Monoids for Monadic Composition

Dr. Ratnesh Prasad Srivastava, Dr. Rajiv Singh

Assistant Professor, Assistant Professor
Department of Information Technology
College of Technology GBPUAT Pantnagar, India.

Abstract
This paper deals with designing an asynchronous concurrent framework built to produce constraints of correct by construction by designing a mechanism where components can perform their operations independently without disturbing the order of their communication and by dealing concurrency side effects. We have developed three different types of monoids (function under composition) to establish communication and coordination between sender and receiver.

Keywords: Promise, Monads, Actors, Concurrency, Eventloop, Frameworks, single thread model, Correct by Construction.

I. INTRODUCTION

The actor model proposed by us uses two queue (Mailbox for storing message to process and callback queue for storing processed response) as shown in figure 1.

Fig. 1: Request/Reply in actor-based framework

Every actor writes their messages to other actor’s mailbox for sending their request to be processed. The response sent by another actor is written back to callback queue of the actors. Every actor reads it’s mailbox queue after reading the messages of it’s callback queue. Finally, each actor is copied to global actor space of a single thread. Thus works as a single threaded model and each actor may work as publisher and subscriber. We are using our actor-based framework as shown in figure 1. It has different actors with their mailbox and callback queue.

The motivation for the presented research work is drawn based on answering the set of following questions and identifying the gaps in research reported in literature so far.

Why promises are required?
Why to design Promise Based Actor Framework (PBAF)?
How to design the PBAF?
Why not use any architectural language to represent architecture of PBAF system?
Why the Petri net has been selected as modeling language to model composed system?
What are the related works in Petri net based composition system and how the current model is different than others?
How the PBAF is flexible than already available component models?
What is the significant improvement by using asynchronous over synchronous communication and promises over callback?

This dissertation tried to answer all of the questions in the consequent chapters. The following details mentioned in the table I, summarizes the issues addressed to find the research gaps. The table II summarizes the steps to logically conclude the requirement of promises and table III logically conclude the purpose and process adopted while designing Promise Based Actor Framework (PBAF).

Examples of pure actor languages include Erlang [6], SALSA [7], E [8], AmbientTalk [9], and Kilim [10]. The major benefit of pure actor languages is that the developer gets strong safety guarantees: low-level data races are ruled out by design. But java based actor libraries such as ActorFoundry [11], Actor Architecture [12], ProActive [13], AsyncObjects [14], JavaCT [15], and Jetlang [16] have in common that they do not enforce actor isolation, i.e., they cannot guarantee that actors do not share mutable state. The java based actor libraries have been introduced prior to the release of java 8, thus do not use lambdas and monads as functional programming constructs.

Motivated by Category theory [18] and paper [19] we decided to use promise as monads [20] to control concurrency and Actors to maintain state and provide message endpoints. This paper is organized as followed: Section II introduces the proposed concurrency based actor model framework and challenges associated in designing the framework, Section III discuss the hypothesis 1, that unifies object and thread to design actor model, Section IV discuss the hypothesis 3, which uses monoids to establish indeterministic monadic composition and
side effects occurring due to concurrency. Finally Section VII introduces conclusion and future work.

**TABLE I:** Summary shows need of PBAF

<table>
<thead>
<tr>
<th>Why to design Promise Based Actor Framework (PBAF)?</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong></td>
<td>2009</td>
</tr>
<tr>
<td><strong>2.</strong></td>
<td>2009</td>
</tr>
<tr>
<td><strong>3.</strong></td>
<td>2012</td>
</tr>
<tr>
<td><strong>4.</strong></td>
<td>2008</td>
</tr>
</tbody>
</table>

**TABLE II:** Logical conclusion of promise requirement

<table>
<thead>
<tr>
<th>Why Promises are Required?</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sr.No.</strong></td>
<td><strong>Year</strong></td>
</tr>
<tr>
<td><strong>1.</strong></td>
<td>2005</td>
</tr>
<tr>
<td><strong>2.</strong></td>
<td>2015</td>
</tr>
<tr>
<td><strong>3.</strong></td>
<td>1997</td>
</tr>
<tr>
<td><strong>4.</strong></td>
<td>2016</td>
</tr>
<tr>
<td><strong>5.</strong></td>
<td>2009</td>
</tr>
<tr>
<td><strong>6.</strong></td>
<td>2008</td>
</tr>
<tr>
<td><strong>7.</strong></td>
<td>2015</td>
</tr>
</tbody>
</table>
II. PROPOSED CONCURRENCY ACTOR MODEL

We want to develop a framework which accepts the request as a promise and produces the response as promise. The request as a promise is handled by actor proxy. This promise as a request is passed to promise based concurrent abstraction layer that, provides facility of concurrency with side effects by composing two or more services. Consider a system like in figure 2. It contains a serving counter which accepts request from actors A1, A2, A3,......, An and serving requests provided by these actors only. So these actors act as dynamic requesters (sender) and server (receiver), dynamic because the requester may act as server and server may act as a requestor. The figure 3, shows customer and chef demonstration, where a customer may act as a chef and chef may act as a customer. The counter is like a message channel where the requests are enqueued. Actor A2 is requesting A1 to process the chicken message. This message will be enqueued in channel(counter) as fA2, A1, Chickeng written in A1’s mailbox for further processing. Similarly A1 orders A2 to process Egg, so channel enqueues fA1, A2, Egg and A1 writes message Egg to A2’s mailbox for further processing. Actors acting as chef reads messages from their mailbox, filter out the messages, process it and writes the response to its CallbackQueue. Finally, a customer reads the CallbackQueue and gets its response.

