Actor Model Implementation in Python
Concurrent Programming with Message Passing

Noah deMers

November 2025

Contents

1 Overview

This report explains the implementation of an actor-based concurrent system in Python. The actor model is a concurrent programming paradigm where independent actors communicate exclusively through asynchronous message passing, providing a clean abstraction for parallel computation without shared state.

2 Core Architecture

2.1 Message Class

Messages are the fundamental communication primitive in the actor model. My `Message` class encapsulates:

```python
@dataclass
class Message:
    msg_type: str
    value: Any
```

Message Type (string): Identifies what action the receiver should take. Examples include `"STORE"`, `"INCREMENT"`, `"SWITCH"`, `"SPAWN"`. This string-based dispatch allows actors to implement different behaviors for different message types without requiring a complex type hierarchy.

Value (any type): The payload accompanying the message. Values can be:

- **Integers**: For numeric computation
- **Strings**: For text processing
- **Actor References**: For forwarding or chaining
- **Tuples**: For bundling multiple values

The flexibility of accepting any type enables rich communication patterns. An actor can send another actor a reference to a third actor, enabling dynamic network topology changes.

Immutability: Once created, messages cannot be modified. This prevents race conditions where multiple actors might try to access the same message simultaneously. Each message represents a single, atomic communication event.

2.2 Actor Base Class

The abstract `Actor` class defines the fundamental behavior all actors share:

```python
class Actor(ABC):
    def __init__(self, name: str):
        self.name = name
        self.message_queue = queue.Queue()
        self.thread = threading.Thread(target=self.run, daemon=True)
        self.running = False
        self.internal_state: Dict[str, Any] = {}
```

Each actor has:

- `name`: Identifier for debugging and logging
- `message_queue`: Thread-safe FIFO queue for incoming messages
- `thread`: Dedicated OS thread running the message loop
- `running`: Boolean flag controlling the actor's lifecycle
- `internal_state`: Dictionary storing actor-specific data

2.2.1 Thread-Safe Message Queue

Each actor owns a `queue.Queue` instance from Python's standard library. This queue is thread-safe by default, using internal locks to ensure atomic operations:

Enqueue (`put`): Adding a message to the queue is atomic. Multiple actors can send messages to the same recipient concurrently without data corruption. The sender never blocks on a full queue (Python's Queue has unbounded size by default).

Dequeue (`get`): Removing a message is also atomic. The receiving actor blocks if the queue is empty, automatically sleeping until a message arrives. This eliminates busy-waiting and conserves CPU resources.

Thread Safety Guarantees: Python's Queue uses a combination of locks and condition variables. When an actor calls `put()`:

1. Acquires a mutex lock protecting the internal deque
2. Appends the message to the deque
3. Signals the `not_empty` condition variable
4. Releases the lock

When an actor calls `get()`:

1. Acquires the mutex lock
2. If queue is empty, waits on the `not_empty` condition variable
3. Removes and returns the next message
4. Releases the lock

This design ensures that the queue's internal state remains consistent despite concurrent access from multiple threads.

2.2.2 Dedicated Thread and Message Loop

Each actor runs in its own `threading.Thread`. The thread executes the `run()` method:

```python
def run(self):
    while self.running:
        try:
            message = self.message_queue.get(timeout=0.1)
            if message.msg_type == "STOP":
                break
            self.handle_message(message)
        except queue.Empty:
            continue
```

The `run()` method continuously:

1. Blocks waiting for a message (with 0.1 second timeout)
2. Retrieves the next message from the queue
3. Dispatches to `handle_message()` for processing
4. Repeats until stopped

Thread Creation: In `__init__`, I create but don't start the thread. This allows actors to be configured before beginning execution.

Thread Start: The `start()` method begins thread execution. Once started, the actor runs independently, checking its queue and processing messages. The thread is marked as a daemon, meaning it won't prevent program termination.

Blocking Behavior: The timeout on `get()` ensures responsiveness. Without a timeout, the thread would block indefinitely, preventing orderly shutdown. With the timeout, the thread checks the `running` flag periodically.

