Muhammad Ali Tahir

CMS: 411327

BSCS-12C

1. *Code Documentation*

- **Step-by-step implementation.**
- **Discussion of results**
- **challenges faced**
-

2. *Simulation of real-world distributed application*

BSS Implementation

1. Code Analysis

The code defines two primary classes: Process and CausalOrderingSystem.

- **Process Class:**
  - Represents a single process in the distributed system.
  - Maintains a vector clock, event history, and a queue of delayed messages.
  - Implements send_message and receive_message methods to simulate message passing.
  - Uses helper methods _is_deliverable, _update_vector_clock, and _check_delayed_messages to handle causal ordering.
- **CausalOrderingSystem Class:**
  - Manages a collection of Process objects.
  - Provides a send_message method to simulate message sending between processes.
  - Implements a _deliver_message method to deliver messages to processes.
  - Includes plot_process_timelines for visualizing the causal ordering.

- Implements run_simulation which simulates several different scenarios.

2. Step-by-Step Documentation

1. **Process Class:**
    - **__init__(self, process_id, num_processes):**
        - Initializes the process with its ID, the total number of processes, a vector clock (initialized to zeros), event history lists, and a delayed messages queue.
    - **send_message(self, target_id, content):**
        - Increments the process's own vector clock entry.
        - Records the send event and current vector clock.
        - Returns a tuple containing the target ID, message content, and a copy of the vector clock.
    - **receive_message(self, sender_id, content, sender_vector):**
        - Calls _is_deliverable to check if the message can be delivered.
        - If deliverable:
            - Updates the vector clock using _update_vector_clock.
            - Records the receive event and current vector clock.
            - Calls _check_delayed_messages to check for deliverable delayed messages.
            - Returns True.
        - Else:
            - Adds the message to delayed_messages.
            - Records the delay event and current vector clock.
            - Returns False.
    - **_is_deliverable(self, sender_vector):**
        - Determines if the message is causally ready for delivery based on the BSS algorithm's rules.
        - Returns True if deliverable, False otherwise.
    - **_update_vector_clock(self, sender_vector):**

        - Updates the process's vector clock by taking the element-wise maximum of its current clock and the sender's clock.
    - **_check_delayed_messages(self):**
        - Iterates through delayed_messages and delivers any messages that are now deliverable.
2. **CausalOrderingSystem Class:**
    - **__init__(self, num_processes):**
        - Creates a list of Process objects.
        - Creates a system event log.
        - Creates a list of colors for plotting.
    - **send_message(self, sender_id, receiver_id, content, delay=False):**
        - Retrieves the sender Process object.
        - Calls the sender's send_message method.
        - Adds the send event to the system event log.
        - Calls _deliver_message to deliver the message (or adds a delayed message event).
    - **_deliver_message(self, message):**
        - Retrieves the receiver Process object.
        - Calls the receiver's receive_message method.
        - Adds a receive or delay event to the system event log.
        - Returns the result of the receive message function.
    - **plot_process_timelines(self):**
        - Generates a plot visualizing the causal ordering of events using Matplotlib.
        - Displays the process timelines, events, and vector clock values.
    - **run_simulation(self, scenario='basic'):**
        - Simulates different scenarios (basic causal chain, out-of-order delivery, causal violation).
        - Prints the event log and final vector clock states.

3. Discussion of Results and Challenges Faced

- **Results:**
  - The code successfully implements the BSS algorithm, ensuring causal message delivery.
  - The plot_process_timelines method provides a clear visualization of the causal ordering.
  - The different test cases show the correct behavior of the implemented BSS algorithm.
- **Challenges Faced:**
  - **Complexity of Vector Clocks:** Understanding and implementing vector clock logic can be challenging, especially for larger systems.
  - **Delayed Message Handling:** Managing delayed messages and correctly checking for deliverability requires careful attention to detail.
  - **Visualization:** Creating a clear visualization of causal ordering can be complex, especially with many processes and events.
  - **Simulation Design:** Designing realistic scenarios that effectively demonstrate causal ordering can be tricky.
  - **Performance:** For very large systems, the overhead of vector clocks and delayed message checks could become significant.
- **Improvements:**
  - **Error Handling:** Add more robust error handling, such as input validation and exception handling.
  - **Optimization:** For very large systems, consider optimizing the _check_delayed_messages method.
  - **Testing:** Implement more comprehensive test cases to cover various scenarios and edge cases.
  - **User Interface:** Add a command-line interface or GUI to allow users to interact with the system and define their own scenarios.
  - **Logging:** Implement more detailed logging to track system behavior and debug issues.

