Information Technology Course
Module Individual Project
by Damir Dobric / Andreas Pech

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Implementation of the light-weight Actor Programming Model framework in .NET

***Abstract*—Actor Programming Model (APM) is an asynchronous message-passing model which is used for fine-grained concurrency and distributed-memory applications. The parallelism that is required for building concurrent applications can be easily provided by the Actor Programming Model (APM). The API described in this paper relies on the findings defined in the previous work that demonstrates how the computation of the model of the cortical column can be easily distributed by using the Actor Programming Model (APM) approach. The APM implementation provides a higher level of abstraction that makes it easier to write concurrent, parallel, and distributed systems. This paper helps developers to easily leverage the Actor Programming Model (APM) paradigm and take full control of the distribution of the compute logic through actors. The contribution of a .Net to develop parallel and distributed programs has also been explained in this paper.**

Keywords— *Concurrency, Dotnet, Distributed System, Threads, Computation, Microsoft Azure*

I. INTRODUCTION

The limitations of the current processor technology have been a growing movement towards the use of processors with several cores [1]. The architectures of these processors aim in improving the throughput, efficiency, and processing power of the computer. To achieve this, one of the major hurdles is to build software that can leverage these facilities while still being a tractable programming model [1]. To overcome this Actor Programming Model (APM) has been implemented.

Actor Programming Model (APM) is a conceptual concurrent computation model which came into the picture in 1973 [2]. It has established some of the rules on how the system components should behave and interact with each other. An actor can be represented as a fundamental unit of computation that can perform certain actions such as create another actor, send a message to another actor, and designate how to handle the next message. Actors are lightweight and millions of them can be created very easily. It has its private state and a mailbox, like a messaging queue where a message from another actor can be stored. A message which an actor gets from another actor is processed in FIFO (first in and first out) order. Actors can be considered as the form of object-oriented programming, which communicates by exchanging messages. It has a direct lifecycle, which means they are not automatically destroyed when no longer referenced, once created it is the user's responsibility to eventually terminate them. This enables the user to free up the resources depending upon the need. Actors generally communicate with each other through messages, if they have the address of other actors. The address can be local or remote. The most widely used implementations for the Actor Programming Model (APM) are Akka and Erlang [2].

The main inspiration behind the APM is to take full advantage of the hardware by a naturally simplified programming model to easily solve complex concurrency tasks. The Actor Programming Model (APM) is a mathematical theory and computation model, which addresses some of the challenges posed by massive concurrency [1]. It has been used in the research work related to scaling the cortical learning algorithm called Spatial Pooler [3]. The implementation techniques used in APM have been discussed in this paper. The structure of the paper is as followed: Section II explains the architecture used in implementing the Actor Programming Model (APM). Section III and IV describes how to run the project and under the hub implementation respectively. Finally, the results are presented in section IV.

II. ARCHITECTURE

The architecture used to implement APM is based upon the fundamental constructs of the Actor model; its axioms [1]. Therefore, the following properties such as (i) encapsulation, (ii) messaging, and (iii) location transparency are exhibited by it [1]. *Encapsulation* refers to the bundling of data with the methods that operate on that data. It is used to hide the values or state of a structured data object inside a class. *Messaging* encompasses asynchronous and synchronous message passing mechanisms. Lastly, mobility of an actor refers to location transparency, an essential feature of a fully modular and versatile computational environment [1]. The architecture of the Actor Programming Model (APM) has been designed on top of these properties. The APM architecture improves the scaling mechanism for multiple nodes in a highly distributed system [3]. There are several components involved in the APM architecture which have been described below.

a) Actor:

An actor is a computational unit that responds to a message it receives. The actor is attached to a mailbox from where it receives its messages. The actors are based on the concept

of autonomous processing primitives that communicate solely through message passing. Our framework implements two methods. The first one is the *Ask* method, in which a message is passed, and a response is returned (*request-response*). The second one is the *Tell* method, which sends a message without response (*fire and forget*). In the case of the *Ask* method, the response is returned to the actors via the mailbox. The actor continuously polls the mailbox attached to it to ascertain whether it has messages or not [4]. The mailbox may also be referred to as a message queue that is attached to an actor. The queue in between the service and client assures that actors never directly receive messages from the service. This is to avoid any loss of response messages in case the client is not available. In this framework, the Azure Service Bus has been chosen as a messaging system. It is used because the Azure Service Bus provides the features like ordered messaging via sessions in subscriptions, load-balancing, security, and reliability [5].

