E510 - Total Order Multicast

Ahmad Faiz

1 Part 1: Pseudo-code for the Totally Ordered Broadcast

Below pseudo code in python style is an approximation to the actual C implementation covered in Part 3. The below pseudo code is for the 2 rounds TOM algorithm. Consider the following scenario:

- On server start:
  - Spawn n number of child processes
  - Each child process is a single threaded TCP socket listening on a defined port range
  - These set of socket processes can be considered as "middleware" of the system
- Client Message Issue:
  - A tcp client issues a message, e.g., "myMessage", encodes it to a specific format, and sends it to any of the middleware server
  - The message is decoded. A unique id is generated for this issued message
  - A node struct with the given message, uid, and other details is created
- Message Broadcast:
  - The issuing process increments its logical clock
  - The message is encoded with the current logical clock and the issuing process's information
  - The encoded message is broadcast to all processes in the system, including the sender
- Message Reception and Queueing:
  - Each process receives the broadcast message
  - The receiving process increments its logical clock
  - Process recieving the broadcast adjust local clock based on $max(lc_{broad}, lc_{local})$

- The message is enqueued in a local queue and the queue is sorted to maintain global order

- Acknowledgment Sending:
  - If the received message is at the head of the queue, the process sends an acknowledgment back to the original sender

- Acknowledgment Collection:
  - The original sender collects acknowledgments from all processes
  - Once all acknowledgments are received, the sender marks the message as 'ready'

- Ready Message Broadcast:
  - The sender broadcasts a 'ready' message to all processes, indicating the message is ready for delivery
  - The sender process now acquires the mutex for the shared file
  - The sender process write to the shared file and releases the mutex

- Message Delivery:
  - Upon receiving a 'ready' message, each process dequeue the process from its local queue identified by the ready message uid

This flow ensures that all processes in the distributed system deliver broadcast messages in the same total order adhering to the Totally Ordered Multicast Algorithm. Using logical clocks (event counters) to timestamp messages along with a two-round communication process (acknowledgment and 'ready' message broadcasting) are key to achieving total order and consistency across the system.

```

def total_broadcast(node, lc):
    """
    Broadcasts a message to all processes, incrementing the logical clock.
    The message is encoded with the current logical clock and the node's information.
    """
    lc += 1
    broadcast_message = encode_broadcast_buffer(node, lc)
    for i in range(NUMPROC):
        send_message(broadcast_message, BASEPORT + i)

