# The Dapr Actors Building Block

The actor model was proposed in 1973 by Carl Hewitt as a conceptual framework for concurrent computing, a method where multiple processes operate simultaneously. At that time, highly parallel computers were not accessible. However, recent advancements in multi-core CPUs and distributed systems have increased the popularity of the actor model.

In this model, an actor functions as an independent unit of computation and state. Actors remain entirely isolated from each other and never share memory. They communicate through messages. Upon receiving a message, an actor can modify its internal state and dispatch messages to other (potentially new) actors.

The actor model simplifies writing concurrent systems by offering a turn-based access model. While multiple actors can operate concurrently, each processes received messages sequentially. This guarantees that only one thread is active within an actor at any given time, making the development of correct concurrent and parallel systems significantly easier.

To illustrate the usefulness of this approach, consider a scenario where multiple threads simultaneously call a method to increment a counter value...

public int Increment()

{

var currentValue = GetValue();

var newValue = currentValue + 1;

SaveValue(newValue);

return newValue;

}

In traditional programming models, solving such issues often involves introducing locking mechanisms. However, relying on explicit locking mechanisms is prone to errors and can lead to deadlocks, impacting performance significantly.

public int Increment()

{

int newValue;

lock (\_lockObject)

{

var currentValue = GetValue();

newValue = currentValue + 1;

SaveValue(newValue);

}

return newValue;

}

The turn-based access model of the actor model eliminates the need to worry about multiple threads, simplifying the creation of concurrent systems.

public int Increment()

{

int newValue;

lock (\_lockObject)

{

var currentValue = GetValue();

newValue = currentValue + 1;

SaveValue(newValue);

}

return newValue;

}

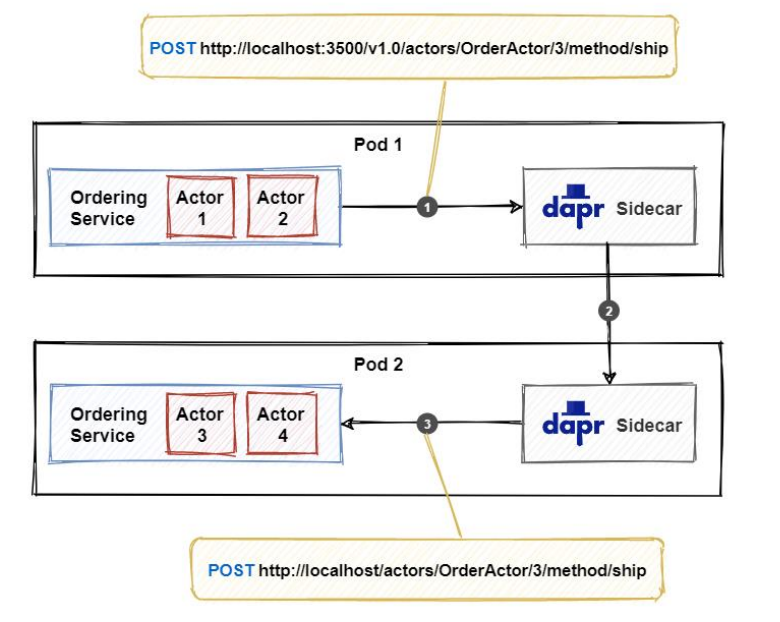
Actor model implementations are typically language or platform-specific. Nevertheless, with Dapr's actor building block, one can harness the actor model from any language or platform.  
**Dapr's implementation is based on the virtual actor pattern introduced by Project "Orleans." This pattern activates actors implicitly when a message is sent for the first time, placing them on a node within the cluster. Additionally, Dapr supports scheduling future tasks using timers and reminders.  
While the actor model offers significant advantages, designing actors requires careful consideration. For instance, having numerous clients call the same actor negatively impacts performance due to serial execution of actor operations.**

To determine if Dapr actors suit a particular scenario, several criteria can be considered:  
• Involvement of concurrency in the problem space, necessitating explicit locking mechanisms in traditional approaches.  
• The problem space being divisible into small, independent, and isolated units of state and logic.  
• Lack of requirement for low-latency reads of actor state, as such reads cannot be guaranteed due to serial actor operations.  
• No need for querying state across multiple actors, which can be inefficient and introduce unpredictable latencies.

One fitting design pattern meeting these criteria is the orchestration-based saga or process manager pattern. A saga manages a sequence of steps required to achieve an outcome, maintaining its current state and triggering subsequent steps. Actors facilitate managing concurrency within the saga and tracking its state.

For scalability and reliability, actors are distributed across all instances of the actor service, tracked by the Dapr placement service. When a new actor service instance starts, the sidecar registers supported actor types with the placement service, updating partitioning information for the actor type and broadcasting it to all instances.

