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*—To take the benefit of hardware concurrent computational models is growing rapidly. The advancements in cloud computing technologies such as the Microsoft Azure service bus allows improving the computational speed. This paper illustrates how the controllable, scalable and easy to use actor model system can be implemented. The API described in this paper relies on findings defined in the previous work that demonstrates how the computation of the model of the cortical column can be easily distributed by using Actor Programming Model approach. Actor model implementation provides a higher level of abstraction that makes it easier to write concurrent, parallel, and distributed systems. Actors are objects which encapsulate the state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox. The project described in this paper *dotnetactors* helps developers to easily leverage Actor Model programming paradigm and take a full control of the distribution of actors.**

Keywords— *Concurrency, Microsoft Azure Service Bus, Actors, Threads, Topics, Queues, Azure Cosmos DB*

I. INTRODUCTION

In this era of cloud computing and distribution, systems concurrency has played a vital role in achieving fast results with low latency. Concurrency is a property of the system to execute multiple activities at the same time. It means how components should work in a concurrent computational environment.

The actor model is a conceptual concurrent computation model, came into the picture in 1973[1]. It establishes some of the rules on how the system's components should behave and interact with each other [2]. An actor in the actor model can be represented as a fundamental unit of computation and it can perform actions such as create another actor, send a message and designate how to handle the next message. Actors are lightweight and millions of them can be created very easily. Also, it is important to note that it takes fewer resources than threads. An actor has its private state and a mailbox, like a messaging queue. A message which an actor gets from another actor is stored in the mailbox and 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. Also, actors have a direct lifecycle that is they are not automatically destroyed when no longer referenced, and once created it is a user's responsibility to eventually terminate them. This enables the user to control how the resources are released. Furthermore, in the case of distributed environments actors can communicate with each other through messages if they have the address of other actors. Actors can have local or remote addresses [1]. The most widely used implementations for the Actor model are Akka and Erlang [1].

The main inspiration behind the actor model is to take full advantage of the hardware by using concurrency. Concurrency means that the ability of the system to perform different tasks simultaneously or out of order without affecting the outcome. DotNetActors is an essential element in the actor model and is responsible to handle the communication between the client and Microsoft's Azure Service Bus. It was motivated by the research project [4] in the field of artificial intelligence to scale the cortical algorithm in hierarchical temporal memory Spatial Pooler [4].

The paper is structured as follows; Section II and III covers the general workflow and how to run the projects respectively. Section IV explains under the hub implementation. Results are presented in section IV.

II. GENERAL WORKFLOW

The main objective of this paper is to implement a .NET API that is very easy to use and provides high computation. To implement such a robust system, the latest high-end technology has been used such as Microsoft's Azure Service Bus, Topics, Queues & Cosmos DB. Following is a brief description of them:

a)*Azure Service Bus:* "Microsoft Azure Service Bus is a fully manages enterprise message broker with message queues and publish-subscribe topics" [3]. It provides many benefits such as load balancing the load across different workers, transferring the data safely, and coordinating transactional work. The service bus is a platform as a service (PaaS) with an additional feature that azure takes care of such as Logging, managing space, handling backups, worrying about hardware failure, etc. The main protocol used is Advanced Messaging Queuing Protocol (AMQP) 1.0. In a Service Bus, namespaces are the containers for all the messaging components. Multiple queues and topics can be in a single namespace. "A Service Bus namespace is your capacity slice of a large cluster made up of dozens of virtual machines" [3]. Due to this, it provides all the

availability and robustness benefits on a very large scale.

b) *Queues (Azure Service Bus):* In our implementation queues are used to send the replies to the clients. In general, queues can send and receive messages. Messages are stored in the Queues until the receiving application has received and processed them. Fig. 1 depicts the queue with a sender and a receiver.

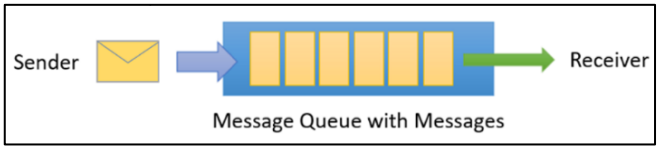


Fig 1. Microsoft Azure Queue [3]

On arrival, the messages in queues are ordered [3] and timestamped. It is spread across availability zones once accepted by the broker. It is always held in triple-redundant storage [3]. Once the client has accepted the message service bus never leaves the messages in memory or volatile storage. Messages are only delivered when requested, which means they are delivered in pull mode. A pull model is not like busy-polling mode, the pull operation can be long-lived and gets completed once the message is available [3].

c) *Topics (Azure Service Bus):* In our implementation topics are used to receive the messages from the clients and to dispatch execution of the actor at the node connected to the subscription. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the topic. As stated above queues are used for point-to-point communication, whereas topics on the other hand are used in publishing/subscribe scenarios.[3]. Fig. 2 depicts a topic with one sender and multiple receivers.

