Report

Submitted By

Anjali Singh

M.Tech Cyber Security

2023JCS2565

Submitted To

Mr. Huzur Saran

Indian Institute Of Technology, Delhi

1. **The Code for the given problem of total-order-multicast is:**

```
import threading
import time
from queue import Queue, Empty

class MessageManager:

    def __init__(self, num_recipients):
        self.local_clock = 0
        self.message_queue = []
        self.commit_queues = {recipient: Queue() for recipient
           in range(num_recipients)}
        self.should_stop = False
        self.message_order = []

    def send_unordered_message(self, content, recipient):
        print(f"Sending an unordered message: '{content}' to
           recipient: {recipient}")

    def send_total_order_message(self, content, recipients):
        self.local_clock += 1
        message = (content, self.local_clock, recipients)
        self.message_queue.append(message)
        self.broadcast_ack(message)

    def broadcast_ack(self, message):
        acks = {}
        for recipient in message[2]:
            ack = self.receive_ack(recipient)
            acks[recipient] = ack

        if all(ack is not None for ack in acks.values()):
            max_time = max(ack[1] for ack in acks.values())
            for recipient in message[2]:
                self.commit_queues[recipient].put((message[0],
                   max_time))

    def receive_ack(self, recipient):
        time.sleep(0.1)  # Simulate network delay
        ack_time = self.local_clock
        return (recipient, ack_time)

    def commit_messages(self, recipient):
        commit_queue = self.commit_queues[recipient]
        while not self.should_stop:
            try:
                message, max_time = commit_queue.get(timeout=1)
            except Empty:
                continue

            if self.local_clock < max_time:
                self.local_clock = max_time
            self.local_clock += 1
            self.message_order.append((message, self.
               local_clock))
            print(f"Committing message: '{message}' at time {
               self.local_clock}")

    def stop_commit_threads(self):
        self.should_stop = True
        self.message_order.sort(key=lambda x: x[1])
        for message, _ in self.message_order:
            print(f"Total order message received: '{message}'")
    def print_total_order_messages(self):
        while not self.should_stop:
            if len(self.message_order) > 0:
                message, _ = self.message_order.pop(0)
                print(f"Total order message received: '{message
                   }'")

if __name__ == "__main__":
    num_recipients = int(input("Enter the number of recipients:
        "))
    num_total_order_messages = int(input("Enter the number of
       total order messages to send: "))

    manager = MessageManager(num_recipients)

    for i in range(num_recipients):
        manager.send_unordered_message(f"Unordered Message {i}"
           , i % num_recipients)

    threads = []

    for i in range(num_recipients):
        t = threading.Thread(target=manager.commit_messages,
           args=(i,))
        threads.append(t)
        t.start()

    for i in range(num_total_order_messages):
        recipients = [j % num_recipients for j in range(
           num_recipients)]
        manager.send_total_order_message(f"Message {i}",
           recipients)
    print_thread = threading.Thread(target=manager.
        print_total_order_messages)
    print_thread.start()

    manager.stop_commit_threads()

    for t in threads:
        t.join()

    print_thread.join()
```

2. **The pseudo-code for the above implemented python code is:**

   - **MessageManager Class:**
     - The central component of the code is the `MessageManager` class, responsible for managing and ensuring total order message delivery among multiple recipients.
   - **Initialization:**
     - The class is initialized with the number of recipients. It sets up various data structures:
       - `local_clock` to maintain a local timestamp.
       - `message_queue` to store messages that need total order delivery.
       - `commit_queues` to maintain individual commit queues for each recipient.
       - `should_stop` to control when threads should stop processing.
       - `message_order` to maintain the total order of received messages.
   - **Sending Unordered Messages:**
     - The `send_unordered_message` method simulates sending messages without

any order guarantees. It prints the content of the message and the recipient.