b) Client:

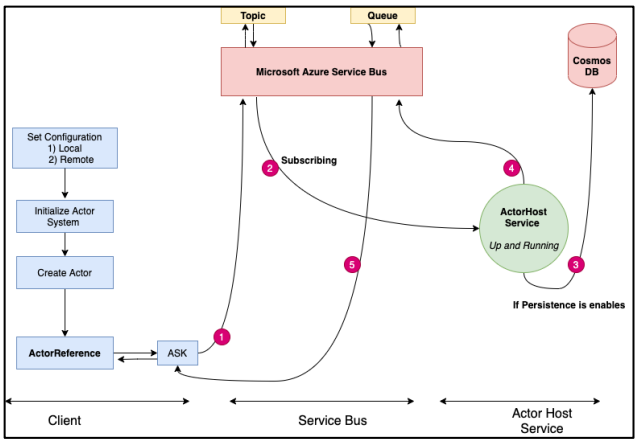
The client implements an actor and sends a message to the service. The messages are sent with the help of the actor by calling its *Ask* or *Tell* methods defined on the object called *ActorReference*. It is important to note that messages are not directly sent to the consumer. They are rather sent to the topic. Consumer, in our case service, is implemented as an actor that is responsible to execute the compute logic at some node in the cluster. The actor is instantiated from the message received at the subscription that is registered on the topic. The reason behind using the topics is that they offer combined and ordered handling of messages and guarantee the ordered delivery. One session can only be executed at a single node. This ensures that the single actor instance of the actor with the given identifier exists in the cluster. This is the most important pillar of the actor programming model [4].

Furthermore, the session property of the service bus has been leveraged here, which allows to identify the messages sent from a specific client to the specific actor and processes them in the right order. It means that messages are processed for one actor in an actor session. In the case of the *Ask* method, the actor replies to the reply queue associated with the client, which has sent the message.

c) Service:

The approach used to implement the Actor Host Service provides a runtime for complex distributed computation on the set of distributed nodes [4]. Service executes actors and results are collected in the queue (mailbox). The approach used to implement the service in APM provides a runtime for complex distributed computation on the set of distributed nodes. To start the computation, the Actor Host Service first subscribes to the messages from the topic. After the computational logic is performed successfully, it then persists the actor in the Azure Table Storage (Azure CosmosDB or Azure Storage). The persistence of the actor enables the service to reuse the actor without recreating it which improves its efficiency. Table Storage is a fully managed NoSQL database provided by Microsoft which is generally used to achieve low latency [5]. Lastly, the service sends the result of the computation to the mailbox (queue) from where the client can retrieve it.

The distributed architecture of the system using actor, client, and service have been shown in Fig. 1. It can be seen that the communication takes place between the client and the host service via service bus. All together five steps are processed when a client sends a message by initiating an actor. These steps have been described briefly below.



1. Client that creates an Actor and sends the message to Topic.2. Actor Service Host subscribes to the message from Topic.3. Actor Service Host persists the Actor in Table 4. Actor Service Host sends a reply to Queue 5. The client gets the reply from the Queue.

Fig 1. The architecture of the distributed system with the dotnetactors

a) *A client sends a request:* As illustrated in Fig. 1, in step 1 a client implements an actor by initializing the actor system and calls its *Ask* method. By calling this method client sends a message to the topic and then waits to get a reply from the queue.

b) *The Actor Host Service processing*: The service subscribes the message from the topic and processes it asynchronously as a part of step 2. After processing, the service persists the actor into the cosmos DB table for future reuse. Lastly, the service prepares the response message and sends it to the reply message queue.

c) *The client receives the response:* In the final step, the client fetches the response message from the queue.

III. How to run the Project (Service & Client)

In this section, a step-by-step guide to run the service and client has been provided. It is evident how easy it is to run a client and service on a local system. This section has been divided into two parts that are part.a and part.b. The first part explains the steps involved in the Actor Service whereas the second part describes the steps involved in the actor client.

a) The Actor Host Service:

Step 1. As a first step service bus connection string needs to be set up as an environment variable as shown in Fig. 2. The process to set up the environment variables is different in various Operating Systems (OS). Follow this link to set it up as per your OS.