```

```
    def handle_broadcast(client_fd, num_process, lc):
        """
        Handles a received broadcast message: increments the logical clock, enqueues the message,
        and sends an acknowledgment if the message is at the head of the queue.
        """
        lc += 1
        message, uid, sender_lc = recv_broadcast_message(client_fd)
        message_queue.put((sender_lc, uid, message))  # Enqueue with logical clock as priority
        check_and_ack_head()

    def check_and_ack_head():
        """
        Checks if the head of the queue can be acknowledged and sends an acknowledgment if so.
        """
        if not message_queue.empty():
            _, uid, _ = message_queue.queue[0]  # Peek at the head of the queue
            send_p2p_ack(uid)

    def send_p2p_ack(uid):
        """
        Sends an acknowledgment for the message with the given UID.
        """
        # Construct and send the ACK message
        # encode message with the pid of process sending ACK and uid of the actual message

    def deliver_message_if_ready(uid):
        """
        Delivers the message to the application if it's at the head of the queue and marked as 'ready'.
        """
        if not message_queue.empty() and message_queue.queue[0][1] == uid:
            _, _, message = message_queue.get()  # Remove from queue
            write_shared(message, len(message))
```

```

    def handle_ack(client_fd, lc):
        """
        Processes an acknowledgment message, potentially marking the corresponding message as ready.
        """
        uid = recv_ack(client_fd)
        # Increment ack count and check if all acks received
        mark_message_ready_if_acks_complete(uid)
            # copy buffer contents to be written to shared file
            # call broadcast_ready_message(node, lc)
            deliver_message_if_ready(uid)

    def broadcast_ready_message(node, lc):
        """
        Broadcasts a 'ready' message for a given message once all acknowledgments have been received.
        """
        lc += 1
        send_ready_to_all(node, lc)

    def handle_ready(client_fd):
        """
        Processes a 'ready' message, marking the corresponding message as ready for delivery.
        """
        uid = recv_ready(client_fd)
        deliver_message_if_ready(uid)

    def main():
        """
        Main function to simulate the flow of operations for the Totally Ordered Multicast Algorithm.
        """
        # Initialize shared file for message delivery
        init_shared_file()

        # issuing a message "myMessage" (assume from a tcp client)
        message = "myMessage"
```

```
        lc = 0  # Logical clock starts at 0
        node = create_node(message, lc)  # Create a node for the message

        # Broadcast the message to all processes
        total_broadcast(node, lc)

        # Simulate receiving the broadcast message at each process
        for i in range(NUMPROC):
            client_fd = simulate_client_connection(BASEPORT + i)
            handle_broadcast(client_fd, NUMPROC, lc)

        # Simulate each process sending an acknowledgment back to the sender
        for i in range(NUMPROC):
            client_fd = simulate_client_connection(BASEPORT + i)
            handle_ack(client_fd, lc)

        # Simulate the sender receiving all acknowledgments and broadcasting a 'ready' message
        broadcast_ready_message(node, lc)

        # Simulate each process receiving the 'ready' message and delivering the message
        for i in range(NUMPROC):
            client_fd = simulate_client_connection(BASEPORT + i)
            handle_ready(client_fd)

        # Cleanup shared file resources
        cleanup_shared_file()

    def simulate_client_connection(port):
        """
        Simulates a client connection to a process

        Parameters:
        - port: The port number of the process to connect to.

        Returns:
        A simulated client file descriptor.
```

```python
        """
        # Actual client connection logic
        return "client_fd_placeholder"

    def create_node(message, lc):
        """
        Creates a node containing the message and its logical clock timestamp.

        Parameters:
        - message: The message to be included in the node.
        - lc: The logical clock value at the time of message creation.

        Returns:
        A node object containing the message and logical clock and other details
        """
        # Placeholder for actual node creation logic
        return {"message": message, "lc": lc}

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

2 Part 2: Correctness Proof

Given: A distributed system with processes $(P_1, P_2, \ldots, P_n)$ that communicate via FIFO message channels and use logical clocks to timestamp messages. Prove that the Totally Ordered Multicast Algorithm ensures that all messages are delivered in the same total order across all processes.

Assumptions:

1. **Logical Clocks:** Each message m sent by a process P_i is timestamped with P_i's current logical clock value, $LC(m)$.

2. **FIFO Channels:** Messages sent from one process to another are received in the order they were sent.

3. **Acknowledgment Mechanism:** A message m is not delivered by any process to the application until it is acknowledged by all processes.

Claim

Assume, a contradiction case, that there exist two messages $i : M$ and $j : N$ such that process A delivers $i : M$ before $j : N$ and process B delivers $j : N$ before $i : M$, given that $LC(i : M) < LC(j : N)$

Proof by contradiction

1. **Acknowledgment Receipt:** For process B to deliver $j : N$, it must have received acknowledgments for $j : N$ from all processes, including A. This implies that A has received $j : N$.

2. **FIFO Violation:** Given $LC(i : M) < LC(j : N)$, and under the FIFO assumption, A must have sent $i : M$ before sending any acknowledgment for $j : N$, since acknowledgments are sent after receiving and processing a message.

3. **Contradiction:** The assumption leads to a contradiction with the FIFO channel assumption and the properties of logical clocks. If A sent $i : M$ before acknowledging $j : N$, then B must receive $i : M$ before or at the same time as the acknowledgment for $j : N$. This is the key to ensure that $i : M$ is processed before or at the same time as $j : N$.

4. **Conclusion:** The initial assumption leads to a contradiction; therefore, it is false. It is not possible for two processes to deliver messages in a different order under the given algorithm and FIFO ordering assumptions

Hence Proved.

3 Part 3: Implementation

This section provides key details of the implementation of the 2 round Total Ordered Multicast algorithm in C. This is a partial implementation and doesn't shows the total order maintained on all processes. This is due to bug in handling the shared file pointer when a process acquires the lock to write to this. I have not been able to resolve this successfully and hence could not implement the final testing required

However, this implementation does cover every other part correctly before this step. It sets up the middleware, implements a message passing protocol, implements queue management, and handles different message format for client to middleware and inter-process communication between different middleware for managing local Lamport clock timestamps.

Given below is a description of the codebase highlighting the flow of control execution across this system:

Initial Operations by the Main Thread (Parent Process)

1. Command-Line Arguments Processing

    - `main()` function: Begins by checking the number of command-line arguments to determine the number of subprocesses (TOM processes) to spawn. It defaults to `NUMPROC` if not specified or uses the provided argument