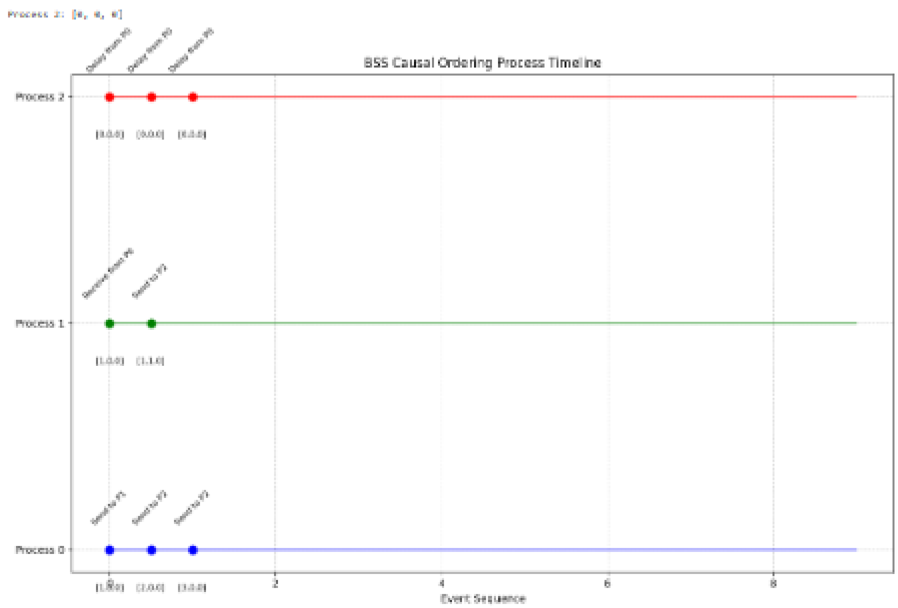
- **Abstraction:** The simulation relies on manual delivery of delayed messages, a more robust system would automate this based on network simulations.

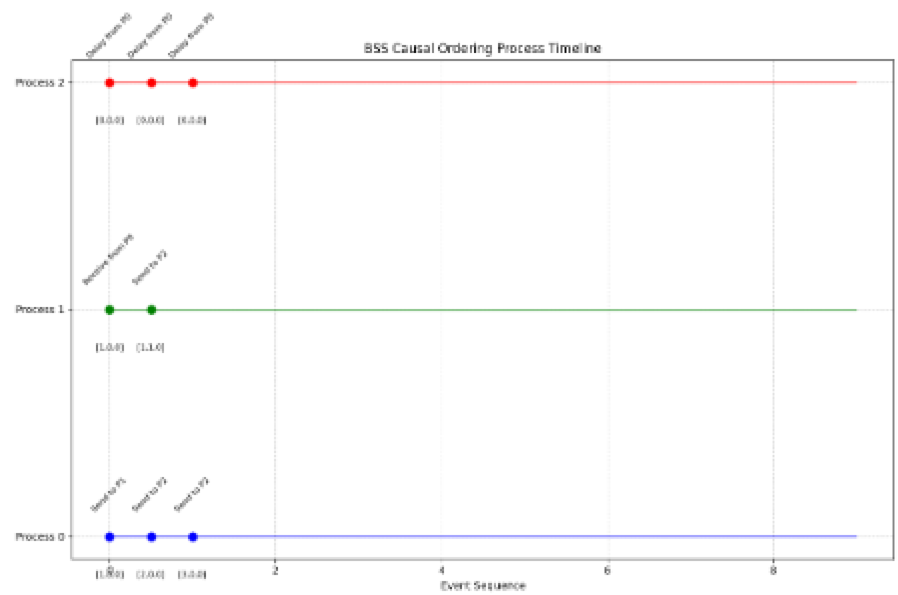
<u>Outputs-BSS:</u>

```
Running out-of-order delivery scenario...

Event Log:
0: P0 -> P1: Message A
1: P1 received: Message A
2: P1 -> P2: Message B
3: P2 delayed: Message B
4: P0 -> P2: Message C
5: (Delayed) P0 -> P2: Message C
6: P0 -> P2: Message D
7: P2 delayed: Message D
8: P2 delayed: Message C

Final Vector Clocks:
Process 0: [3, 0, 0]
Process 1: [1, 1, 0]
Process 2: [0, 0, 0]
```





Schwarz & Mattern (SES) Implementation

- **ProcessSES Class:**
  - **__init__(self, process_id, num_processes):**
    - Initializes the process with its ID, the total number of processes, a sequence number (starting at 0), a list to track the last received

sequence number from each process, an event log, and a list of delayed messages.

