# Simplified Pub/Sub IPC Implementation

Implementing a basic publish-subscribe inter-process communication (IPC) mechanism within mCertikOS, allowing processes to publish messages to a pre-defined topic ("goodbye\_topic") and have subscriber processes receive and display these messages via a callback function.

## Phase 1: Foundation - Core Data Structures & Memory Management

Define core data structures for topic, subscriber, and messages, and establish memory management using mCertikOS containers.

* **Define Data Structures (pubsub.h):**
  + message\_t: Struct containing char data[MESSAGE\_SIZE].
  + subscriber\_t: Struct containing:
    - int pid: Process ID of the subscriber.
    - message\_callback\_t callback: Function pointer to the callback.
    - queue\_t\* message\_queue: Message queue for the subscriber.
    - spinlock\_t lock: Lock for queue access.
  + topic\_t: Struct containing:
    - char name: Topic name ("goodbye\_topic").
    - subscriber\_t \*subscriber: Pointer to a single subscriber.
    - bool subscriber\_present: Flag if the subscriber is present.
    - spinlock\_t lock: Lock for subscriber management.
  + queue\_t: Define queue\_t structure (consider a circular buffer).
  + Define constants: TOPIC\_NAME, MESSAGE\_SIZE, QUEUE\_SIZE.
* **Implement Message Management (message.c):**
  + message\_create(const char \*message\_data): Allocates memory for a message\_t using container\_alloc and copies message\_data. Returns message\_t\*.
  + message\_enqueue(subscriber\_t \*subscriber, message\_t \*message): Enqueues a message to the subscriber's queue, manages queue overflow (dequeue oldest message).
  + message\_dequeue(subscriber\_t \*subscriber): Dequeues a message from the subscriber's queue.
  + queue\_create(size\_t queue\_size): Creates a message queue of size queue\_size, using container\_alloc for internal storage.
  + queue\_destroy(queue\_t\* queue): Destroys a message queue, freeing allocated memory with container\_free.
  + queue\_enqueue(queue\_t\* queue, message\_t\* message): Enqueues a message to the queue.
  + queue\_dequeue(queue\_t\* queue): Dequeues a message from the queue.
  + queue\_is\_full(queue\_t\* queue): Checks if the queue is full.
* **Implement Topic Initialization (topic.c):**
  + topic\_init(): Initializes the global goodbye\_topic structure with the topic name, setting subscriber to NULL and subscriber\_present to false, initializing spinlock.

## Phase 2: Topic and Subscriber Management

Implement functions for subscribing and unsubscribing to the pre-defined topic.

* **Implement topic\_subscribe (topic.c):**
  + Checks if topic\_name is equal to TOPIC\_NAME.
  + Acquire goodbye\_topic.lock
  + Checks if a subscriber already exists (subscriber\_present). If so, return an error.
  + Allocates memory for a subscriber\_t using container\_alloc.
  + Allocates memory for message\_queue using queue\_create.
  + Sets subscriber->pid to the current process ID (current\_process\_id()).
  + Sets subscriber->callback to the provided callback function.
  + Initializes subscriber->lock.
  + Set subscriber\_present to true.
  + Releases goodbye\_topic.lock.
  + Returns success or failure code.
* **Implement topic\_unsubscribe (topic.c):**
  + Checks if topic\_name is equal to TOPIC\_NAME.
  + Acquire goodbye\_topic.lock
  + Checks if a subscriber exists (subscriber\_present). If not, return an error.
  + Frees the message\_queue using queue\_destroy.
  + Free the subscriber\_t structure using container\_free.
  + Sets subscriber\_present to false.
  + Releases goodbye\_topic.lock.
  + Returns success or failure code.

## Phase 3: System Call Integration

Expose the Pub/Sub functionality through system calls.

* **Add Syscall Definitions (syscall.h):**
  + Define system call numbers for SYS\_PUB, SYS\_SUB, and SYS\_UNSUB.
* **Implement Syscall Handlers (syscall.c):**
  + sys\_sub(const char \*topic\_name, message\_callback\_t callback, size\_t queue\_size): Calls topic\_subscribe with the provided arguments.
  + sys\_unsub(const char \*topic\_name): Calls topic\_unsubscribe with the provided topic\_name.
  + sys\_pub(const char \*topic\_name, char \*message\_data):
    - Checks if topic\_name is equal to TOPIC\_NAME.
    - Acquires goodbye\_topic.lock.
    - Checks if a subscriber exists. If not, return an error.
    - Creates a message\_t using message\_create.
    - Enqueues the message using message\_enqueue.
    - Releases goodbye\_topic.lock.
    - Calls trap\_send(subscriber->pid) to trigger the callback execution in the subscriber process.
    - Returns success or failure code.
* **Integrate with mCertikOS Syscall Mechanism:**
  + Modify the mCertikOS kernel to handle the new system calls (add entries in the syscall table, etc.).

## Phase 4: Message Delivery and Callback Execution

Implement the trap-based mechanism for delivering messages and executing callbacks in the subscriber process.

* **Implement trap\_send(int pid) (trap.c):**
  + This function needs to trigger a trap to the process with an ID pid. This will likely involve using mCertikOS-specific functions to send a signal or interrupt to the target process.
* **Implement trap\_handler(int trap\_number) (trap.c):**
  + This function is the trap handler that will be executed in the *subscriber's* context.
  + Verify that the trap is for the correct reason/number.
  + Acquire goodbye\_topic.lock.
  + Check if a subscriber exists and that the PID matches the current process.
  + Dequeues the message using message\_dequeue.
  + Releases goodbye\_topic.lock.
  + If a message was dequeued:
    - Calls the subscriber's callback function (subscriber->callback(message->data)).
    - Free the message using container\_free.
* **Implement trap\_init() (trap.c):**
  + Initialize the trap handler by calling trap\_set\_handler (or equivalent mCertikOS function) to register the trap\_handler function for a specific trap number.
* **Modify subscriber.c:**
  + Call trap\_init() after subscribing to the topic to initialize the trap handler.

## Technical Challenges and Mitigations:

* **Challenge 1: Synchronization:**
  + Solution: Use spinlocks carefully to protect shared data structures. Avoid holding locks for extended periods.
* **Challenge 2: Trap Handling Complexity:**
  + Solution: Carefully study the mCertikOS trap mechanism and ensure that the trap handler is correctly implemented and executed in the subscriber's context.
* **Challenge 3: Memory Management:**
  + Solution: Use container\_alloc and container\_free consistently to adhere to mCertikOS's resource quotas. Check return values from allocation functions and handle failures gracefully.
* **Challenge 4: Deadlock:**
  + Solution: Be careful when using spinlocks to avoid deadlock, avoid nested locks.