TokenRing – Ass-5

In distributed systems, the Token Ring is a communication protocol and network topology that allows multiple nodes to communicate in a sequential manner using a token. The token is a special message or control token that circulates among the nodes in a ring-like fashion.

The token circulates in a predetermined order, and only the node possessing the token has the right to transmit data over the network. When a node receives the token, it can either keep it and transmit its own data or pass it along to the next node in the ring. Each node checks the token as it passes by and takes appropriate actions based on its state and requirements.

The Token Ring protocol ensures that each node has a fair opportunity to transmit its data by following a strict sequence defined by the ring topology. It prevents conflicts and ensures that data transmission occurs in an orderly fashion.

The provided code implements a simplified version of the Token Ring protocol for a distributed system. It simulates multiple processes or nodes communicating in a sequential manner by passing a token.

Here's a summary of the code:

1. The number of processes is defined as **N**, and the token value is set as **-1**. The **CS\_TIME** variable represents the time spent in the critical section.
2. The **hasToken** array keeps track of whether each process has the token, and the **inCS** array indicates if a process is in the critical section. The **tokenHolder** variable stores the current token holder's index.
3. Each process runs in an infinite loop. If a process has the token (**hasToken[id]** is true), it enters the critical section, performs some actions (represented by a delay using **Thread.sleep()**), and then releases the token.
4. When a process exits the critical section, it releases the token by setting its own **hasToken** value to false and passing the token to the next process in a circular manner.
5. If a process does not have the token, it waits for some time before checking again.
6. The **main** method initializes the token holder as process 0, starts threads for each process, and waits for the threads to finish execution.

In short, this code simulates a Token Ring-based system where multiple processes take turns entering a critical section to perform some actions. The token is passed among the processes in a circular manner, ensuring fair access to the critical section.

Bully Algorithm – Ass -6

1. Bully Algorithm: The Bully Algorithm is a leader election algorithm where processes in a distributed system determine and elect a leader among themselves. The algorithm works as follows:

* Each process has a unique ID, and the process with the highest ID is considered the leader.
* When a process detects the absence of a leader (e.g., due to a failure or crash), it initiates an election by sending election messages to processes with higher IDs.
* If a process receives an election message, it compares its own ID with the sender's ID. If the sender has a lower ID, it acknowledges the election and does not participate further.
* If a process receives an election message and has a higher ID, it takes over as the leader and sends a coordinator message to inform other processes about its new leadership.
* If a process does not receive any response after initiating an election, it assumes that it is the leader and sends coordinator messages to other processes.

The Bully Algorithm ensures that the process with the highest ID becomes the leader. It guarantees the election of a new leader when the existing leader fails, but it requires communication between processes and assumes that messages are reliably delivered.

The provided code implements a simplified version of the Bully Algorithm for leader election in a distributed system. It allows processes to bring themselves up or down, send messages, and elect a coordinator. Here's a summary of the code:

* The code defines a class named "Bully" that contains methods for bringing a process up (**up()**), bringing a process down (**down()**), and sending a message (**mess()**).
* The state of each process is represented by the **state** boolean array, where **true** indicates that a process is up, and **false** indicates that it is down.
* The coordinator is represented by the **coordinator** variable, which is initially set to process 5 (**state[4]**).
* The **up()** method brings a process up by setting its corresponding **state** value to **true**. It initiates an election by sending election messages to processes with higher IDs and checks for an alive message from higher-ID processes.
* The **down()** method brings a process down by setting its corresponding **state** value to **false**.
* The **mess()** method sends a message from a process to initiate an election. If the process is up and the coordinator is down, it starts a new election by sending election messages to higher-ID processes and waits for a coordinator message.
* The **main()** method provides a menu-driven interface for interacting with the processes. It allows bringing processes up or down, sending messages, and exiting the program.

In summary, this code demonstrates the basic functionality of the Bully Algorithm for leader election. It allows processes to join or leave the system, sends messages to initiate elections, and elects a new coordinator based on the process IDs.

Ring Alogorithm – ass -6

1. Ring Algorithm: The Ring Algorithm is a distributed coordination algorithm that uses a logical ring topology to coordinate actions among processes. The algorithm works as follows:

* Processes are organized in a logical ring, where each process is connected to its neighboring processes.
* A token or message circulates around the ring, and only the process holding the token has the right to perform certain actions.
* When a process receives the token, it can either execute its own task or forward the token to the next process in the ring.
* Each process checks the token as it passes through and takes appropriate actions based on its state or requirements.
* The token continues to circulate until all processes have completed their tasks or until a termination condition is met.