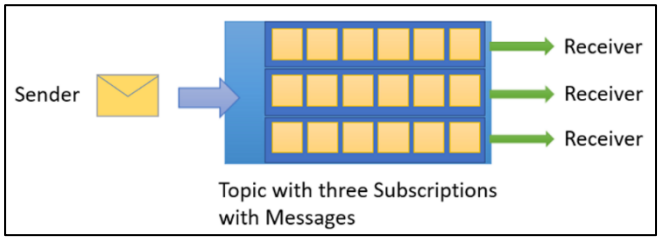


Fig 2. Microsoft Azure Topic [3]

The key difference is that multiple and independent subscriptions can attach to a topic, apart from this it works exactly like a queue. Copy of each message that is sent to a topic is received by a subscriber. “Subscriptions are names as entities” [3]. A set of rules can be defined in a subscription known as a filter which can keep a check on which message to be copied to a subscription and an optional action that can modify the message metadata [3].

d) *Azure Table Storage:* In our implementation tables are used to store the actors so that they can be loaded easily. Azure Table Storage or Cosmos DB Table API serves as storage for structured data in the cloud with schema-less design. This makes it easier to adapt the data as per the requirement of the application. Table storage can be utilized for storing flexible datasheets and other types of metadata as per the service requirements. Table storage is consisting of a storage account, storage table, and entities, as shown in Fig. 3. [2]

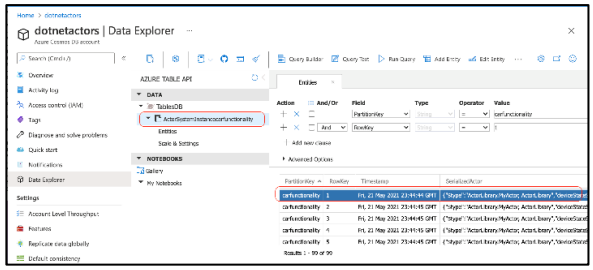
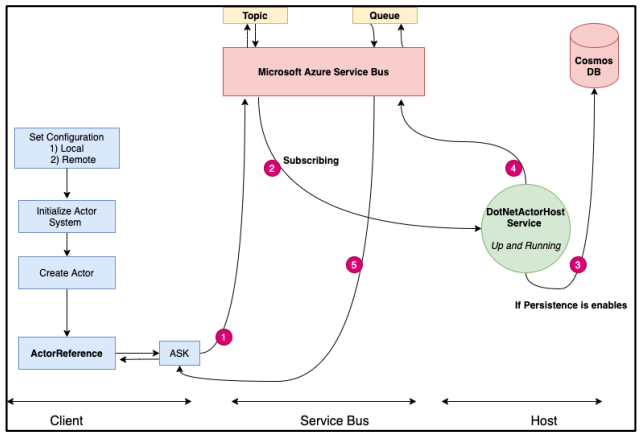


Fig 3. Microsoft Azure Cosmos DB Table [2]

Using the above-mentioned technologies, a potent infrastructure has been implemented as shown in Fig4. It can be seen that the communication takes place between the client and the dotnet actor host service via service bus. The client implements an actor and sends a message to the service bus (topics), Host subscribes the message from the topics and processes it. After that service stores, the respective actor in a table and then sends a reply to the service bus (queue). The client then gets the reply from the service bus (queue). All these steps have been numbered in Fig. 4.



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

Fig 4. General Workflow

As illustrated in Fig. 4 five steps get processed when a client sends a message. To get a better understanding these steps have to be divided into client and service steps. Two steps are processed by the client whereas three steps are processed by the service. These steps have been explained in three parts below.

a) *A client sends a request:* As illustrated in Fig. 4, in step 1 a client implements an Actor by initializing the Actor System and calls its *Ask()*

method. By calling the *Ask()* method client sends a message to the Topic and then waits to get a reply from the reply message queue.

b) *DotnetActor Host Service Processing*: As shown in Fig4, in step2. service subscribes the message from the Topic and processes it asynchronously. After processing the actor message, as a next step service stores the actor into the Cosmos DB table for future use. After that service prepares the reply message and sends it to the reply message queue as shown in Fig. 4, step 4.

c) *The client receives the response:* In the final step that is step 5 client gets the reply message from the Queue.