Thus, customers places their orders in the mailbox of receiving actors and receiving actors process those messages and they put the processed messages in the callback queue of sending actors. So here we have two queues, one for storing messages to be processed and another for processed messages. The purpose of taking these two queues is to separate the storage of sender and receiver. Sender uses callback queue and receiver uses mailbox queue. Counter represents a loop which is always ready to accept messages as and when they arrive so it acts as an event loop.

An actor can be regarded as an advanced version of thread with data structures. So we can choose a dedicated thread to an actor or more than one actor assigned to a particular single thread. We get single threaded logic by binding data structure to a thread under execution.

In our framework, a message using a method call is executed asynchronously and a call to a method is automatically transformed into a message and later put into the queue(actor’s mailbox). All the messages stored in the queue are processed asynchronously in a single threaded environment.

III. MONADIC COMPOSITION THROUGH MONOID

You can take any sequential program and turn it to purely functional program by doing continuation passing style(CPS) which transforms to basically sent state as argument to the continuation and we have sequential program that is totally functional if it was imperative. So there is a need of the concept of promise, if it is implemented in functional language which says about to a process of a message, what is the function that respond to the next message?

This promise is a monad which is indeterministic which means the timing of occurrences of events are not decided. This can be in 3 different states at a time i.e. resolved, unresolved & error. So composing these promises may produce side effects. The function under composition using these promises are monoids. The side effects in concurrency has been discussed further in this section. Concurrent Monadic Composition with Side Effects We have identified 4 different types of side effects in Concurrency:

i) Uncertainty

ii) Dependency
iii) Latency
iv) Failure

These side effects are dealt by designing the monoids. The three different kinds of monoids named as Terminator, Pipelining and Connector have been designed.

A. Uncertainty

Due to nondeterministic threading behavior promise may have 3 different possibility of states at any given time. When a program tries to access the object which checks the status of value in a promise and receives the value if ready. The program also interact with the proxy which has a reference to the program. Promises provide unified control over both synchronous and asynchronous control. Promises are created by an explicit construct using lambda expression which delegates a thread for computing promise value. The value of promise can be consumed by passing it to proxy or callback.

The promise is created using following snippet of code presented in listing 1:

```java
public Promise(T Error, Object Error)
{
    this.Result = result;
    this.error = error;
    hadResult = true;
}
```

Callback and the Promise both are lambda function. Callback is created at caller side and runs on the callee. Promise is created at callee side and the caller then registers a lambda receiving the result thrown by the callee. Promises (Monadic Composition) are the value of asynchronous computation i.e. a Monad[20] describing an effect.

Promises are a kind of wrapper for values in asynchronous context and they can be in any one of 3 different states i.e. unresolved, resolved and exception. Promises holds the value and value can be represented as function and function can be represented as value in functional programming language. So the value or function producing the value held by promises can be brought from a context to asynchronous context and can be tossed anywhere in the program using lambda expression, lambdas facilitates closures (capturing thing out of scope).

![Message exchange mechanism by Actors](image.png)

**Fig. 2:** Message exchange mechanism by Actors
A value can be substituted from one context to another using lambda expression. So its a unit function returns lambda expression. Unit because the function takes a single value from the environment and does not change value except put it to different container or context (thread) environment. To put a function from one context of sender to another context of receiver (i.e. a functor), instead of value, can be implemented by applying function complete(Object result, Object error) and put this to different context promise. So complete(Object result, Object error) method is called twice once by callback to retrieve value if its not available and put it to promise context and called by promise listener to retrieve a value from the promise. Using the complete method as callback we ensure to deal the uncertainty of double firing of callback, receiving value using callback listener and finally promise listener on complete(Object result, Object error) method is used to retrieve a value from promises. Hence lambda expression as unit function puts the value from one context to another context and complete(Object result, Object error) function puts a function from one context to another context.

B. Dependency
Promises are objects so it must have getters and setters. Getters are used to retrieve a value from the promise and as stated many a times these values can be either in a state of resolved, unresolved or error. Similarly, setters are used to set values for properties resolved as result or error i.e. value settled in time. The monoids discussed below solves the issues of dependency.

1. Termination Monoid: This is a termination monoid where "after sending value to receiver, sender does not get any response back from receiver". In figure 3, Methods can have either void or promise return type, promises are used only when we need to synchronize method calls with other methods, we may have functions they don’t return any type in the case, functions acting as terminating function. In listing 1, boolean variable "hadResult", inside the promise constructor shows that promise can have three states, resolved, unresolved or error so it represents a function having side effects i.e. monad. So promise is a monad. The unresolved state help in providing non blocking facility with synchronization where as resolved state provides completion or termination of sender/receiver with ordered message execution.

