{^^}Thespian

Overview

Thespian
Python Actor System

By: Kevin Quick <quick@sparq.org>

2020 Mar 10 (#1.5)

Thespian Project

TheDoc-05

PUBLIC DOCUMENT

Contents

1 Home **2**

2 Quick Start **3**

2.1 Simple Installation 3

2.2 Starting a Thespian Instance 3

2.3 Hello World 4

2.4 Hello World Redux 4

2.5 Hello World, Distributed 6

2.6 Hello World, Semi-Deterministic 6

2.7 What Next? 7

3 Documentation **7**

3.1 Thespian Documentation 7

3.2 Background and Related Efforts 7

3.3 Unrelated Efforts 8

4 Download **8**

5 Contribute **8**

6 News **9**

6.1 Blogs and Success Stories 9

1 Home

Introducing: Thespian

Thespian is a Python library providing a framework for developing concurrent, distributed, fault tolerant applications.

Thespian is built on the Actor Model which allows applications to be written as a group of independently executing but cooperating "Actors" which communicate via messages. These Actors run within the Actor System provided by the Thespian library.

Concurrent All Actors run independently within the Actor System. The Actor System may run the Actors as threads, processes, or even sequential operations within the current process—all with no change to the Actors themselves.

Distributed Actors run independently… anywhere. Multiple servers can each be running Thespian and an Actor can be run on any of these systems—all with no change to the Actors themselves. Thespian handles the communication between the Actors and the management process of distributing the Actors across the systems.

Location Independent Because Actors run independently anywhere, they run independently of their actual location. A distributed Actor application may have part of it running on a local server, part running on a server in Amsterdam, and part running on a server in Singapore… or not, with no change or awareness of this by the Actors themselves.

Fault Tolerant Individual Actors can fail and be restarted—automatically—without impact to the rest of the system.

Scalable The number of Actors in the system can be dynamically extended based on factors such as work volume, and systems added to the Distributed Actor System environment are automatically utilized.

One of the key aspects of the Actor Model is that it represents a higher level of abstraction than is provided by most frameworks. When writing an Actor-based application, the concurrency and transport layers are completely abstracted, which both simplifies the design and allows the concurrency or transport to be changed in the future *without* requiring changes in the Actor-based application.

The above qualities of Actor programming make it ideally suited for Cloud-based applications as well, where compute nodes are added and removed from the environment dynamically.

2 Quick Start

2.1 Simple Installation

Install the Thespian library on your local host system using pip:

```
$ pip install thespian
```

2.2 Starting a Thespian Instance

Start a default Thespian Actor System on the current host system:

```
$ python
>>> from thespian.actors import *
>>> ActorSystem("multiprocTCPBase")
```

2.3 Hello World

A Hello World example:

```
from thespian.actors import *

class Hello(Actor):
    def receiveMessage(self, message, sender):
        self.send(sender, 'Hello, World!')

if __name__ == "__main__":
    hello = ActorSystem().createActor(Hello)
    print(ActorSystem().ask(hello, 'hi', 1))
    ActorSystem().tell(hello, ActorExitRequest())
```

The above will create an Actor, send it a message and get a "Hello World" response, and then tell that Actor to exit because it is no longer needed.

```
$ python helloActor.py
Hello, World!
$
```

Note that any additional `ActorSystem().ask()` calls following the tell of `ActorExitRequest` above will return None because the target actor does not exist

... except the first such `ask()` call, because an Actor is always called with its `ActorExitRequest` message before being killed; this allows the Actor to perform shutdown activities, which is just sending another greeting back in the case of the Hello Actor, and the tell does not read a response, so the next `ask()` will return that queued response.