The diagram illustrates an ordering service instance in Pod 1 invoking the "ship" method of an OrderActor instance with ID 3. Due to the actor with ID 3 being placed in a different instance, the call redirects to a different node in the cluster:



The service calls the actor API on the sidecar. The JSON payload in the request body contains the data to send to the actor.  
The sidecar utilizes locally cached partitioning information to determine which actor service instance (partition) hosts the actor with ID 3. In this case, it's the service instance in pod 2. The call is then forwarded accordingly.  
The sidecar instance in pod 2 triggers the service instance to invoke the actor. The service instance activates the actor (if necessary) and executes the actor method.

## Sample application:

Subject: We have an input and output function to calculate the average holding period of books. We will calculate whether the book remains longer than it should.

Create interface called IBookActor which inherited from IActor:

public record struct BookRegistered(int Lane, string LicenseNumber, DateTime Timestamp);

public interface IBookActor:IActor

{

public Task RegisterEntryAsync(BookRegistered msg);

public Task RegisterExitAsync(BookRegistered msg);

}

**Note: Every methods return type must be Task or Task<T> and T must be serializable**

Create class which inherited from IBookActor:

public class BookActor : Actor, IBookActor, IRemindable

{

public readonly ITimingViolationCalculator \_timingViolationCalculator;

private readonly string \_libraryId;

private readonly DaprClient \_daprClient;

public BookActor(ActorHost host, DaprClient daprClient, ITimingViolationCalculator timingViolationCalculator) : base(host)

{

\_daprClient = daprClient;

\_timingViolationCalculator = timingViolationCalculator;

\_libraryId = \_timingViolationCalculator.GetLibraryId();

}

}

ActorHost:

* Is a required constructor parameter of all actors
* Is provided by the runtime
* Must be passed to the base class constructor
* Contains all of the state that allows that actor instance to communicate with the runtime

Implement the entry and exit methods:

Use BookState state the store the books state

public record struct BookState

{

public string LicenseNumber { get; init; }

public DateTime EntryTimestamp { get; init; }

public DateTime? ExitTimestamp { get; init; }

public BookState(string licenseNumber, DateTime entryTimestamp, DateTime? exitTimestamp = null)

{

this.LicenseNumber = licenseNumber;

this.EntryTimestamp = entryTimestamp;

this.ExitTimestamp = exitTimestamp;

}

}

public async Task RegisterEntryAsync(BookRegistered msg)

{

try

{

Logger.LogInformation($"ENTRY detected in library {msg.Lane} at " +

$"{msg.Timestamp.ToString("hh:mm:ss")} " +

$"of book with license-number {msg.LicenseNumber}.");

// store book state

var bookState = new BookState(msg.LicenseNumber, msg.Timestamp);

await this.StateManager.SetStateAsync("BookState", bookState);

// register a reminder for books that taken but don't given within 20 days

await RegisterReminderAsync("BookLost", null,

TimeSpan.FromDays(20), TimeSpan.FromDays(20));

}

catch (Exception ex)

{

Logger.LogError(ex, "Error in RegisterEntry");

}

}

There is a RegisterReminderAsync method which is helping the remind for books that taken but don't given within 20 days.

Unregister the BookLost timer. Set book status.

Publish a new event for book if any violation occurs.

public async Task RegisterExitAsync(BookRegistered msg)

{

try

{

Logger.LogInformation($"EXIT detected in lane {msg.Lane} at " +

$"{msg.Timestamp.ToString("hh:mm:ss")} " +

$"of book with license-number {msg.LicenseNumber}.");

// remove lost book timer

await UnregisterReminderAsync("BookLost");

// get book state

var bookState = await this.StateManager.GetStateAsync<BookState>("BookState");

bookState = bookState with { ExitTimestamp = msg.Timestamp };

await this.StateManager.SetStateAsync("BookState", bookState);

// handle possible timing violation

int violation = \_timingViolationCalculator.DetermineTimingViolationInKmh(

bookState.EntryTimestamp, bookState.ExitTimestamp.Value);

if (violation > 0)

{

Logger.LogInformation($"Timing violation detected ({violation} KMh) of book " +

$"with license-number {bookState.LicenseNumber}.");

var timingViolation = new TimingViolation

{

BookId = msg.LicenseNumber,

LibraryId = \_libraryId,

ViolationInMinutes = violation,

Timestamp = msg.Timestamp

};

// publish timingviolation (Dapr publish / subscribe)

await \_daprClient.PublishEventAsync("pubsub", "timingviolations", timingViolation);

}

}

catch (Exception ex)

{

Logger.LogError(ex, "Error in RegisterExit");

}

}

}

**It’s guaranteed that entry and exit methods will not be working for the same actor. Because of that there is no need to lock mechanism or etc.**