As shown above, it can be inferred out that how easy and quickly an actor system can be established to achieve the high computation. Let us have a look at the detailed explanation of the code used to implement such a powerful system in the next section.

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. An illustration fig to each step has also been added.

a) Dotnet Actor Service:

In this section steps to run instructions for the service have been written. Each step has a figure that provides more information to execute that step.

Step1. In the first step service, bus connection string needs to be set up as an environment variable as shown in the Fig. 5.

Fig 5. Service bus connection string as an environment variable

Step2. Open the terminal in the *DotNetActorsHost* project folder and enter the below command as shown in Fig. 6

dotnet run
--SystemName=HelloCluster
-- RequestMsgTopic=actorsystem/actortopic
-- RequestMsgQueue=actorsystem/actorqueue
--ActorSystemName=actorsystem
--SubscriptionName=default

Hit enter after entering the command. It will start the service

Fig 6. Service bus connection string as an environment variable

That's its service is up and running.

b) Dotnet Actor Client:

In this section steps to implement and run the client are provided. An illustration of each step is also provided to ease the implementation.

Step 1. The first step is to implement a method and create an ActorSystem object. Then pass the configurations as shown in Fig. 7. The configurations should have the same topic, 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 7. Code snippet showing actor configurations for client

Step 2. Implement a class that extends the ActorBase class and implement the *Receive()* method as per the requirement as shown in Fig. 8

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

    }
}
```

Fig 8. Code snippet showing actor base class for client

Step 3. Using the ActorSystem object, create an actor as shown in Fig. 9. Then call *Ask()* method to get the result from the service. That's it, run this method from the Main() method.

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

Fig 9. Code snippet showing actor creation for client

IV. UNDER THE HUB

In this section, a detailed explanation of the code used to implement the architecture as illustrated in Fig. 4 has been covered. As shown in section II, Fig. 4 firstly the client creates an actor and calls the *Ask()* method. Service then processes the message by subscribing it from a topic and sends the reply to the queue where it is available to the client. Let us discuss about Actors, Client, and Service in more detail.

a) ACTOR:

An actor is a container that encapsulates the state, behavior, and mailbox. An actor can modify its state and behavior. It provides the high-level abstraction for writing concurrent and distributed systems. It makes it easier for a developer to write correct concurrent and parallel systems. In the implementation ActorBase.cs class is the parent class that is extended by all the client actor classes as shown in Fig. 10.

Fig 10. Code snippet showing Actor base class

It can be seen in Fig5 that the *ActorSampleLibrary* project implements MyActor.cs class that extends the ActorBase class. Now as per the requirement clients' needs to add a *Receive()* method. The implemented *Receive()* method registers the receive handler, which means how the service should respond to the received data type. In the Fig6 highlighted method is for a String, which means the service will invoke this action when a String data type is received.

b) CLIENT:

The client creates an actor and sends a message to the service. To implement an actor in a client, firstly an ActorSystem should be implemented by providing the basic configurations as shown in Fig. 11.

```
CancellationTokenSource src = new CancellationTokenSource();
var cfg :ActorSbConfig = GetLocaSysConfig();
logger?.LogInformation( message: "Car sample started with reply message queue as "+cf
logger?.LogInformation( message: "Loaded Configuration, Messaging-Queue:"+cfg.ReplyMs
ActorSystem sysLocal = new ActorSystem( name: $"CarFunctionalityTest", cfg);
logger?.LogInformation( message: "Created ActorSystem");
```

Fig 11. Code snippet showing ActorSystem initialization

In the configurations information such as service bus connection string, service bus queue name, service bus topic name, and the actor system name should be provided as shown in Fig. 12.