2.4 Hello World Redux

It's easy to extend the Hello World example to include multiple actors which communicate with each other. This example shows a number of additional details:

- the messages exchanged between the Actors can be anything that can be pickled.
- Actors can create other Actors dynamically
- Actor Addresses can be passed around
- Actors remain until they are removed by sending them an `ActorExitRequest()`

```
from thespian.actors import *

import sys
```

```

class Greeting(object):
    def __init__(self, msg):
        self.message = msg
    def __str__(self): return self.message

class Hello(Actor):
    def receiveMessage(self, message, sender):
        if message == ’hi’:
            greeting = Greeting(’Hello’)
            world = self.createActor(World)
            punct = self.createActor(Punctuate)
            greeting.sendTo = [punct, sender]
            self.send(world, greeting)

class World(Actor):
    def receiveMessage(self, message, sender):
        if isinstance(message, Greeting):
            message.message = message.message + ", World"
            nextTo = message.sendTo.pop(0)
            self.send(nextTo, message)

class Punctuate(Actor):
    def receiveMessage(self, message, sender):
        if isinstance(message, Greeting):
            message.message = message.message + "!"
            nextTo = message.sendTo.pop(0)
            self.send(nextTo, message)

if __name__ == "__main__":
    hello = ActorSystem().createActor(Hello)
    print(ActorSystem().ask(hello, ’hi’, 0.2))
    print(ActorSystem().ask(hello, ’hi’, 0.2))
    ActorSystem().tell(hello, ActorExitRequest())
    print(ActorSystem().ask(hello, ’hi’, 0.2))
```

Running above will create an Actor and send it a message. That Actor will create two other Actors, passing the message along to the first which then passes it to the second before finally sending the message back to the original requestor.

The original requestor is code outside of the Actor environment. This external requestor uses the `ask()` API call which assigns it an Actor Address just like any other Actor.

```
$ python helloActor.py
Hello, World!
$
```

2.5 Hello World, Distributed

In the previous example, there were three Actors that participated in generating the desired result. One of the key principals of an Actor is that while it may maintain internal state, it does not share state with other Actors. This principal allows each Actor to run independently… in some cases, independently can mean that each Actor is run on a different system!

Thespian will automatically handle delivery of a message to an Actor running on a different system when the `self.send()`, `ActorSystem.tell()`, or `ActorSystem.ask()` functions are called. There is no change needed to the logic of each Actor, aside from a `staticmethod` that can indicate whether the Actor should run on the current system or not.

Thespian uses a "capabilities"-based methodology, along with the notion of a "Convention" of Actor Systems for determining where to run an Actor. On each host system that will potentially run an Actor, an instance of Thespian is started, with parameters to indicate the Convention that this instance should join, along with a dictionary of capabilities for the current ActorSystem instance.

The capabilities are simply a dictionary of values: the keys and values are completely up to the implementer. When `createActor()` is called to create a new Actor, and if that Actor class has a special staticmethod, that staticmethod is called with the current ActorSystem's capabilities dictionary as an argument. The staticmethod returns a `True` or `False` to indicate whether the Actor should be run on the current system, based on examining the dictionary of capabilities. If the staticmethod returns `False` then Thespian will automatically pass the `createActor` operation to other instances in the Convention until one is found where the Actor's staticmethod returns `True`: the Actor will then be created on that system and the Actor's address will ensure that messages are routed to that Actor on that destination system.

The distribution of Actors across multiple host systems is thusly controlled simply by the capabilities that the Thespian instance on each host system is initialized with. The `receiveMessage()` of each Actor is **unchanged** for supporting distributed systems, and the manner and configuration of the distribution is controlled independently of the functioning of each Actor. It is even possible for a Thespian ActorSystem's capabilities to be updated dynamically after startup to reflect changes in the distributed system.

Controlling the distribution configuration independently of the business logic is a powerful separation of concerns that makes it easier to adapt the Actor-based application to changing needs and to scale the application as needed to handle the demand.

2.6 Hello World, Semi-Deterministic

Actors are a concurrency mechanism whereby each Actor runs independently. The invocation of each Actor's `receiveMessage()` method is determined by network delivery and the system scheduler and may change from one run of the Actor-based application to the next. While parallelism is highly useful for efficiency and scalability, it can be very difficult to diagnose or even reproduce issues given this level of variability.