The Ring Algorithm ensures that processes in the ring take turns executing their tasks in a sequential manner. It provides a fair and distributed approach for coordination, but it requires the entire ring to operate correctly, and failures can disrupt the coordination process.

Both algorithms are widely used in distributed systems to handle leadership and coordination tasks, and each algorithm has its own advantages and considerations depending on the specific system requirements and characteristics.

The provided code implements the Ring Algorithm for leader election in a distributed system. Here's a summary of the code:

* The code defines a class named "Ring" that contains the main method.
* It also defines a class named "Rr" to represent the individual processes in the system. Each process has an index, id, a flag (**f**), and a state (active or inactive).
* The main method initializes an array of Rr objects to represent the processes.
* It prompts the user to enter the number of processes and their IDs. The processes are sorted based on their IDs.
* One process is marked as inactive, and it is assumed to be the coordinator.
* The program enters a loop where the user can choose to initiate an election or quit.
* When an election is initiated, the user selects a process to start the election. Messages are then passed in a ring-like manner from one process to another until they reach the process that initially started the election.
* The IDs of the processes that receive the messages are stored in an array.
* Finally, the process with the highest ID among the received messages is selected as the new coordinator.

In summary, this code demonstrates the basic functionality of the Ring Algorithm for leader election. It allows processes to communicate in a ring-like structure to elect a coordinator based on their IDs.

Ass- 1

RMI (Remote Method Invocation) is a distributed computing technology in Java that enables communication and interaction between objects in a distributed system. It allows Java objects running in different Java Virtual Machines (JVMs) to invoke methods on remote objects, as if they were local objects.

In a distributed system, RMI provides a mechanism for remote objects to communicate with each other by making remote method calls. RMI abstracts away the complexities of network communication, serialization, and remote object activation, allowing developers to focus on the logic and functionality of their distributed applications.

1. Client.java: This code represents the client-side application. It prompts the user to enter the server address, two strings, and communicates with the server using RMI to concatenate the strings together.
2. Servant.java: This code defines the Servant class, which acts as the server-side implementation of the remote interface. It extends the UnicastRemoteObject class and implements the ServerInterface interface. The **concat** method in this class simply concatenates two strings and returns the result.
3. ServerInterface.java: This interface defines the methods that the server provides to clients. In this case, it includes a single method called **concat**, which takes two strings as parameters and returns their concatenation.
4. Server.java: This code represents the server application. It creates an instance of the Servant class, binds it to a specific name ("Server"), and makes it available for clients to connect to using RMI.

In summary, these codes establish a client-server communication model where the client sends requests to the server to concatenate two strings, and the server performs the concatenation operation and returns the result to the client. RMI is used to facilitate the communication between the client and server processes.

Ass – 4

The Berkeley algorithm is a clock synchronization algorithm used in distributed systems to coordinate the clocks of multiple machines or processes. It was developed by researchers at the University of California, Berkeley.

In a distributed system, different machines or processes may have their own clocks that are not perfectly synchronized. Clock synchronization is important for various reasons, such as timestamping events accurately, ordering events in distributed systems, and coordinating distributed algorithms.

The Berkeley algorithm follows a master-slave model, where one machine is designated as the master and the rest are slaves.

Clock synchronization algorithms are used in distributed systems to ensure that the clocks of different machines or processes are closely aligned with each other. This alignment is important for various applications and protocols that rely on accurate timekeeping across multiple devices.

The code you provided implements the Berkeley algorithm, which is a clock synchronization algorithm used in distributed systems. Here's a summary of what the code does:

1. The main method listens for incoming client connections on port 2000 using a ServerSocket.
2. When a client connects, a new thread is created to handle the client's request.
3. The ClientHandler class implements the logic for each client request.
4. In the run method of the ClientHandler, it receives a time request from the client, representing the client's local time.
5. It retrieves the current time from the server using **System.currentTimeMillis()** and sends it back to the client.
6. It calculates the clock difference between the client's request time and the server's current time.
7. It establishes a connection to a server (in this case, the same local machine) and sends the clock difference to the server.
8. The server receives the clock difference and calculates the average clock difference across all clients.
9. The server sends back the average clock difference to the client.
10. The client adjusts its local clock by adding the average clock difference received from the server.
11. The adjusted time is printed to the console.
12. The sockets are closed, and the process repeats for the next client connection.

Overall, the code allows multiple clients to connect and synchronize their clocks using the Berkeley algorithm by exchanging clock differences with a central server.