(self.servers) # Ensure circular rotation
        return server

    def distribute_requests(self, num_requests):
        """Distribute incoming requests to servers using Round Robin."""
        print(f"\nDistributing {num_requests} requests:")
        for _ in range(num_requests):
            server = self.get_next_server()
            server.process_request()

    def show_server_status(self):
        """Show the current load on each server."""
        print("\nServer Status:")
        for server in self.servers:
            print(server)

# Example Usage
if __name__ == "__main__":
    servers = [Server(i) for i in range(1, 4)] # 3 servers
    load_balancer = LoadBalancer(servers)

    load_balancer.show_server_status()

    # Simulate distributing 10 requests to the servers
    load_balancer.distribute_requests(10)

    # Show final load on each server
    load_balancer.show_server_status()

```

Explanation of the Code:

1. Server Class:

- Each Server object represents a server in the load balancer.
- It has a load attribute that tracks the number of requests handled by the server.
- The process_request() method simulates the server processing a request.

2. LoadBalancer Class:

- Manages a list of Server objects and distributes requests using the **Round Robin** algorithm.
- The get_next_server() method rotates through the servers in a circular manner.
- The distribute_requests() method simulates the distribution of requests to the servers.

3. Simulation:

- We create 3 servers and distribute 10 requests among them using Round Robin.
- The load on each server is updated as requests are processed.

Sample Output:

Server Status:

Server 1 [Load: 0]

Server 2 [Load: 0]

Server 3 [Load: 0]

Distributing 10 requests:

Server 1 is processing request. Total load: 1

Server 2 is processing request. Total load: 1

Server 3 is processing request. Total load: 1

Server 1 is processing request. Total load: 2

Server 2 is processing request. Total load: 2

Server 3 is processing request. Total load: 2

Server 1 is processing request. Total load: 3

Server 2 is processing request. Total load: 3

Server 3 is processing request. Total load: 3

Server 1 is processing request. Total load: 4

Server Status:

Server 1 [Load: 4]

Server 2 [Load: 3]

Server 3 [Load: 3]

Weighted Round Robin Load Balancing

For **Weighted Round Robin**, we can assign weights to the servers based on their processing power. Servers with higher weights will receive more requests.

Code: Weighted Round Robin Load Balancing

```
import time
```

```
class WeightedServer:
```

```

def __init__(self, server_id, weight):
    self.server_id = server_id
    self.weight = weight # Weight based on server capacity
    self.load = 0

def __str__(self):
    return f"Server {self.server_id} [Weight: {self.weight}, Load: {self.load}]"

def process_request(self):
    """Simulate processing a request."""
    self.load += 1
    print(f"Server {self.server_id} (Weight {self.weight}) is processing request. Total load: {self.load}")
    time.sleep(0.5)

class WeightedLoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.index = 0
        self.total_weight = sum(server.weight for server in self.servers)

    def get_next_server(self):
        """Select the next server based on weights."""
        server = self.servers[self.index]
        self.index = (self.index + 1) % len(self.servers)
        return server

    def distribute_requests(self, num_requests):
        """Distribute incoming requests to servers using Weighted Round Robin."""
        print(f"\nDistributing {num_requests} requests (Weighted Round Robin):")
        for _ in range(num_requests):
            server = self.get_next_server()
            server.process_request()

    def show_server_status(self):
        """Show the current load on each server."""
        print("\nServer Status:")
        for server in self.servers:
            print(server)

# Example Usage
if __name__ == "__main__":
    servers = [WeightedServer(i, weight) for i, weight in enumerate([2, 1, 3], start=1)] # 3
    servers with weights
    load_balancer = WeightedLoadBalancer(servers)

    load_balancer.show_server_status()

```

```
# Simulate distributing 10 requests to the servers
load_balancer.distribute_requests(10)
```

```
# Show final load on each server
load_balancer.show_server_status()
```

Explanation of the Code:

- **WeightedServer Class:** Similar to the Server class but includes a weight attribute.
- **WeightedLoadBalancer Class:** Implements the Weighted Round Robin algorithm where the server with a higher weight gets more requests.

The requests are processed in a way that **servers with higher weights** will handle more requests.

Conclusion:

- **Round Robin** is simple and works well when all servers are identical.
- **Weighted Round Robin** is more flexible and works well when servers have different capacities.

Would you like to explore more advanced **load balancing algorithms**, like **Least Connections** or **Randomized Load Balancing**?