It would be much easier to debug problems and write unit tests if the behavior was deterministic and predictable. This can be done easily for Thespian actor-based applications simply by using the `simpleSystemBase` when starting the application. The Actors themselves are unchanged: the `simpleSystemBase` simply executes each Actor's `receiveMessage()` sequentially in the order that messages are generated. When run multiple times, the same scheduling order will be followed, and the sequential invocation of `receiveMessage()` methods ensures that only one thing occurs at a time.

The downside to the `simpleSystemBase` is that there is no parallelism in execution, and that it does not support distributed configurations. It is therefore highly useful for certain situations (which could include production uses), but it is also very limited in its capabilities. Just as with the distributed Thespian convention arrangement however, this re-configuration of the actor-based application occurs **without** any changes to the actor's internal logic: it's a separation of concerns that is handled without needing to modify code involved in other areas of concern.

2.7 What Next?

This has just been a very simple introduction to using Thespian. A more detailed introduction can be found in the Thespian In-Depth Introduction document, and Using Thespian is the primary API and utilization document.

There are also a number of examples in the code repository itself (`https://github.com/kquick/Thespian/tree/master/examples`), including a hands-on tutorial on using multiple actor systems: (`https://github.com/kquick/Thespian/tree/master/examples/multi_system`).

3 Documentation

3.1 Thespian Documentation

TXT	PDF	HTML	Description
TXT	PDF	HTML	Using Thespian is the main documentation reference for developers writing and implementing Actor-base
TXT	PDF	HTML	Thespian Director is the main documentation reference for the Director utility.
TXT	PDF	HTML	Thespian Developer's Notes provides documentation for developers working on Thespian itself.
TXT	PDF	HTML	Release History is maintained in the Thespian Release Notes.
TXT	PDF	HTML	An In Depth Introduction to Thespian.

Examples can be found in the code repository (`https://github.com/kquick/Thespian/tree/master/examples`), including a hands-on tutorial on creating a distributed system using multiple actor systems: (`https://github.com/kquick/Thespian/tree/master/examples/multi_system`).

3.2 Background and Related Efforts

- The Actor Model at Wikipedia
- A video with Carl Hewitt, Eric Meijer, and Clemens Szyperski discussing the Actor Model
- `https://c2.com/cgi/wiki?ActorsModel`
- Akka is a popular Actor Library for Java and Scala; it is roughly the equivalent to Thespian for those languages.

- Akka.NET is an implementation of Akka for the .NET and Mono environments making it useable for C# and F# applications.
- The Erlang language has Actor support built-in (see `https://www.erlang.org/doc/getting_started/conc_prog.html`).

The Wikipedia site has a good list of Actor Model implementations, but please let us know of others that should be listed here.

3.3 Unrelated Efforts

The following projects or sites are completely unrelated to this site, despite apparent similarities in naming or purpose. These sites are legitimate and appropriate to their domain, however, they have no relation to the Python Thespian project and are listed here to provide clarification on this point and avoid confusion.

- `http://pythonhackers.com/p/DrewEaster/thespian`

4 Download

The recommended way to obtain Thespian is by using `pip` to download the latest copy from PyPi:

```
$ pip install thespian
```

The PyPi page can be consulted directly at `https://pypi.python.org/pypi/thespian`

Thespian source and releases are also available from the `https://github.com/kquick/Thespian` github source maintenance location.

Thespian may also be provided by your system's packaging tools (e.g. yum, rpm, apt-get, etc.).

5 Contribute

Contributions to and involvement in Thespian development itself is welcomed. The Thespian Developer's Notes document provides more details on how to get involved.

- Source Code: `https://github.com/kquick/Thespian`
- Mailing List: thespianpy@googlegroups.com
- Report a Bug: `https://github.com/kquick/Thespian/issues`

6 News

2015-08-31 Thespian Publicly Released

Current and previous Thespian release information is available in the Thespian Release Notes.

6.1 Blogs and Success Stories

- Thespian overview article by Peter Sabaini, 2020 Feb
- Article comparing Actors to Python asyncio/await, 2016 Feb
- GoDaddy blog article introducing Thespian, 2015 Aug

If you have a blog, article, or success story you would like published here, please contact one of the project administrators.