I have used a processing ring (token passing) for implementation.

A token is being circulated in a specific circular order of nodes. The token is only available with one node (by the broadcast server of this node) at a time. After using the token, the server forwards this token to the next node's broadcast server. (I am using the terms server and broadcast servers interchangeably in this document, mentioning this so that server is not confused with a centralized server. I have not used a centralized server for any purpose)

The implementation guarantees the broadcast of all the messages delivered from any application process to its broadcast server. (Assuming none of the nodes fail, and the token is not lost.)

Generating unique sequence number:

At any time, the node that has the token can assign sequence numbers to its messages using the token. Specifically, the token contains a variable denoting the current assignable sequence number. Thus, any node with the token can assign this number to one of its messages and increment this variable. The node's broadcast server will assign sequence numbers using this technique for all messages in its queue. The broadcast server will assign the sequence number using toking for each message in its queue and append the messages to the token. Finally, after appending all messages and reading all available messages from the token, the broadcast server will forward this token to the next node's broadcast server.

Stability of messages:

For each message, the token also stores the information of which node has read that message from the token. i.e., let m be the message, then say m_k is true if the k^{th} node has read this message successfully from the token (i.e., the broadcast server of k^{th} node has successfully stored the message m with it), otherwise it is false. The token keeps this information for all values of k.

When a node has a token, it will copy the messages from the token and set this value as True for each message. Now, when another node has a token, it will check if m_k is true for all k. If it is, then m is stable.

Note: These above notations are just for the sake of explanation and simplicity. I have used longer variable names to denote this information.

Stability time of message:

Each server monitors how many times that token has come to that server after appending **its own** message to the token. 0^{th} encounter when the server appends the message to token, 1^{st} encounter when the token comes back to this server, and the 2^{nd} time when the token is again back at this server. The server saves the timestamping at the 0^{th} encounter, and in the 2^{nd} encounter, it calculates {the current time} – {saved timed stamp}, which is precisely the time taken for two complete circulations.

Each server is responsible for calculating the stability time of **its own** messages only. This stability time is also saved in the token for each message. When this information is available for all messages, one of the nodes will print this information and the average stability time. (Sum of all stability times, divided by the total number of broadcasted messages).

Running the code:

1. **config file**: This file contains the configuration of the processor ring in the following format
    - The first line should contain the total number of nodes, say n
    - The details of n lines should follow in the following format (one after the another):
        - First line of node will contain 4 details separated by space as follows:
          node_id app_count node_ip_address broadcast_server_port

          Here app_count is the number of application processes on the node.
          Example: 1 3 localhost 49200
          In this example, 1 is the node id, 3 denotes that there are 3 application processes in this node, localhost is the IP address of this node and 49200 is the port address of broadcast server at this node.

        - If app_count = c, the next c lines should describe the details of each application process as follows
          application_process_number application_port message_file

          Example: 1 49201 msg1
          In this example, 1 is the application process number, 49201 is the port address of application process 1, and msg1 is the file which contains messages to be broadcasted by application process 1.

      The above details should be given for all the n nodes one after the other, without any empty line in middle.
    - The next line should contain n integers, describing the order in which the token will be circulated. These integers should be the node_id's
    - The next line will contain a single integer denoting the node_id of the node which has the token initially.
    - No empty lines are allowed in between
    - The contents of this file should be the same for all nodes.
    - Ensure that any other process uses none of those mentioned above sockets.

2. **Running a node:** Ensure that header.py, broadcast_server.py, application_process.py, and node.py are in the same folder. This folder should also contain the config file and the message_file with the same as mentioned in the config file (i.e., if node if for this node is 1 and the message_file mentioned for node_id 1 is msg1 in the config file, then msg1 should be there in this folder. Enter all the messages for this node in the message_file, and each message should be present on a different line.
   To start a node with the desired node_id, use the following command:
   For Windows*: py node.py node_id*
   For Linux: *python3 node.py node_id*
   Where node_id is an integer.

   Run as many nodes as mentioned in the config file. To avoid problems, start all nodes except the one with the token initially. The node having the token should be started at last (otherwise, if the next node is not started, then this node will raise an error while forwarding the token)

3. Alternatively, the nodes can also be started using automated scripts instantaneously.
   i) On Windows, you can use the runner.bat script.
      For each node_id, write the following line in the runner.bat file:
      start cmd.exe /c py node.py node_id

      To run this batch file, double click the batch file or run the following command:
      runner.bat

   ii) On Linux, you can use the runner.sh script.
      For each node_id, write the following line in the runner.sh file:
      gnome-terminal -- bash -c "python3 node.py node_id; exec bash"

      To run this batch file, double click the batch file or run the following command:
      ./runner.sh
4. **Closing the nodes**: Close all the terminal windows after execution. (Just Ctrl + C won't work because the broadcast_server and application_process processes will keep on running, hence close the terminal, which will, in turn, kill these processes)

Output and Performance:

1. **Output Details:** The output of i-th application is redirected to "./output/app_i" file. The messages received by application processes will be visible in its corresponding file. The node that initially had the token will print the details of broadcasted messages, then the details of stability time and the average stability time.

2. **Average messages broadcasted:** On average, ***100% of the messages received by a broadcast server from its application process*** are broadcasted. Since the token keeps on circulating infinitely, thus, all the messages in the queue of the broadcast server will be appended to the token.

3. **Average stability time:** I ran the implementation for 2 to 6 nodes (4 to 10 application processes) and 20 to 500 messages. I observed that the average stability time increases with an increase in the total number of messages or the total number of nodes.
   In case of an increase in the number of messages, the token holding time increases since each message is read from the node before forwarding, thus increasing the time required for 2 complete rotations (per new message).
   In case of an increase in the number of nodes, the ring circulation time increases, thus increasing the time for 2 complete rotations (per new message).
   Hence, both of these lead to increasing in average stability time. Therefore, the average stability time is directly proportional to the total number of messages and the total number of nodes.