Fig 2. Service bus connection string as an environment variable

Step 2. Open the terminal in the *DotNetActorsHost* project folder and enter the below command as shown in Fig. 3. The command-line arguments consists of paramters namely: *SystemName*, *RequestMessageTopic*, *RequestMsgQueue*, *ActorSystemName*, *and SubscriptionName.*

- *RequestMessageTopic*: It is a Service Bus Topic from where the service will subscribe to the client messages.
- *RequestMsgQueue*: It is a Service Bus Queue where the output results are pushed for the client.
- *SubscriptionName*: It is the node for the Topic from where the service should subscribe its messages

The detailed information about the command and its parameters can be found here

Fig 3. Command to start the service

After entering the command hit enter and it will start the service. That's its service is up and running.

b) Dotnet Actor Client:

Step 1. Create an ActorSystem instance and pass the configurations shown in Fig. 4. It is important to note that the configurations should have the same topic name, as used to start the service.

```
ActorSbConfig cfg = new ActorSbConfig();
cfg.SbConnStr = SbConnStr;
cfg.ReplyMsgQueue = "actorsystem/rcvlocal";
cfg.RequestMsgTopic = "actorsystem/actortopic";
//cfg.TblStoragePersistenConnStr = tblAccountConnStr;
cfg.ActorSystemName = "inst701";
ActorSystem sysLocal = new ActorSystem( name: "local", cfg);
```

Fig 4. Code snippet showing actor configurations for client

Step 2. Implement a class that extends the ActorBase and override the *Receive* method as per the need as illustrated in Fig. 5

```
public class MyActor : ActorBase
{
    public MyActor(ActorId id) : base(id)
    {
        Receive<long>( handler: (long num) =>
        {
            receivedMessages.TryAdd(num, num);
            return num + 1;
        });

    }
}
```

Fig 5. Code snippet showing actor base class for client

Step 3. Using the ActorSystem object, create an actor as depicted in Fig. 6. Finally, call the object's *Ask* method to receive the result from the service.

```
ActorReference actorRef1 = sysLocal.CreateActor<MyActor>( id: 1);
var response :long = actorRef1.Ask<long>( msg: (long)42).Result;
Assert.IsTrue(response == 43);
```

Fig 6. Code snippet showing invocation of the actor operation initiated by the client and executed in the Actor Host Service process implemented in the actor.

IV. UNDER THE HUB

In this section, a detailed explanation of the code used to implement the service architecture has been covered. Once the application (*ActorServiceHost*) is started the actor local system is instantiated. To start the compute logic a generalized, *Ask* or *Tell* method can be called. "This method routes the request to the actor running on some node in the cluster, requests the calculation and awaits a result" [3]. The service *DotnetServiceHost*, which hosts the actors, is built in a generalized manner and can be used for any kind of distributed calculation. The logic inside the service is used to receive messages, find actors with the requested identifier, execute the compute logic and send back the result of the computation. The same code with the same configuration executes on every physical node in the cluster [3]. Service executes actors on nodes in the cluster and results are collected in queue [3]. The whole implementation that starts the service is in Program.cs class which is situated in the *DotNetActorsHost* solution in the project folder as shown in Fig. 7.

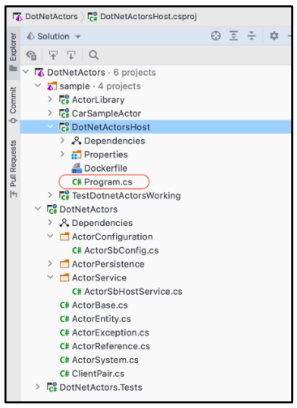


Fig 7. Code snippet showing service project

Program.cs file contains the main method (Fig. 8), which starts the Actor Service Host, which will listen for messages and route them to actors.

```
/// <summary>
/// Represents Main class that initialize the ActorSBHostService
/// </summary>
class Program
{

    static void Main(string[] args)
    {
        Console.WriteLine("Hello HTM Actor Model DotNetActorsHost sample :)");

        ILoggerFactory factory = LoggerFactory.Create(configure: logBuilder =>
        {
            logBuilder.AddDebug();
            logBuilder.AddConsole();
        });

        //--SystemName=node1 --RequestMsgQueue=actorsystem/actorqueue --ReplyMsgQueue=actorsystem/rcvnode1
        ActorSbHostService svc = new ActorSbHostService(factory.CreateLogger(nameof(ActorSbHostService)));
        svc.Start(args);
    }
}
}
```