Graceful Shutdown: The `stop()` method sets the running flag to False and sends a "STOP" message. This ensures the actor completes its current message before terminating, preventing partial processing or data loss.

2.2.3 Actor State

Each actor maintains a `internal_state` dictionary for storing values:

- **Private**: Only accessible to the actor's own thread, preventing race conditions
- **Persistent**: Values survive across messages, enabling stateful computation
- **Flexible**: Can store any type of data (numbers, strings, lists, actor references)
- **Isolated**: No other actor can directly access or modify it

The key insight is that state is never shared between actors. If one actor needs another's state, it must send a message requesting it. This eliminates the need for locks around actor state, since only one thread (the actor's own) ever accesses it.

3 Actor Types

3.1 CounterActor: Numeric Computation

The `CounterActor` demonstrates numeric computation and state management.

3.1.1 Purpose

CounterActor stores integer values, performs arithmetic, and forwards results to other actors. This showcases:

- State management (storing values in `internal_state`)
- Arithmetic operations (incrementing counters)
- Message forwarding (sending results to other actors)

3.1.2 Message Handlers

STORE: Saves a numeric value in `internal_state['counter']`.

Example usage: `counter.send(Message("STORE", 10))` sets the counter to 10.

This demonstrates the first required capability: storing values for later use.

INCREMENT: Adds a value to the stored counter and sends the result to another actor.

The message value is a tuple: `(amount, target_actor)`.

Example usage: `counter.send(Message("INCREMENT", (5, other_actor)))` adds 5 to the counter and sends the new value to `other_actor`.

This demonstrates both computation (arithmetic operation) and message forwarding (sending results).

GET: Displays the current counter value. Useful for debugging and verification.

Example usage: `counter.send(Message("GET", None))`.

3.1.3 Design Rationale

The INCREMENT operation exemplifies actor model principles:

Encapsulation: The counter state is private; other actors can't read it directly. They must request it via messages.

Computation: Arithmetic happens within the actor's thread, avoiding concurrent access to shared data.

Communication: Results are sent via messages, not accessed through shared memory.

Chaining: Results can be forwarded through multiple actors, creating processing pipelines.

3.2 StringSwitcherActor: String Manipulation

The `StringSwitcherActor` demonstrates string manipulation and message transformation.

3.2.1 Purpose

StringSwitcherActor manipulates strings and message structure, showcasing the flexibility of message-based communication. This shows that actors can handle non-numeric data and perform transformations.

3.2.2 Message Handlers

STORE: Saves a string value in `internal_state['stored_string']`.

Example: `switcher.send(Message("STORE", "Hello"))`.

SWITCH: Swaps the message type and value fields, then sends the new message to a target actor.

The message value is a tuple: `(string_value, target_actor)`.

Example: If the message is `Message("SWITCH", ("NewType", target))`, the actor sends `Message("NewType", "SWITCH")` to the target.

This demonstrates message transformation and meta-programming capabilities.

REVERSE: Reverses the stored string and sends it to the target actor specified in the message value.

Example: After storing "Hello", `switcher.send(Message("REVERSE", target))` sends `Message("RESULT", "olleH")` to the target.

3.2.3 Design Rationale

The SWITCH operation is particularly interesting because it manipulates message structure itself. This demonstrates that messages are first-class data that actors can transform, not just pass through. String manipulation showcases that actors can handle any data type, not just numbers.

3.3 SpawnerActor: Dynamic Creation

The `SpawnerActor` demonstrates dynamic actor creation and management.

3.3.1 Purpose

SpawnerActor creates new actors at runtime and manages collections of actors, demonstrating the dynamic nature of actor systems. This is crucial for systems that need to adapt to workload changes.

3.3.2 Message Handlers

SPAWN: Creates a specified number of new `CounterActor` instances, starts them, and sends each an initial message.

The message value is an integer specifying how many actors to create.

Example: `spawner.send(Message("SPAWN", 3))` creates three new actors named "Spawner_Child_0", "Spawner_Child_1", and "Spawner_Child_2", each receiving an initial `STORE` message with values 0, 10, and 20 respectively.

