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.

// TODO Examples of staged architectures are ... // TODO XtreemFS 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 redeploying 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 implicitely 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 reqeuests 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-

DEBS 2009 Nashville, Tennessee USA

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 continously 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.

The ping-pong approach centralizes routing logic in a single place by introducing an additional stage. This can be understood as the consolidation of a control flow that otherwise would be scattered throughout the source code. Additionally, for each request, the routing stage stores intermediate results and associated state for reuse by later events, as well as provides the ability to pause a request's control flow while waiting for the execution of processing logic by some other stage.

3.2 Partial Actor Continuations

Extracting routing logic requires a mechanism for *pausable, stateful control-flow.* Introducing an extra stage is just one way to achieve this. Other possible techniques are the use of coroutines or continuations the latter on which this article concentrates.

// google arbitrary often

// TODO: More on Continuations (?)

The continuation of a computation describes the part of a computation that yet needs to be computed. Some programming languages, especially the scheme-dialect of lisp, provide means for explicitely capturing the continuation at runtime and by this allow it to be executed arbitrary often. This can be used to implement new control structures inside the programming language.

// Parameterization

This ability may be used to create pausable, stateful control-flow: Each stage only processes messages that actually are anonymous functions that represent the current continuation of some request. Such incoming continuations are executed by calling them with the executing stage as their sole argument. Through this means, request continuations gain access to the functionality of local stages.

When the execution of a request continuation at a stage is about to finish, the follow-up continuation may be captured in a last step. This follow-up continuation is then simply sent to the next stage where request handling continues.

This approach does not require any intermediate stages for the execution of the request logic and through this avoids the additional messages of the ping-pong approach. It is also stateful, since continuations contain all visible state of their stack frame at capture time. On the downside, it requires some overhead for continuation capturing. The impact of these different properties will be shown in the evaluation section.

4. SOFLEUSE: A REQUEST ROUTING DSL

Now a framework for using partial actor continuations, Sofleuse, is presented. Sofleuse has been implemented in the Scala programming language using the scala actor's library. Sofleuse provides a domain specific language (DSL) for writing routing scripts that execute over a set of locally running actors.

Routing script are implemented by subclassing the class Play and overriding the apply method. Play instances are provided with a set of DSL-commands for writing scripts. Commands provide the ability to structure scripts as a sequence of blocks that are each executed at different stages. To capture continuations, commands are chained by using scala's CPS-transforming for-generator-expressions (explained below). The following command set is provided:

- ***v*** **<- remember(*value*)** Bind *value* to *v* for reuse by later routing decisions of the script
- ***v*** **<- compute(*thunk*)** Compute *thunk* at the current local stage. The return value is bound to *v* and may be reused in later routing decisions of the script
- ***v*** **<- computeWith(*o*)(*thunk*)** Like **compute(*thunk*)** but *thunk* takes *o* as its first argument
- ***s*** **<- goto(*stage*)** Continue execution at stage *stage* and return a reference *s* for gainining access to the local processing logic of *stage* (usually *stage* itself)
- ***s*** **<- jump(*thunk*)** Execute thunk at the current (active) stage in order to determine the next stage for script execution. Return a suitable reference *s* to gain access to the local processing logic of that stage
- ***t*** **<- cast[*T*](*stage*)** Like **goto(*stage*)** but cast the result of **goto(*stage*)** to type *T*
- **yield(result)** The yield command of the for-expression may optionally be used to return a result to the initial caller of the script
- **endOfPlay** Syntactically denote the end of a routing script that does not return a value to its caller.

Play places an upper bound on the type of actor-stages over which the routing script commands operate. Routing scripts can also be written as simple for-expressions without using class Play. This may lead to more typecasts and the use of the untyped DSL-commands provided by the object Play. Routing scripts based on simple for-expressions may be run using two additional commands of the DSL:

- **run(*forExpr*)** Run *forExpr* and wait until its execution yields a result (blocks current actor)
- **asyncRun(*forExpr*)** Runs *forExpr* without waiting for a result (non-blocking)

```
def rpc(initialStage, input) = {
  val request = for(
      stageRep < − goto(initialStage)
      result < − compute { stageRep.compute(input) }
  ) yield result
  return run(request)
}
```

Figure 1: Simple RPC in Sofleuse

As an example, consider the execution of a simple RPC call (Fig. 1). The call is wrapped as a function that initially creates a new script based on a simple for-expression. The script itself first transfers the execution to the stage for the rpc using **goto**. Then, the actual rpc is executed at that stage using **compute**, and finally a return value is yielded. To actually execute this routing script for-expression, it is started with **run**.

4.1 Implementation

Sofleuse uses some of scala's unique features, like CPS-transform in generator-expressions, anonymous functions, and self-types.

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

5. EVALUATION

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

6. RELATED WORK

// Let's see...

7. SUMMARY

// Fazit
// Interesting applications

8. ACKNOWLEDGMENTS