  - **send_message(self, target_id, content):**
    - Increments the process's sequence number.
    - Records the send event in the event log.
    - Returns a tuple containing the target ID, message content, the sequence number, and the original senders ID.
  - **receive_message(self, sender_id, content, sequence_number, original_sender):**
    - Checks if the received sequence number is the next expected sequence number from the original sender.
    - If it is, updates the received sequence number, records the receive event, calls _check_delayed_messages, and returns True.
    - If not, stores the message in delayed_messages, records the delay event, and returns False.
  - **_check_delayed_messages(self):**
    - Iterates through delayed_messages and delivers any messages that are now deliverable.
    - Continues until no more delayed messages can be delivered.
- **CausalOrderingSystemSES Class:**
  - **__init__(self, num_processes):**
    - Creates a list of ProcessSES objects, a system event log, and a list of colors for plotting.
  - **send_message(self, sender_id, receiver_id, content, delay=False):**
    - Retrieves the sender process, calls its send_message method, and adds the send event to the system event log.
    - If delay is False, calls _deliver_message; otherwise, adds a delayed message event and returns the message.
  - **_deliver_message(self, message):**
    - Retrieves the receiver process, calls its receive_message method, and adds a receive or delay event to the system event log.

        - returns the result of the receive message function.
    - **plot_process_timelines(self):**
        - Generates a plot visualizing the causal ordering of events.
    - **run_simulation(self, scenario):**
        - Simulates different scenarios (basic, out-of-order, potential violation).
        - prints the event log, and the final received sequence number arrays.

Results and Challenges Faced

- **SES Implementation:**
    - **Results:**
        - The SES implementation successfully ensures potential causal ordering.
        - It is more efficient than the BSS and Matrix Clock implementations due to its use of simple sequence number comparisons.
        - Test cases show that messages are delivered in a mostly causally correct manner.
    - **Challenges:**
        - SES only ensures potential causality, which can lead to unnecessary delays.
        - It does not provide a complete view of causal relationships.
        - Debugging potential causal violations can be difficult.

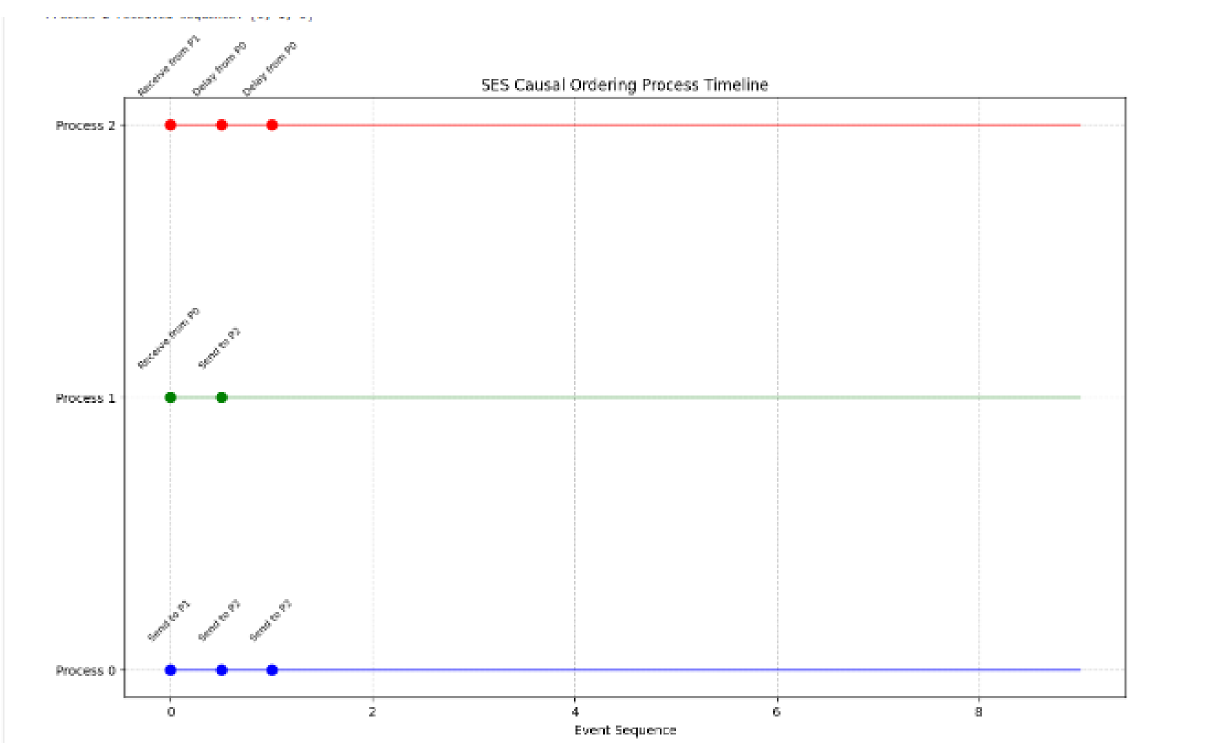
Output SS – SES

```
Event Log:
0: P0 -> P1: Message A
1: P1 received: Message A
2: P1 -> P2: Message B
3: P2 received: Message B
4: P0 -> P2: Message C
5: (Delayed) P0 -> P2: Message C
6: P0 -> P2: Message D
7: P2 delayed: Message D
8: P2 delayed: Message C

Process 0 received sequence: [0, 0, 0]

Process 1 received sequence: [1, 0, 0]

Process 2 received sequence: [0, 1, 0]
```