```
private static ActorSbConfig GetLocaSysConfig()
{
    ActorSbConfig cfg = new ActorSbConfig();
    cfg.SbConnStr = Environment.GetEnvironmentVariable("SbConnStr");
    cfg.ReplyMsgQueue = "actorsystem2/rcvlocal";
    cfg.RequestMsgTopic = "actorsystem/actortopic";
    cfg.ActorSystemName = "actorsystem2";
    return cfg;
}
```

Fig 12. Code snippet showing general configurations

After successfully creating the Actor System, an actor can be created by using the method *CreateActor()* as shown in Fig. 13. Once the actor is created then the *Ask()* method can be called which will start step 1 as shown in general workflow Fig. 4, section II.

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

Fig 13. Code snippet showing general configurations

c) SERVICE

The service, *DotnetActors* has been designed and implemented in C# .NET Core, because it integrates very easy with the service bus. Approach used to implement the service provides a runtime for complex distributed computation on the set of distributed nodes [4]. Service executes actors on nodes in the cluster and results are collected in queue [4]. To describe the functionality firstly, the messages that have been published in the topic by the clients are subscribed by this service. The service then performs the operations, mentioned by the client, on the subscribed messages. Once the operations are successfully performed then a reply message is created by the service which is then pushed to the queue. To understand the service functionality in more detail an example of the service has been implemented by the class Program. cs. It is situated in the *DotNetActorsHost* solution in the project folder as shown in Fig. 14.

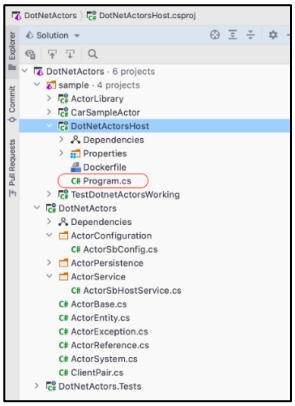


Fig 14. Code snippet showing service project

In the Program.cs file there is the main method that starts the Host application is shown in Fig. 15

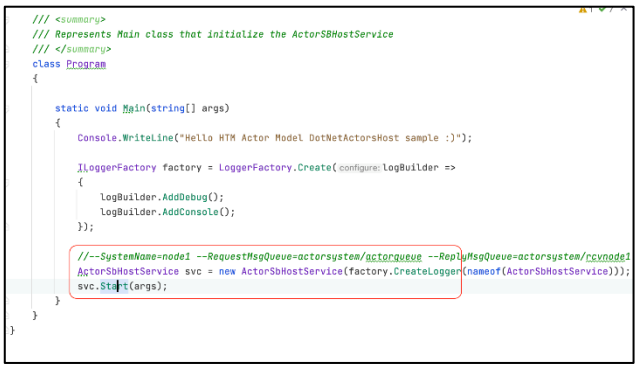


Fig 15. Code snippet showing the Main method of the service

To run the Service the following command-line arguments must be passed that are *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

It is important to note that clients should have the same configurations as the service is having else the communication will not take place.

Also, before starting the service it is important to set up the ServiceBus connection string as an environment variable else the service will throw a null pointer exception. It cannot be passed as a command-line argument for security reasons. Once the service has been started the first method that is being called is *Start()* method.

Fig 16. Code snippet showing configurations set-up for the service

As shown in Fig. 16 when the *Start()* is called the service initializes its variables with values. The variables that are initialized 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).

After the service is configured successfully, the ActorSystem object is created. In the next step, the ActorSystem object calls the *start()* method as shown in Fig. 17

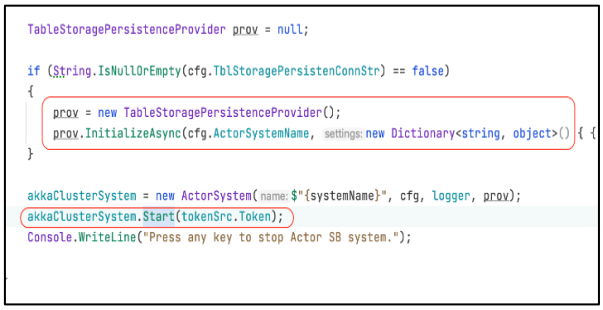


Fig 17. Code snippet showing start method call by ActorSystem.

