Process Communication:

Each process in the system communicates with its neighboring processes over a network using sockets. A single process has incoming and outgoing connections, where the "successor" is the next process to which it forwards the token, and the "predecessor" is the process from which it receives the token. The algorithm ensures that every process listens for incoming messages and periodically passes the token to the successor.

The communication between processes is handled using two main types of messages:

- **Token Message**: Represents a pass of control or state information.
- **Marker Message**: Used during a snapshot to record the state of the system.

Token Passing Logic:

The token-passing mechanism allows processes to coordinate activities in a distributed environment. If a process holds the token and has established outgoing connections, it sends the token to its successor after a specified delay (`tokenDelay`). If the outgoing connection to the successor is not yet established, the process will wait and retry.

The process ensures no busy-waiting by using a small delay between checks, thus conserving system resources. The token-passing task continues as long as the process is running.

Snapshot Mechanism:

The algorithm implements a distributed snapshot using a version of the Chandy-Lamport algorithm. A snapshot is initiated when a process reaches a specific state, called the `snapshotTriggerState`. When triggered, the snapshot mechanism does the following:

1. Records the local state of the process, including its current state and whether it has the token.
2. Sends a "marker" message to all outgoing connections to record the state of the channels.
3. Waits for all incoming marker messages from neighboring processes. This indicates that all channels' states have been recorded.
4. Once all markers have been received, the snapshot is complete, and the state is finalized.

Concurrency and Synchronization:

To handle concurrency between multiple threads, the code uses mutexes (`std::mutex`) to prevent race conditions when processes modify shared data structures, such as connection maps and snapshot queues. Each process has three threads:

1. **Listening Thread**: Handles incoming messages from other processes and handles message-specific logic, such as responding to "token" or "marker" messages.
2. **Token Thread**: Manages the token-passing task, ensuring that the token is passed from one process to the next.
3. **Snapshot Thread**: Responsible for initiating the snapshot and managing its progress.

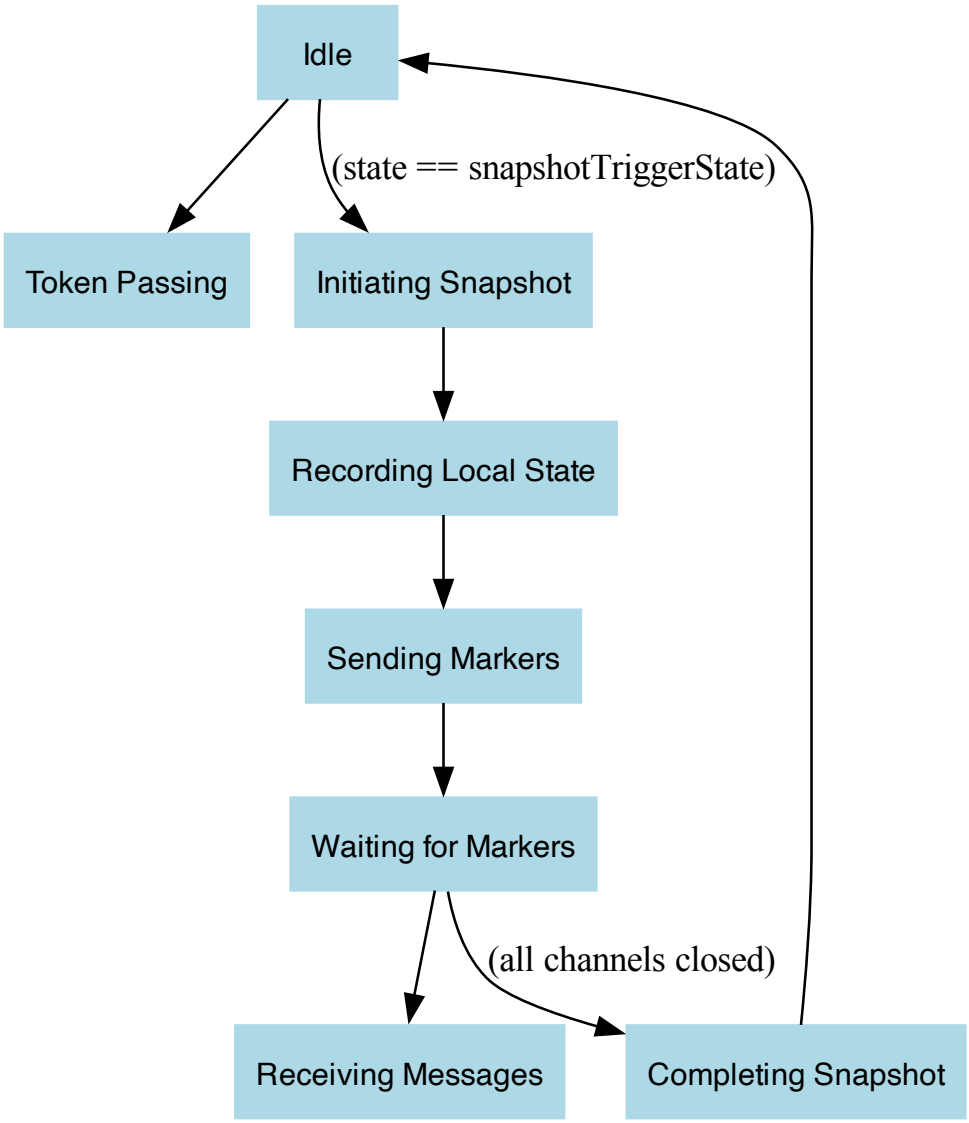
Message Handling:

When a message is received, the process identifies the message type (either "token" or "marker") and updates its state accordingly. For token messages, the process increments its internal state and prepares to pass the token. For marker messages, the process records the snapshot and waits for further markers to ensure consistency across all channels.

The snapshot's state is stored in a `curRecord` array, which holds the process ID, current state, snapshot ID, and whether the process currently holds the token. The snapshot's progress and state are logged, providing clear visibility into the algorithm's operation.

Error Handling:

To ensure robust connections between processes, the algorithm includes error-checking mechanisms. It handles scenarios such as connection failures and broken sockets. If a connection is broken, the process attempts to re-establish the connection.



State diagram