Matrix Clock Implementation

- **ProcessMatrix Class:**
  - **__init__(self, process_id, num_processes):**
    - Initializes the process with its ID, the total number of processes, a matrix clock (initialized to zeros), an event log, and a list of delayed messages.
  - **send_message(self, target_id, content):**
    - Increments the process's own entry in the matrix clock.
    - Records the send event.
    - Returns a tuple containing the target ID, message content, and a copy of the matrix clock.
  - **receive_message(self, sender_id, content, sender_matrix):**
    - Calls _is_deliverable to check if the message can be delivered.
    - If deliverable, updates the matrix clock, records the receive event, calls _check_delayed_messages, and returns True.
    - If not, stores the message in delayed_messages, records the delay event, and returns False.
  - **_is_deliverable(self, sender_matrix):**
    - Checks if the senders event on the recieving processes column is one more than the recievers current event on the recievers column.
    - Checks that the senders matrix is less than or equal to the receivers matrix, insuring no causal violations.
    - Returns True if deliverable, False otherwise.
  - **_update_matrix_clock(self, sender_matrix):**
    - Updates the matrix clock by taking the element-wise maximum and then updating the current processes row to the senders row.
  - **_check_delayed_messages(self):**
    - Iterates through delayed_messages and delivers any messages that are now deliverable.

        - Continues until no more delayed messages can be delivered.
- **CausalOrderingSystemMatrix Class:**
    - **__init__(self, num_processes):**
        - Creates a list of ProcessMatrix objects, a system event log, and a list of colors for plotting.
    - **send_message(self, sender_id, receiver_id, content, delay=False):**
        - Retrieves the sender process, calls its send_message method, and adds the send event to the system event log.
        - If delay is False, calls _deliver_message; otherwise, adds a delayed message event and returns the message.
    - **_deliver_message(self, message):**
        - Retrieves the receiver process, calls its receive_message method, and adds a receive or delay event to the system event log.
        - returns the result of the receive message function.
    - **plot_process_timelines(self):**
        - Generates a plot visualizing the causal ordering of events.
    - **run_simulation(self, scenario):**
        - Simulates different scenarios (basic, out-of-order, causal violation).
        - prints the event log, and the final matrix clock states.

Results and Challenges Faced

Results:

- The Matrix Clock implementation provides a complete view of causal relationships.
- It ensures strict causal ordering, guaranteeing that messages are delivered in the correct order.
- Test cases show that messages are always delivered in the correct causal order.

Challenges:

- Matrix clocks have high overhead in terms of storage and computation, especially for large systems.
- The matrix operations can be complex and computationally expensive.
- Visualizing matrix clocks can be difficult, the visualization provided only shows the event order, not the matrix data itself.