- **Sending Total Order Messages:**
  - The `send_total_order_message` method sends messages that must be delivered in total order. It:
    - Increments the local clock to assign a timestamp.
    - Creates a message tuple with content, timestamp, and the list of recipients.
    - Appends this message to the `message_queue`.
    - Calls `broadcast_ack` to acknowledge the message.
- **Broadcasting ACKs:**
  - The `broadcast_ack` method is responsible for ensuring acknowledgments are received from all recipients before messages are committed.
    - It uses a dictionary to track acknowledgments received from each recipient.
    - If acknowledgments are received from all recipients, it calculates the maximum timestamp among acknowledgments and sends the message to the commit queues of each recipient with the maximum timestamp, ensuring total order delivery.
- **Receiving ACKs:**
  - The `receive_ack` method simulates network delay with a 0.1-second sleep before sending an acknowledgment with the local clock as the timestamp.
- **Committing Messages:**
  - The `commit_messages` method runs as a separate thread for each recipient.
    - It continuously checks the recipient's commit queue for messages to commit.
    - When a message with a maximum timestamp is found, it updates the local clock and adds the message to the message order list, simulating

the commit operation. This ensures total order delivery.

- **Stopping Commit Threads:**
  - The `stop_commit_threads` method is used to signal the commit threads to stop processing.
    - It sets the `should_stop` flag to True, sorts the message order list to ensure messages are in total order, and then prints each message in order.
- **Printing Total Order Messages:**
  - The `print_total_order_messages` method continuously checks for messages in the message order list and prints them in total order.
- **Main Execution:**
  - In the `if _name_ == "_main_":` block, the code accepts the number of recipients and the number of total order messages to send as input.
    - It initializes the `MessageManager` and starts a thread for each recipient to handle message commitment.
    - It sends a specified number of unordered messages to recipients.
    - It sends total order messages to recipients by specifying the recipients for each message.
    - It starts a thread for printing total order messages and then signals the commit threads to stop.
    - Finally, it waits for all threads to complete their execution.

3. **Design Analysis for the above implemntation:**
   Code Design Details

   The Python code demonstrates a simplified message management system designed to ensure total order message delivery among multiple recipients using threading and message queues. This report will break down the code's design and its components:

   (a) **MessageManager Class:** The central component of the code is the `MessageManager` class, responsible for managing and ensuring total order message delivery among multiple recipients.

(b) **Initialization:**

The class is initialized with the number of recipients. It sets up various data structures:

(c) **Sending Unordered Messages:**

The `send_unordered_message` method simulates sending messages without any order guarantees. It prints the content of the message and the recipient.

(d) **Sending Total Order Messages:**

The `send_total_order_message` method sends messages that must be delivered in total order.

(e) **Broadcasting ACKs:**

The `broadcast_ack` method is responsible for ensuring acknowledgments are received from all recipients before messages are committed. It uses a dictionary to track acknowledgments received from each recipient. If acknowledgments are received from all recipients, it calculates the maximum timestamp among acknowledgments and sends the message to the commit queues of each recipient with the maximum timestamp, ensuring total order delivery.

(f) **Receiving ACKs:**

The `receive_ack` method simulates network delay with a 0.1-second sleep before sending an acknowledgment with the local clock as the timestamp.

(g) **Committing Messages:**

The `commit_messages` method runs as a separate thread for each recipient. It continuously checks the recipient's commit queue for messages to commit. When a message with a maximum timestamp is found, it updates the local clock and adds the message to the message order list, simulating the commit operation. This ensures total order delivery.

(h) **Stopping Commit Threads:**

The `stop_commit_threads` method is used to signal the commit threads to stop processing. It sets the `should_stop` flag to True, sorts the message order list to ensure messages are in total order, and then prints each message in order.

(i) **Printing Total Order Messages:**

The `print_total_order_messages` method continuously checks for messages in the message order list and prints them in total order.

(j) **Main Execution:**

In the main block, the code accepts the number of recipients and the number of total order messages to send as input. It initializes the `MessageManager` and starts a thread for each recipient to handle message commitment. It sends a specified number of unordered messages to recipients. It sends total order messages to recipients by specifying the recipients for each message. It starts a thread for printing total order messages and then signals the commit threads to stop. Finally, it waits for all threads to complete their execution.

(k) **Overall Design:**

The code demonstrates a simplified message management system for ensuring total order message delivery among multiple recipients. Threading is used to handle message delivery and acknowledgments concurrently. Total order messages are guaranteed by sending messages to commit queues with a maximum timestamp once all acknowledgments are received. The messages are committed in total order, and this order is printed after all threads finish execution.

4. **The test cases for the above code are:**

- The first test case:

  The number of recipients: 100

  The number of total order messages to send: 5

- The second test case:

  The number of recipients: 5

  The number of total order messages to send: 100

- The third test case:

  The number of recipients: 8

  The number of total order messages to send: 7