The ObjectWeb Consortium

Specification

Perseus:
The concurrency manager
–
Specification

AUTHORS:

S Chassande-Barrioz (France Telecom R&D)

P Dechamboux (France Telecom R&D)

L Garcia-Banuelos

Released: January 27, 2004

Status: Draft

Version: 1.3

Table of contents

TABLE OF FIGURES

1 INTRODUCTION

1.1 Overview

This document presents the definition (API and concepts) of a cache manager component.

1.2 Scope

1.3 Rationale

1.4 Goals

1.5 Document Convention

A Times Roman font is used for the default text.

```
A courier font is used for code fragments.
```

2 Architecture

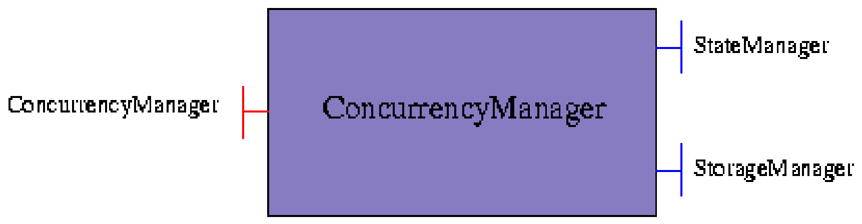

Figure 1: ConcurrencyManager component

The ConcurrencyManager componenent is in charge of the management of concurrent accesses to resources between several contexts (a working set for example). Several policies can be implemented: mutex, pessimistic, optimistic, ... Depending on the policy the use of a dependency manager avoids to create dependency cycle and then dead locks. It is supposed that each resource has an unique identifier (resourceId). Depending on the concurrency policy, a resource can have several states. For example, an optimistic policy requires to isolate contexts using the same persistent objects. The state management depends on the resource type. This document describes a generic concurrency manager which must be extended for each resource type.

2.1 API

The main methods of the interface ConcurrencyManager are:

```
  package org.objectweb.perseus.concurrency.api;
  public interface ConcurrencyManager {
     void begin(Object ws);
```

It demarcates the begining of a working set.

```
     boolean validate(Object context);
```

The `validate` method determines if the working set can be validated. A false value means that the working set must be rolled back and the `abort` method must be called. On the other hand a true value means that the working set can be finalized and the `finalize` method can be called.

```
     void finalize(Object context);
```

It demarcates the normal end of a working set.

```
     void abort(Object context);
```

It demarcates the end of a working set which has been aborted.

```
     Object readIntention(Object context, Object resourceId,
            Object hints) throws ConcurrencyException;
```

It permits to a context to indicate a read intention on a resource represented by its identifier. The concurrency manager must check the locking and provide the state (returned object) to use in this working set.

```
     Object writeIntention(Object context, Object resourceId,
            Object hints) throws ConcurrencyException;
```

It permits to a context to indicate a write intention on a resource represented by its identifier. The concurrency manager must check the locking and provide the state (returned object) to use in this working set.

2.2 Usage of the concurrency manager methods

The following diagram shows how the methods of the ConcurrencyManager must be called for a given working set:

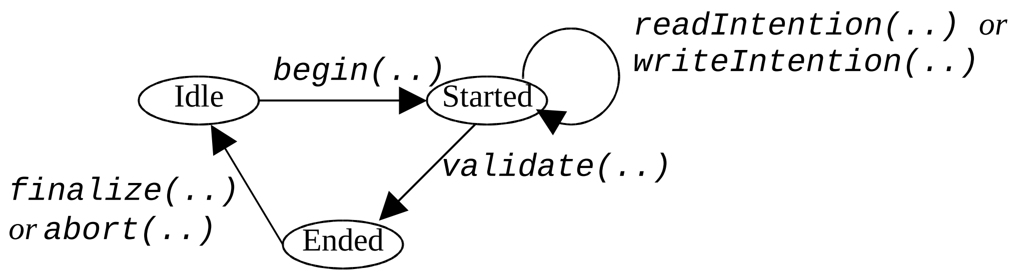

Figure 2: Use of the Concurrency Manager

2.3 Concurrency policies

This interface can be implement with different concurrency policies:

- "Very pessimistic": All accesses are considered as writing with mutual exclusion on the resource access. The readIntention and the writeIntention methods can be blocking operation. Then, a dead lock checker is needed. This mode requires only the use of reference states.
- Pessimistic: Several readers can share a resource, but a writer is alone to access a resource. The readIntention and the writeIntention can be blocking operation. Then, a dead lock checker is needed. This mode requires only the use of reference states.
- Optimistic: The readers and the writers can have access to all resources and the conflicts are resolved at validation step by rolling back some working sets. This mode requires the use of several states in addition to reference states.

In all implementations, the ConcurrencyManager can roll back working sets. To inform the ConcurrencyManager that the working set must be rolled back (due to a conflict or a dead lock) a RollBackException is thrown. In this case, the next validate call will return false.