In the *start()* method an infinite while loop starts which only stops when a cancellation token is passed to it. Otherwise, it keeps on subscribing to the topic messages from the service bus. Once the message is received *RunDispatcherForSession()* method is called to process the received message as shown in Fig. 18.

Fig 18. Code snippet showing RunDispatcherForSession method.

In the *RunDispatcherForSession()* method several steps are executed. Let's discuss them turn by turn

Step1. 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.

Step2. In the next step, it is checked if an Actor is existing in a cache. If it exists, then it is directly loaded from there. If not then it is checked if the actor has been persisted in the database, if yes then it is loaded from there. In the case of an actor, it is not found in the database too, then it is considered as a new entry and is stored in the cache with the key as SessionsID. It is important to note that the service 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.

Step3. In this step, the received message is deserialized and required operations are performed as per its data type as shown in Fig. 19.

Fig 19. 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.

Step4. 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()* as shown in Fig. 20

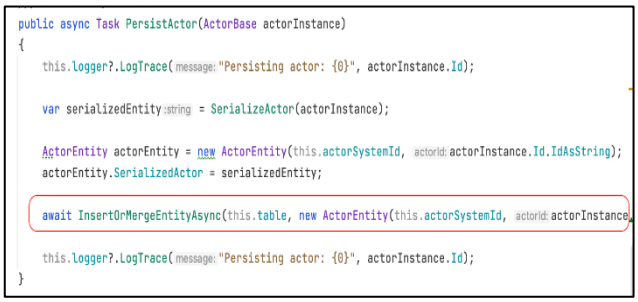


Fig 20. Code snippet showing PersistActor method

Step5. After this, a reply is sent to the queue using the method *sendReplyQueueClients()*. In this method, the response message that was created in Step3 is pushed to the queue from where it is available to the client.

V. RESULTS

The extensive stream of methods discussed in this paper takes much less time on accounts of running the experiment. It's due to the robustness of the Microsoft Azure Service Bus and our implemented actor model system. The following images illustrate the running code.

Fig. 21 shows the service logs when the service is running

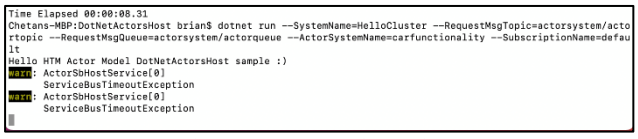


Fig 21. Actor Host Service running

Fig. 22 shows the client logs when the client starts.

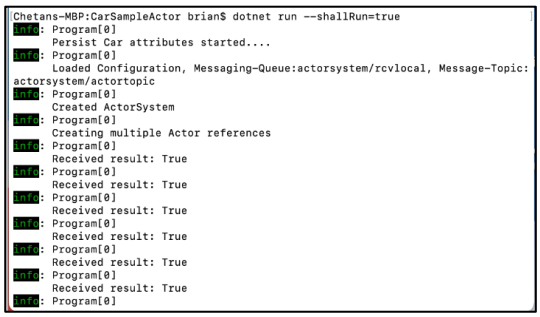


Fig 22. Sample client solution running

Fig. 23 represents the service logs when messages are received from the client.

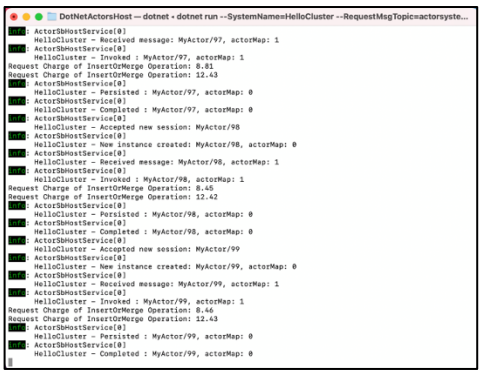


Fig 23. Dotnet Actor Host Service Logs

Fig. 24 shows the persisted entities in Cosmos DB.

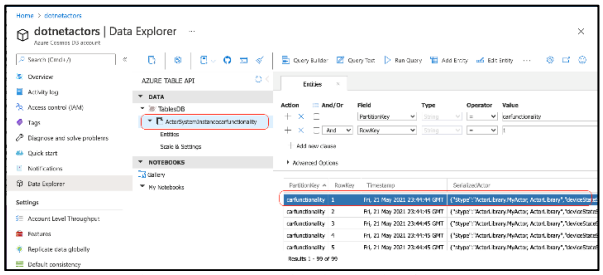


Fig 24. Cosmos DB showing persisted Actors

REFERENCES

[1] Akka.net, "Actors ", [Online]. Available: https://getakka.net/articles/concepts/actors.html

[2] Microsoft Corporation, "Azure Cosmos DB". [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

[3] Microsoft, "Azure Service Bus". [Online]. Available: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

[4] D.Dobric, July 2020, "Scaling the HTM Spatial Pooler", July 2020. [Online]. Available: https://www.researchgate.net/publication/343605270_Scaling_the_HTM_Spatial_Pooler

[5] B.Storti, "The actor model", July 2015. [Online]. Available: https://www.brianstorti.com/the-actor-model/