2. Port File Initialization
   - Opens or creates a file named `PORTSFILE` to write the listening ports of each TOM process. This file serves as a registry for clients or other processes to know which ports the TOM processes are listening on
3. Shared File Initialization
   - Calls init_shared_file () to open or create a shared file named `SHAREDFILE` in append mode. This file is used by TOM processes to write messages that have been totally ordered and are ready for delivery
4. Process Spawning
   - For each TOM process (as determined by num_process), the parent process forks a child process
   - Each child process calls sock_serve(port_num, num_process) to start a server that listens on a specific port (calculated as BASEPORT + i for the i-th process) and handles incoming connections according to the TOM algorithm
5. Synchronization and Cleanup
   - The parent process waits for all child processes to exit using waitpid()
   - After all child processes have exited, calls cleanup_shared_file () to close and clean up the shared file descriptor

Flow of Operations on recieving an ISSUE message from a client

1. **Client Issues a Message "message1"**:
   - The client sends a message to one of the TOM processes. This triggers the handle_issue() function in the server code.
2. **Initialization and Broadcast**:
   - Inside handle_issue(), the server increments its logical clock (lc) to account for the new event (the issue of a message).
   - The server reads the message from the client socket using recv_buffer().
   - The server then calls total_broadcast() to broadcast the message to all TOM processes, including itself.
3. **Broadcast Message Handling**:
   - Upon receiving the broadcast message (including receiving its own broadcast), each TOM process calls handle_broadcast().
   - The server reads the message from the client socket using recv_buffer().

- This function increments the logical clock, parses the message, and enqueues it in a local queue, sorting the queue based on its timestamp.

4. **Acknowledgment Sending**:

    - Each process checks if the newly enqueued message is at the head of its queue. If so, it sends an acknowledgment back to the original sender using send_p2p_ack().

5. **Acknowledgment Reception and Ready Marking**:

    - The original sender process collects acknowledgments for the message. Once it has received acknowledgments from all processes, it marks the message as 'ready' by calling broadcast_ready_message().
    - write_shared() acquires a mutex lock to ensure exclusive access to the shared file, writes the message buffer to the file, and then releases the lock.

6. **Ready Message Broadcasting**:

    - broadcast_ready_message() sends a 'ready' message to all TOM processes, indicating that the message has been acknowledged by all and can be delivered.

7. **Ready Message Handling**:

    - Upon receiving a 'ready' message, each process calls handle_ready() which dequeues the node by searching for it for the given UID.

Given below is a description of the different functions defined in the codebase:

- **generate_uid(unsigned int lc)**
    - **Description:** Generates a unique identifier for a message using the process ID and the logical clock value.
    - **Parameters:**
        - lc: The current value of the logical clock.
    - **Returns:** A unique identifier for the message.
- **send_buffer(int *pid_fd_ptr, char *buffer, size_t *buf_len)**
    - **Description:** Sends the content of buffer to the process identified by pid_fd_ptr until the entire buffer is sent.
    - **Parameters:**
        - pid_fd_ptr: Pointer to the file descriptor of the receiving process.
        - buffer: The message buffer to send.
        - buf_len: The length of the message buffer.

- **Returns:** 0 on success, 1 on failure.

- **recv_buffer(int *pid_fd_ptr, char *buffer, int *buf_len)**
  - **Description:** Receives data into buffer from the process identified by pid_fd_ptr until the buffer is full or the connection is closed.
  - **Parameters:**
    - pid_fd_ptr: Pointer to the file descriptor of the sending process.
    - buffer: The buffer to store received data.
    - buf_len: The length of the buffer.
  - **Returns:** 0 on success, 1 on failure.

- **check_and_ack_head()**
  - **Description:** Checks if the message at the head of the local queue can be acknowledged and sends an acknowledgment if conditions are met.
  - **Parameters:** None.
  - **Returns:** None.

- **send_p2p_ack(int sender_port, int base_pid, unsigned int uid)**
  - **Description:** Sends an acknowledgment message to the original sender of a message.
  - **Parameters:**
    - sender_port: The port number of the original sender.
    - base_pid: The process ID of the original sender.
    - uid: The unique identifier of the message being acknowledged.
  - **Returns:** None.

- **write_shared(const char* buffer, size_t buf_len)**
  - **Description:** Writes the content of buffer to a shared file, ensuring mutual exclusion via a mutex lock.
  - **Parameters:**
    - buffer: The message buffer to write to the shared file.
    - buf_len: The length of the message buffer.
  - **Returns:** None.

- **handle_ack(int * client_fd_ptr , unsigned int *lc)**
  - **Description:** Handles the receipt of an acknowledgment message, increments the acknowledgment count for the corresponding message, and checks if the message is ready to be marked as 'ready'.
  - **Parameters:**

- client_fd_ptr : Pointer to the file descriptor

* **handle_ready(int * client_fd_ptr)**
  - **Description:** Handles the receipt of a 'ready' message, indicating that a message has been acknowledged by all processes and can be delivered. Delivers the message to the local application or process.
  - **Parameters:**
    - client_fd_ptr : Pointer to the file descriptor of the process from which the 'ready' message was received.
  - **Returns:** None.
* **encode_broadcast_buffer(node_t *node, char *buffer, unsigned int *lc)**
  - **Description:** Encodes a message and its metadata (e.g., sender ID, timestamp) into a buffer for broadcasting, ensuring proper ordering based on the logical clock value.
  - **Parameters:**
    - node: The node containing the message to be broadcasted.
    - buffer: The buffer to encode the message into.
    - lc: Pointer to the logical clock value.
  - **Returns:** None.
* **total_broadcast(node_t *node, unsigned int *lc)**
  - **Description:** Initiates the broadcasting of a message to all processes, including itself, by encoding the message, sending it to all other processes, and updating the logical clock value.
  - **Parameters:**
    - node: The node containing the message to be broadcasted.
    - lc: Pointer to the logical clock value.
  - **Returns:** 0 on success, 1 on failure.
* **broadcast_ready_message(node_t *node, unsigned int *lc)**
  - **Description:** Once all acknowledgments for a message have been received, broadcasts a 'ready' message to all processes, indicating that the message can be delivered.
  - **Parameters:**
    - node: The node containing the message that is ready to be delivered.
    - lc: Pointer to the logical clock value.
  - **Returns:** None.
* **handle_issue(int * client_fd_ptr , int num_process, unsigned int *lc, int base_por**

· **Description:** Handles the issuance of a new message from a client. Prepares the message for broadcasting by generating a unique identifier, encoding it, and initiating the broadcast process.

· **Parameters:**

- client_fd_ptr : Pointer to the file descriptor of the client issuing the message.
- num_process: The total number of processes in the system.
- lc: Pointer to the logical clock value.
- base_port: The base port number for process communication.

· **Returns:** None.

3.1 Running the binaries

Listing 1: Compiling and executing binaries