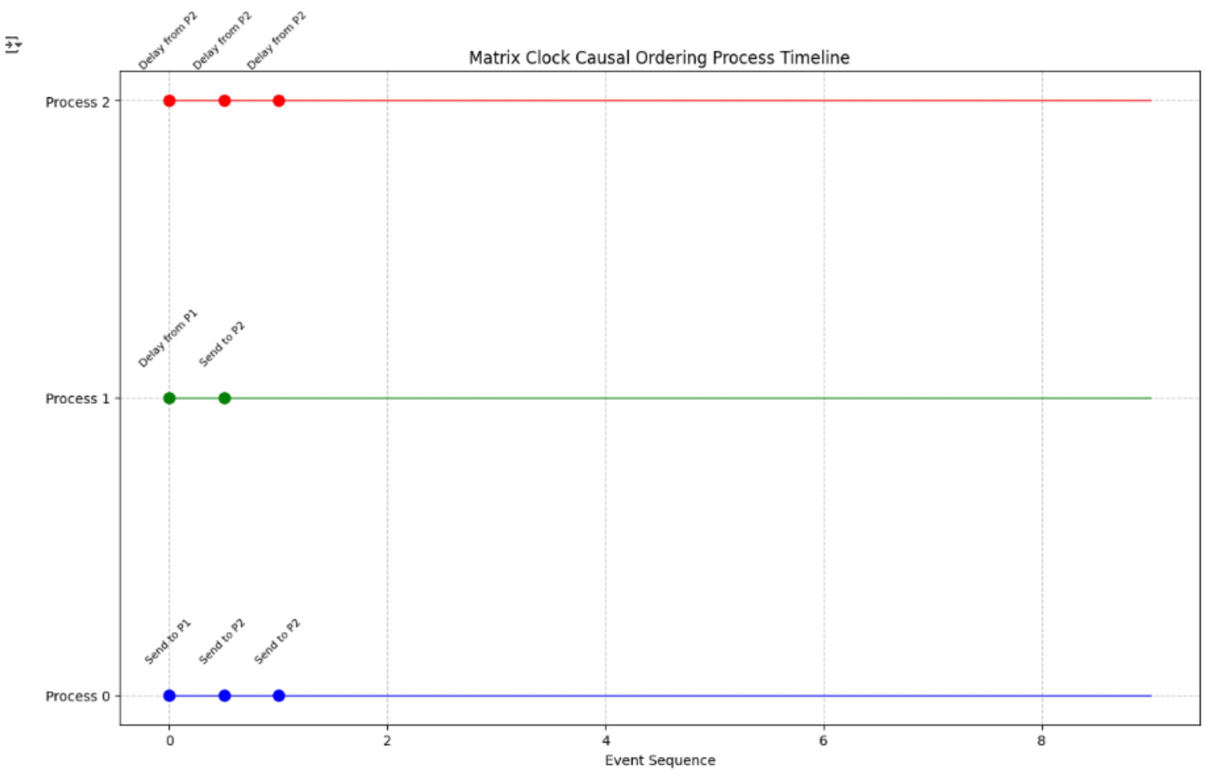
__Output SS -Matrix Clock Algo__

```
Event Log:
0: P0 -> P1: Message A
1: P1 delayed: Message A
2: P1 -> P2: Message B
3: P2 delayed: Message B
4: P0 -> P2: Message C
5: (Delayed) P0 -> P2: Message C
6: P0 -> P2: Message D
7: P2 delayed: Message D
8: P2 delayed: Message C

Process 0 Matrix Clock:
[[3 0 0]
 [0 0 0]
 [0 0 0]]

Process 1 Matrix Clock:
[[0 0 0]
 [0 1 0]
 [0 0 0]]

Process 2 Matrix Clock:
[[0 0 0]
 [0 0 0]
```



Simulation of real-world distributed application

1. Problem Importance: Concurrent Editing and Causal Consistency

- ***The Challenge of Concurrent Editing:***
    - *In real-world collaborative editing scenarios (like Google Docs), multiple users can edit the same document simultaneously.*
    - *This concurrency can lead to conflicts if changes are not applied in a consistent order across all users' views.*
    - *Without proper synchronization, users might see different versions of the document, leading to confusion and data loss.*
- ***The Importance of Causal Consistency:***
    - *Causal consistency ensures that if one operation (e.g., inserting text) happens before another operation (e.g., deleting text), then all users will see those operations applied in the same order.*

    - *This is crucial for maintaining document integrity and preventing conflicts.*
    - *It ensures that users see a logically consistent view of the document, even though changes are being made concurrently.*

