

Partial Actor Continuations

Efficient and Extensible Request Routing for Event-Driven Architectures

Stefan Plantikow
Zuse Institute Berlin (ZIB)
Takustrasse 7
14195 Berlin, Germany
plantikow@zib.de

ABSTRACT

The request routing logic between the different stages in event-driven architectures is often distributed over different portions of the source code. This can make it hard to change and understand the flow of events in the system.

The article presents an approach that allows writing request routing logic as a set of routing scripts. Requests are executed step-wise according to their script by sending partial continuations that encapsulate their respective request's current execution state to stages for local processing and optional forwarding of follow-up continuations. The implementation of a simple domain specific language for routing scripts for the scala actor library is described and evaluated. The results show that request routing with partial actor continuations performs equally or better to using a separate stage for request routing logic for scripts of at least 3 sequential steps.

Categories and Subject Descriptors

H.2.4 [Information Systems]: Systems—*Concurrency*; D.1.3 [Software]: Programming Techniques—*Concurrent Programming*; D.3.3 [Programming Languages]: Language Constructs and Features—*Concurrency*; D.2.11 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Extensibility*

Keywords

Request Routing, Event-Driven Architecture, Partial Continuation, Actor Model, Scala

1. INTRODUCTION

Using a staged, event-driven architecture is an approach to the design of server software that can provide high degrees of concurrency and throughput. This is achieved by structuring the software as a set of stages that communicate exclusively via event queues and by optionally performing admission control on each queue.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS 2009 Nashville, Tennessee USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

```
// TODO Examples of staged architectures are ... //
TODO XtreamFS example
```

Depite their benefits, event-driven architectures can lead to a distribution of application logic over different stages. The implementation of each stage usually resides in a different portions of the source code and therefore the dynamic routing of requests through different stages is not described by a single source location.

This reduces the understandability of the system and in turn makes it harder to modify the request routing logic. Additionally, it makes it difficult to add new request types without changing the source code of existing stages and re-deploying the system.

```
// Problem deScription
// Article overview
```

2. PRELIMINARIES

```
// Actor Model
```

3. REQUEST ROUTING

The application logic of event-driven architectures can be split into two categories. First, processing logic, is the part of application logic that necessarily must be executed at a specific stage in order to access resources or state that are only available locally. Second, the request routing logic describes how a given incoming request is handled by executing processing logic at many stages in some order.

Often, processing logic is contained implicitly in the event handling of single stages and the follow-up events created by them. It is this implicit containment that can make systems difficult to understand and modify.

Additionally, request routing logic may be stateful, i.e. the result of executing processing logic at some stage may determine how and at which stages request handling needs to be continued. This places further burdens on the implementation of single stages, as incoming- and outgoing events need to be amended with the necessary state data, although it might be completely independent from the intended purpose of the stage.

To give an example, imagine a simple system for launching satellites into space. Incoming requests are amended with authentication information in the first stage. In the second stage, this information is then used to authorize the request and eventually launch the rocket. Only after the satellite has begun to operate, some third stage (i.e. the press office) is informed. Now, imagine that the initial request needs to be amended with extra information (name and owner of satellite) for the press office. Passing this in-

formation down requires modifying the events to and from the rocket launching stage with fields for the additional payload, although this extra information is of no importance to launching the rocket.

This intertwining of processing and request routing logic is a case of insufficient separation of concerns, calling for a different way to describe both types of application logic. Next, two different approaches that address this issue are described.

3.1 Ping-Pong-Approach

A straightforward way to deal with this problem is to transfer the execution of the request routing logic to a separate stage. Requests enter the system as separate events at such a routing stage. This stage then continuously forwards events to some stage, waits for a reply, and upon receipt, decides how to continue based on this and previous reply-events for the request. In the following, this will be called the ping-pong-approach.

While this approach allows to write the event flow portion of the application logic in a single stage, it has two disadvantages: First, it requires the creation of additional stages and the associated computational overhead. Second, and more importantly, it results in extra messages between request routing and regular processing stages.

3.2 Partial Actor Continuations

```
// PAC - Approach, requires introducing continuations
somewhere
// Continuations
// Partial Continuations (?)
```

4. SOFLEUSE: A DSL FOR REQUEST ROUTING

```
// Scala and Actors (briefly)
// Plays
// Example
// Describe execution
```

5. IMPLEMENTATION

```
// Scala and Actors (as needed for implementation)
// CPS-Transform
```

6. EVALUATION

```
// Describe setting
// Present results
// Discuss in detail esp w regard to actor library
```

7. RELATED WORK

```
// Let's see...
```

8. SUMMARY

```
// Fazit
// Interesting applications
```

9. ACKNOWLEDGMENTS