```
clang server.c queue.c client_handle.c -o server
./server

clang client_app.c -o client

./client samplemessage
```

3.2 Screenshots

The entire codebase can be found at this Github repo:
https://github.iu.edu/afaiz/distsys$_{s}p24/tree/main/TOB$

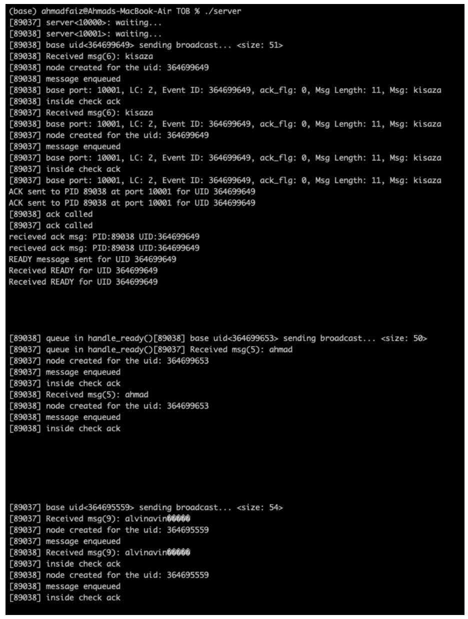

Figure 1: Server Process

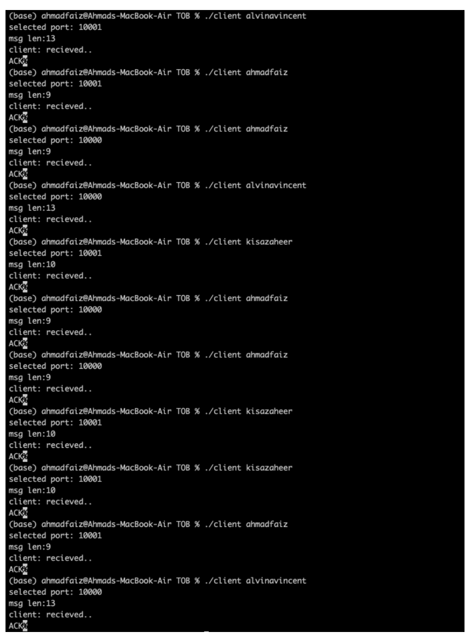

Figure 2: Client Process

Figure 3: Shared File