BROADCAST: Sends the same message to all spawned actors.

The message value is a tuple: `(msg_type, msg_value)` describing what to broadcast.

Example: `spawner.send(Message("BROADCAST", ("GET", None)))` sends a GET message to all spawned actors.

COUNT: Reports the number of actors spawned so far.

Example: `spawner.send(Message("COUNT", None))`.

3.3.3 Design Rationale

Dynamic actor creation is crucial for adaptive systems. The spawner demonstrates:

Scalability: New actors can be created in response to workload changes.

Hierarchy: Actors can manage collections of child actors, enabling hierarchical structures.

Broadcasting: One-to-many communication patterns enable efficient multi-target messaging.

Initialization: New actors can receive initial configuration messages, allowing flexible setup.

The spawner maintains references to created actors in `internal_state['spawned_actors']`, demonstrating how actors can manage complex relationships.

4 Threading and Synchronization

4.1 Thread Safety Mechanisms

My implementation uses several thread safety mechanisms to ensure correct concurrent behavior.

4.1.1 Queue-Based Communication

I use Python's built-in `queue.Queue` rather than implementing my own locks because:

Correctness: Queue is well-tested and handles edge cases properly. It's part of Python's standard library and has been debugged over many years.

Efficiency: Uses optimized C implementations of locks and condition variables, much faster than Python-level implementations.

Simplicity: No need to manually manage locks, avoiding common pitfalls like forgetting to release locks or acquiring in wrong order.

Standard: Following Python conventions makes the code more maintainable and understandable.

Alternative approaches like manual locks (`threading.Lock`) or lock-free data structures would be more error-prone without significant performance benefit for this scale.

4.1.2 Actor Isolation

The actor model's key insight is avoiding shared mutable state. Each actor:

Owns its state: No other actor can directly access or modify it. All access must go through message passing.

Processes sequentially: Messages are handled one at a time, ensuring no concurrent access to state within the actor.

Communicates via copies: Messages contain data, not references to shared state. Even if message values are objects, each message is independent.

This eliminates the need for locks around actor state, as only one thread (the actor's own) ever accesses it. The only shared resource is the message queue itself, which is protected by the Queue's internal locks.

4.1.3 Message Immutability

Messages are implemented as `@dataclass` instances:

Encapsulation: `msg_type` and `value` are bundled together in a single object.

Effectively Immutable: While Python dataclasses aren't immutable by default, I never modify messages after creation.

Safe Sharing: Multiple actors can reference the same message without interference, since they can't modify it.

4.1.4 Blocking vs Non-Blocking

My implementation carefully balances blocking and non-blocking operations:

Non-Blocking Sends: `actor.send(message)` returns immediately, regardless of whether the recipient has processed the message. This is crucial for asynchrony—senders don't wait for receivers.

Blocking Receives: Actors block on empty queues, sleeping until messages arrive. This is efficient (no busy-waiting) and responsive (immediate wakeup on message arrival).

Timeout on Blocking: I use `queue.get(timeout=0.1)` rather than indefinite blocking. This allows periodic checking of the `running` flag, enabling graceful shutdown.

4.2 Race Condition Avoidance

Traditional concurrent programs often have race conditions when multiple threads access shared data. My actor model avoids this by:

No Shared State: Actors don't share memory beyond message queues.

Explicit Communication: All communication is through explicit message sending.

Sequential Processing: Each actor handles one message at a time.

Queue Atomicity: Message queue operations are atomic, protected by locks.

The only shared data structures are the message queues themselves, which are protected by the Queue's internal locks. All critical sections are handled by the Queue implementation, not my code.

4.3 Deadlock Freedom

The actor model naturally avoids many deadlock scenarios:

No Lock Ordering: Actors don't acquire multiple locks in complex orders, so circular wait conditions are impossible.

No Blocking Sends: Sending never waits for acknowledgment, so senders can't deadlock waiting for receivers.

No Circular Waiting: Actors don't wait for each other synchronously.