Using this monoid, sender is trying to send a value as parameter to receiver to be computed by receiver. So for transferring value, control must be switched from sender thread to receiver thread. So this monoid overrides function then(Callback()), which works as an abstraction to mutual exclusion, which further overrides the complete() method with fairness abstraction. Receiver computes its function once control is transferred to it and again calls complete() method with no arguments to transfer control back to sender with promise having no value. So this monoid has function of fairness abstraction, mutual exclusion abstraction and receiver function under composition.

2) Pipelining Monoid: The representation of above algorithm in code structure is shown in listing 2. This is a pipelining monoid where after sending value to receiver, sender gets response back from receiver.

Listing 2: Pipelining Monoid

```java
public <OUT> IPromise<OUT> thenAnd(final Function<T, IPromise<OUT>> function) {
    Promise res = new Promise<>();
    then (new Callback<T>() { @Override
        public void complete(T result, Object error) {
            if (Actor.isError(error)) {
                res.complete(null, error);
            } else {
                function.apply(result).then(res);
            }
        }
    });
    return res;
}
```

The code structure method thenAnd () presented in listing 2, shows that, a monoid which takes a function as a parameter (because it’s a functional composition) and that function run outside the context of this monoid i.e. running on some other thread separately. The job of the monoid function thenAnd () is to create a promise and wrap the transformed promise inside the promise created. So this thenAnd monoid returns a promise of promise. The significance of this is that it holds a promise computed by a function which may be in three different states at a time. For example if P1=new promise() and p2=add(int 2, int 2), so res above in figure will return P2(4), so it will be equivalent to P1=promise(P2) i.e. p2 of type promise. So <OUT> here is p2 and outer most <OUT> is p1. Function thenAnd() monoid is a sender function which has the reference of receiver function as the parameter function.

Return type of thenAnd () shows that it returns promise of promise i.e. a kind of pipelining, a monoid type and this monoid obeys rule of associativity by using then(res) which makes communication complete from sender to receiver & receiver to sender, and switch is provided through identity function then(Callback) discussed previously because it transfers the same promise to different context(Sender/Receiver) for further computation.
3) **Connector Monoid:** The representation of above algorithm in code structure is shown in listing 3. This monoid gets the promise from supplier and transfer the control to receiver. So in this case “sender is borrowing input as promise from some other supplier and then this promise is transferred to receiver for the purpose of processing”.

```
Listing 3: Connector Monoid

public IPromise<T> thenAnd
(Supplier<IPromise<T>> callable)
{
    Promise res = new Promise<>();
    then (new Callback<T>(){
        @Override
        public void complete
(T result, Object error)
        {
            if (Actor.isError(error))
                res.complete(null, error);
            else
                IPromise<T> call = null;
                call = callable.get().then(res);
        }
    });
    return res;
}
```

IV. CONCLUSION AND FUTURE WORK

Promises as an input and output produced by this framework facilitates correct by construction mechanism. The promises as input is supplied to framework by using 3 different monoids. The monoids are functions under monoids”. In Terminator monoid receiver accepts value from sender & does not produce any response, in pipelining monoids, sender receives response from receiver for it’s request. Whereas in Connector monoids sender supplies input to receiver by 3rd party or supplier. These monoids abstract mutual exclusion, fairness, communication and coordination, that was dealt explicitly in earlier practices of concurrency using multi thread programming.

This framework may further be extended to support distributed application. Callback serialization will definately be interesting to solve in case of distributed programming.

REFERENCES


[17] Stan W. Smith, ”An experiment in bibliographic mark-up: Parsing metadata for XML export”, Proceedings of


[28] Haller; Philipp and Martin Odersky;, "Actors that unify threads and events",Coordination Models and Languages. Springer Berlin Heidelberg, 2007

[29] Brodu, Etienne, Stephane Frnot and Frdric Obl; , "Toward automatic update from callbacks to Promises",AWeS. 2015

AUTHORS’ BIOGRAPHIES

Dr. Ratnesh Srivastava is currently a research scholar at Indian Institute of Information Technology, Allahabad and full-time Assistant Professor in the department of Information Technology, College of Technology, GBUAT, Pantnagar since 2011. Prior his current job, he was working with software industry for 08 years. He has experience of using Java based technologies on various domains including health and banking.

Dr. Rajiv Singh, is presently with the Department of Electrical Engineering, College of Technology, G.B. Pant University of Agriculture and Technology, Pantnagar, Uttarakhand. He has around twelve years teaching and research experience. The areas of his research interests include Control, Instruments and Sensors, Wind, PV and other renewable energy systems, Energy policy studies, Smart materials etc. He has several publications in peer-reviewed journals/conferences of national and international repute along with book chapters published by reputed international publishers. He has also attended several national and international conferences in India.