Fig 8. Code snippet showing the Main method of the service

Once the service has been started it calls the *Start* method which the required variables as shown in Fig.9.

```
string rcvQueue = configArgs["ReplyMsgQueue"];
if (!String.IsNullOrEmpty(rcvQueue))
    throw new ArgumentException(message: "ReplyMsgQueue must not be specified when starting t
cfg.RequestMsgTopic = configArgs["RequestMsgTopic"];
cfg.ActorSystemName = configArgs["ActorSystemName"];
cfg.RequestSubscriptionName = configArgs["SubscriptionName"];
string systemName = configArgs["SystemName"];
```

Fig 9. Code snippet showing configurations set-up for the service

The initialized variables are *ServiceBusConnectionString*, *RequestMsgTopic* (Topic name from where the client messages must be subscribed), *RequestMessageSubscription* (Topic Node), *rcvQueue* (Queue where reply need to be pushed), and *TblStoragePersistenceConnStr* (To access the Table where the actors are required to be stored). Once the service is configured successfully, the ActorSystem object is created which further calls the *start* method as shown in Fig. 10

```
TableStoragePersistenceProvider prov = null;

if (String.IsNullOrEmpty(cfg.TblStoragePersistenConnStr) == false)
{
    prov = new TableStoragePersistenceProvider();
    prov.InitializeAsync(cfg.ActorSystemName, settings: new Dictionary<string, object>()) {
}

akkaClusterSystem = new ActorSystem(name: $"{systemName}", cfg, logger, prov);
akkaClusterSystem.Start(tokenSrc.Token);
Console.WriteLine("Press any key to stop Actor SB system.");
```

Fig 10. Code snippet showing start method call by ActorSystem.

The *start* method starts an infinite while loop which keeps on subscribing the messages from the service bus topics. Once the message gets subscribed it calls *RunDispatcherForSession* method is to further process it as shown in Fig. 11. The implemented code can be found at this git repository.

```
_ = RunDispatcherForSession(session, cancelToken).ContinueWith(
    continuationFunction: async (t:Task ) =>
    {
        Interlocked.Decrement(ref runningTasks);
        if (t.Exception != null)
        {
            logger?.LogError(t.Exception, message: $"Session error: {session.SessionId}");
        }

        await session.CloseAsync();

        logger?.LogTrace(message: "Session closed.");
    }); // Task<Task>
```

Fig 11. Code snippet showing RunDispatcherForSession method.

In the *RunDispatcherForSession* method, several steps are executed.

Step 1. At first, the message type is checked if it is null, then an exception is thrown, and execution is terminated which returns the control to the while loop where the message subscription from the topic is happening.

Step 2. In the next step, it is checked if an actor is existing in a cache. If yes, then it is directly loaded from there else it is newly created. This newly created actor is cached using its SessionsID. It is noteworthy to point that the service here uses the session receive as a key to ensuring that messages are sent to the same session id as actor id that is received in order as they have been sent. It is one of the important design decisions that has been taken into consideration while implementing this robust service.

Step 3. In this step, the received message is deserialized and required operations are performed as per its data type as shown in Fig. 12.

Fig 12. Code snippet showing InvokeOperationOnActorAsync method.

In the *actor.Invoke* method the operation defined by the client takes place. If the operation is completed a response message is created for the clients.

Step 4. It is checked if the persistence is enabled that is if the actor needs to be stored or not. If yes, then it is persisted using the method *persistAndCleanupIfRequired* method as shown in Fig. 13

```
public async Task PersistActor(ActorBase actorInstance)
{
    this.logger?.LogTrace(message:"Persisting actor: {0}", actorInstance.Id);

    var serializedEntity:string = SerializeActor(actorInstance);

    ActorEntity actorEntity = new ActorEntity(this.actorSystemId, actorId:actorInstance.Id.IdAsString);
    actorEntity.SerializedActor = serializedEntity;

    await InsertOrMergeEntityAsync(this.table, new ActorEntity(this.actorSystemId, actorId:actorInstance.

    this.logger?.LogTrace(message:"Persisting actor: {0}", actorInstance.Id);
}
```

Fig 13. Code snippet showing PersistActor method

Step 5. After this, a reply is sent to the queue using the method *sendReplyQueueClients.* In this method, the response message that has been created in Step3 is pushed to the queue from where it is available to the client.