However, actors can still form logical deadlocks if the application logic creates circular dependencies. For example, if Actor A waits for a message from B, and B waits for a message from A, they deadlock. This requires careful design of message protocols.

5 Test Scenarios

5.1 Test 1: Basic Counter Operations

Tests fundamental counter behavior:

- STORE: Initialize counter value
- INCREMENT: Compute new value and forward to another actor
- GET: Verify state

This test validates that:

- State persists between messages
- Arithmetic operations work correctly
- Messages can be forwarded to other actors

Output shows sequential message processing with correct counter state management.

5.2 Test 2: String Switching

Tests string manipulation:

- STORE: Save string

- SWITCH: Transform message structure
- REVERSE: Perform string computation

This test validates that:

- Non-numeric data is handled correctly
- Messages can be transformed
- Computed results can be sent to other actors

5.3 Test 3: Dynamic Actor Spawning

Tests actor creation and management:

- SPAWN: Create multiple actors at runtime
- BROADCAST: Send messages to all spawned actors
- COUNT: Verify number of created actors

This test validates that:

- Actors can be created dynamically
- Collections of actors can be managed
- One-to-many communication works

5.4 Test 4: Asynchronous Behavior (Critical Test)

This is the most important test, demonstrating true concurrency. I:

1. Create multiple actors
2. Send many messages rapidly without waiting
3. Measure send time (should be very fast)
4. Observe interleaved processing

Key observations proving asynchronous behavior:

Non-Blocking Sends: All messages are sent in milliseconds (typically ¡ 1ms), proving sends don't wait for processing. If sends were synchronous (waiting for processing), sending 8 messages would take at least 8× the message processing time.

Concurrent Processing: Output shows messages from different actors interleaved, with both actors processing messages simultaneously. If execution were sequential, all messages from one actor would complete before any from another.

Order Independence: Messages may be processed in different orders than sent, demonstrating asynchrony. This shows that actors process messages as they arrive, not in a predetermined order.

True Parallelism: On multi-core systems, different actors may execute simultaneously on different cores. Even on single-core systems, thread switching provides concurrency.

This test definitively proves my system is truly concurrent, not just sequential with a message-passing facade.

5.5 Test 5: Chain Communication

Tests complex communication patterns:

1. Counter1 increments and sends to Counter2
2. Counter2 increments and sends to Counter3
3. Counter3 increments and sends to Printer

This validates:

- Messages can flow through multiple actors
- Each actor adds its contribution
- Actor references can be passed through messages
- Complex processing pipelines work correctly

Output demonstrates a value flowing through the pipeline, accumulating increments at each stage.

6 Design Decisions

6.1 Why Python's Queue?

I chose Python's built-in `queue.Queue` for thread-safe message queues because:

Robustness: Extensively tested, used in production code worldwide.

Efficiency: Implemented in C, much faster than Python-level code.

Simplicity: Abstracts away lock management, reducing bugs.

Compatibility: Standard library, no external dependencies.

6.2 Why Abstract Base Class?

Using Python's ABC (Abstract Base Class) for Actor:

Contract Enforcement: Subclasses must implement `handle_message()`.

Code Reuse: Common functionality (queue, thread, send) is inherited by all actors.

Polymorphism: All actors can be treated uniformly despite different implementations.

Documentation: The base class clearly defines what an actor is.

6.3 Why Dataclass for Messages?

Using `@dataclass` for Message:

Automatic Initialization: No need to write `__init__` manually.

Readable String Representation: Automatic `__repr__` for debugging.

Type Hints: Clear documentation of message structure.

Minimal Boilerplate: Simple, Pythonic way to define data containers.

6.4 Why Daemon Threads?

Actor threads are marked as daemons (`daemon=True`):

Clean Termination: Program can exit even if actors are still running.

Testing Convenience: Tests don't hang if shutdown isn't perfect.

Reasonable Default: Most actors are worker threads that shouldn't block program exit.

For production systems requiring guaranteed message delivery, explicit shutdown with join() would be more appropriate.

7 Concurrency Analysis

7.1 Parallelism Achieved

My system achieves true parallelism through multiple mechanisms:

Thread-Level Parallelism: Each actor runs in its own OS thread. On multi-core systems, different actors can execute simultaneously on different cores.

I/O Parallelism: While one actor is blocked on I/O (printing, file access), others continue processing. Python releases the Global Interpreter Lock (GIL) during I/O operations.

Context Switching: Even on single-core systems, the OS rapidly switches between threads, providing concurrency if not true parallelism.

7.2 Python's Global Interpreter Lock (GIL)

Python's GIL restricts CPU parallelism but doesn't prevent concurrency:

GIL Limitation: Only one thread can execute Python bytecode at a time, preventing true CPU parallelism for pure computation.

GIL Release on I/O: I/O operations (prints, file access) release the GIL, allowing other threads to run.

Still Concurrent: Even with the GIL, threads provide concurrency. My test scenario demonstrates this clearly.

Alternative: For CPU-bound workloads, multiprocessing (separate processes) bypasses the GIL.

7.3 Scalability

The actor model scales well because:

Independent State: Adding actors doesn't increase synchronization overhead.

Asynchronous Communication: Senders don't wait for receivers.

Dynamic Creation: Actors can be created as needed.

Message Buffering: Queues allow bursty workloads.

Limitations:

Thread Overhead: Each actor needs OS resources (stack, thread control block).

Queue Memory: Unbounded queues can grow without limit if receivers are slow.

GIL Contention: Python threads contend for the GIL, limiting CPU parallelism.

For massive scale (thousands of actors), lightweight threads (green threads, async/await, or actor frameworks like Akka) would be more efficient.

8 Extensions and Improvements

8.1 Implemented Beyond Requirements

My implementation includes several features beyond the assignment requirements:

PrinterActor: Helper actor for visualizing results from other actors.

Chain Communication: Test demonstrating multi-hop message passing through multiple actors.

Error Handling: Try-except blocks around message processing for robustness.

Graceful Shutdown: STOP message for clean actor termination.

Debug Output: Extensive logging showing message flow and state changes.

8.2 Possible Future Enhancements

Production actor systems include many advanced features:

Supervision: Parent actors monitoring children, restarting on failure (Erlang-style supervision trees).

Message Priorities: Priority queues ensuring urgent messages are processed first.

Selective Receive: Pattern matching on message types for flexible message handling.

Remote Actors: Communication across network boundaries for distributed systems.

Actor Pools: Reusable actor instances for efficiency.

Back Pressure: Bounded queues with flow control to prevent memory explosion.

Persistence: Saving actor state for recovery from failures.

Monitoring: Metrics on message rates, queue depths, processing times for observability.

Clustering: Multiple actor systems communicating to form a distributed cluster.

9 Conclusion

This implementation successfully demonstrates the actor model's principles and advantages:

9.1 Required Components Fully Implemented

Three Actor Characteristics:

- Thread-safe message queues using Python's atomic Queue operations
- Dedicated threads running message processing loops
- Three message handling capabilities: store, calculate, forward

Three Actor Types:

- CounterActor: Numeric computation and state management
- StringSwitcherActor: String manipulation and message transformation
- SpawnerActor: Dynamic actor creation and management

Comprehensive Testing:

- Five test scenarios covering all functionality
- Asynchronous behavior test proving true concurrency
- Chain communication demonstrating complex patterns

Proper Threading and Synchronization:

- Thread-safe queue operations protected by locks
- Non-blocking message sending for true asynchrony
- Blocking message receiving with timeout for responsiveness
- Avoidance of shared mutable state
- Race condition prevention through actor isolation

9.2 Key Insights

The actor model provides an elegant abstraction for concurrent programming:

Safety: Isolation eliminates vast categories of concurrency bugs.

Clarity: Message passing is explicit and easier to reason about than locks.

Scalability: Independent actors enable horizontal scaling.

Flexibility: Dynamic actor creation and message routing enable adaptive systems.

Composability: Complex behaviors emerge from simple, independent actors.

By eliminating shared state and using asynchronous message passing, I achieve both safety and performance—the holy grail of concurrent programming.