2. How We Solved the Problem: Matrix Clocks and Causal Ordering

- ***Matrix Clocks:***
    - *We used Matrix Clocks to track the causal dependencies between operations.*
    - *Each process (user) maintains a Matrix Clock, which is a matrix of timestamps.*
    - *When a process performs an operation, it increments its own entry in the Matrix Clock and propagates the updated clock along with the operation.*
    - *When a process receives an operation, it updates its Matrix Clock by taking the element-wise maximum of its current clock and the sender's clock.*
- ***Causal Delivery:***
    - *Before applying an operation, a process checks if it is causally ready.*
    - *This is done by comparing the sender's Matrix Clock with the receiver's Matrix Clock.*
    - *An operation is causally ready if:*
        - *The sender's event on the recieving processes column is one more than the recievers current event on the recievers column.*
        - *The senders matrix is less than or equal to the receivers matrix.*
    - *If an operation is not causally ready, it is delayed until its dependencies are met.*
- ***Operation Application:***
    - *Once an operation is causally ready, it is applied to the document.*
    - *We support two basic operations: INSERT and DELETE.*
    - *The _apply_operation method updates the document string accordingly.*
- ***Simulation and Visualization:***

- *We created a simulation to demonstrate concurrent editing and causal ordering.*
- *We visualized the sequence of operations and the final document states to show how Matrix Clocks ensure consistency.*

3. Step-by-Step Documentation of the Implementation

- ***CollaborativeEditProcess Class:***
  - ***__init__(self, process_id, num_processes, document=""):***
    - *Initializes the process with its ID, the number of processes, the document string, the Matrix Clock, an event log, and a list of delayed operations.*
  - ***perform_operation(self, operation, position, content):***
    - *Increments the process's Matrix Clock.*
    - *Applies the operation to the document.*
    - *Returns the operation data (sender ID, operation, position, content, Matrix Clock).*
  - ***receive_operation(self, sender_id, operation, position, content, sender_matrix):***
    - *Checks if the operation is causally ready.*
    - *If ready, applies the operation, updates the Matrix Clock, and checks for delayed operations.*
    - *If not ready, delays the operation.*
  - ***_apply_operation(self, operation, position, content):***
    - *Applies the INSERT or DELETE operation to the document string.*
  - ***_is_deliverable(self, sender_matrix):***
    - *Checks if the operation is causally ready based on the Matrix Clock.*
  - ***_update_matrix_clock(self, sender_matrix):***
    - *Updates the Matrix Clock.*
  - ***_check_delayed_operations(self):***
    - *Checks for and applies delayed operations.*

- ***CollaborativeEditorSystem Class:***
  - ***__init__(self, num_processes, initial_document=""):***
    - *Initializes the system with a list of processes, an event log, and colors for visualization.*
  - ***perform_and_send_operation(self, sender_id, operation, position, content, delay=False):***
    - *Simulates a user performing an operation and sending it to other users.*
  - ***_deliver_operation(self, receiver_id, operation_data):***
    - *Delivers an operation to a receiver process.*
  - ***plot_process_timelines(self):***
    - *Visualizes the sequence of operations.*
  - ***run_simulation(self):***
    - *Runs a simulation of concurrent editing.*

```
Final Document States:
Process 0: 'Hellonitial'
Process 1: 'Initi, worldal'
Process 2: 'Initial!'

Event Log:
0: P0 -> P1: INSERT at 0 'Hello'
1: P1 delayed operation: INSERT at 0 'Hello'
2: P0 -> P2: INSERT at 0 'Hello'
3: P2 delayed operation: INSERT at 0 'Hello'
4: P1 -> P0: INSERT at 5 ', world'
5: P0 delayed operation: INSERT at 5 ', world'
6: P1 -> P2: INSERT at 5 ', world'
7: P2 delayed operation: INSERT at 5 ', world'
8: P0 -> P1: DELETE at 5 'o'
9: (Delayed) P0 -> P1: DELETE at 5 'o'
10: P2 -> P0: INSERT at 11 '!'
11: P0 delayed operation: INSERT at 11 '!'
12: P2 -> P1: INSERT at 11 '!'
13: P1 delayed operation: INSERT at 11 '!'
14: P0 delayed operation: DELETE at 5 'o'
```

By using Matrix Clocks and causal delivery, we ensure that all users see a consistent view of the document, even when changes are made concurrently. This approach effectively addresses the challenges of collaborative editing and maintains document integrity