V. RECAP

The extensive stream of methods discussed in this paper takes much less time on accounts of implementing the distribute computation logic. The running sample can be illustrated by the following images. In the sample, a client has been implemented that starts the computation of 500 actors. Around 500 integers, numbered from 1 to 500 have been passed by the client which is then computed by the service. The outputs have been shown below. At first, single service node has been started. The service logs are shown in Fig. 14

```
Time Elapsed 00:00:08.31
Chetans-MBP:DotNetActorsHost brian$ dotnet run --SystemName=HelloCluster --RequestMsgTopic=actorsystem/acto
rtopic --RequestMsgQueue=actorsystem/actorqueue --ActorSystemName=carfunctionality --SubscriptionName=defau
lt
Hello HTM Actor Model DotNetActorsHost sample :)
warn: ActorSbHostService[0]
      ServiceBusTimeoutException
warn: ActorSbHostService[0]
      ServiceBusTimeoutException
```

Fig 14. Actor Host Service running

Now a client has invocated the actor operation by calling the actor's *ASK* method as shown in Fig.15. This method has been called in a loop of 500. For a detailed implementation refer to section III, dotnet actor client.

```
ActorReference actorRef1 = sysLocal.CreateActor<MyActor>(id: 1);
var response:long = actorRef1.Ask<long>(msg: (long)42).Result;
Assert.IsTrue(response == 43);
```

Fig 15. Code snippet showing invocation of the actor operation initiated by the client and executed in the Actor Host Service process implemented in the actor.

The client-side logs after computing the *ASK* method have been shown in Fig.16.

```
Chetans-MBP:CarSampleActor brian$ dotnet run --shallRun=true
info: Program[0]
      Persist Car attributes started....
info: Program[0]
      Loaded Configuration, Messaging-Queue:actorsystem/rcvlocal, Message-Topic:
actorsystem/actortopic
info: Program[0]
      Created ActorSystem
info: Program[0]
      Creating multiple Actor references
info: Program[0]
      Received result: True
info: Program[0]
      Received result: True
info: Program[0]
      Received result: True
info: Program[0]
      Received result: True
info: Program[0]
      Received result: True
info: Program[0]
      Received result: True
info: Program[0]
```

Fig 16. Sample client solution running

Fig. 17 represents the service logs when it computes the subscribed messages.

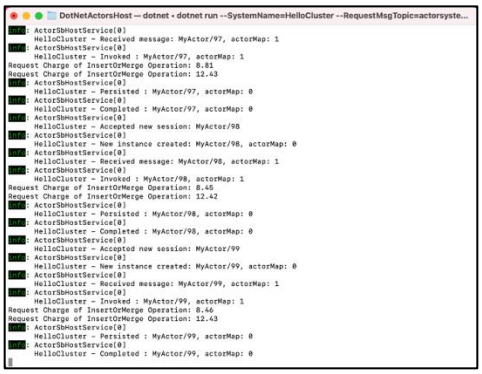


Fig 17. Dotnet Actor Host Service Logs

After the successful computation, the actors are persisted in the cosmosDB as shown in Fig. 18.

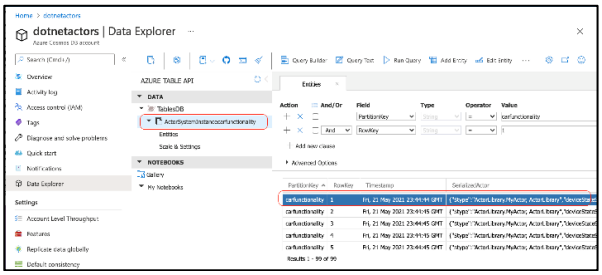


Fig 18. Cosmos DB showing persisted Actors

While the client is still running, five more service nodes have been started and their computation can be seen in Fig.19.

Fig 19. The cluster of actor host service

REFERENCES

[1] K. Aning and K. L. Mannock, "An architecture and implementation of the actor model of concurrency," 2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA), 2017, pp. 1-6, doi: 10.1109/IISA.2017.8316391.

[2] G. Agha, "Actors Programming for the Mobile Cloud," 2014 IEEE 13th International Symposium on Parallel and Distributed Computing, 2014, pp. 3-9, doi: 10.1109/ISPDC.2014.31.

[3] D.Dobric, A.Pech, B.V.Ghita and T.Wennekers, "The Parallel HTM Spatial Pooler with Actor Model" 06 June 2020. Available on: https://aircconline.com/csit/abstract/v10n6/csit100606.html

[4] D.Dobric, A.Pech, B.V.Ghita and T.Wennekers, "Scaling the HTM Spatial Pooler", July 2020 International Journal of Artificial Intelligence & Applications 11(4):83-100, DOI: 10.5121/ijaia.2020.11407

[5] Microsoft Corporation, “Azure Cosmos DB”, Available on: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview