CUSTOMIZABLE CONCURRENCY CONTROL FOR PERSISTENT JAVA

Laurent Daynès, M.P. Atkinson
and Patrick Valduriez

Abstract: We report on the issues raised when designing a customizable locking mechanism for Persistent Java, a type-safe, object-oriented, orthogonally persistent system based on the language Java. Customizable locking mechanisms are supported by locking capabilities. A locking capability is a bookkeeper of locks that automatically acquires locks with a customizable conflict detection mechanism. It implements the concepts of delegation of locks and ignorable conflicts, automatically keeps track of the dependencies created because of ignored conflicts, and supports the setting of user-defined notifications for conflicts that can’t be ignored. Locking capabilities are one of the primitive components of a more general framework that gives the ability to expert application programmers to implement new transaction behaviors in Java. The framework doesn’t change the Java language specification, and allows the use of any Java classes to implement the body of transactions without change to either their source or compiled form.

7.1 INTRODUCTION

Persistent programming languages offer an attractive alternative to the increasing number of applications whose needs cannot be satisfied with traditional database support. The requirement of these so called non-traditional applications have prompted the development of numerous transaction models whose semantics vary from the traditional transaction model as well as from each other [Elmagarmid, 1992, Barghouti and Kaiser, 1991]. The ever growing proliferation of transaction models, all unable to satisfy all needs, has definitively buried the hope of finding an universal model in the short term, if at all.
the absence of a proper transaction model, most persistent application builders
end up investing a significant amount of time developing in-house transaction
models to circumvent the proposed transaction support in order to better ac­
commodate the needs of their application.

In order to minimize the cost of realizing new transaction models, appli­
cation builders must be offered a simple framework which they can use to
quickly define the transaction behavior they want and to incorporate it into the
persistent programming system. Ideally, these extensions should not require
the programmers to have an in-depth knowledge of how transaction processing
mechanisms are implemented. Furthermore, each addition of a new transaction
model should not require that the system be rebuilt. Instead, the system should
be able to dynamically adjust itself to incorporate these extensions. Lastly,
the user's extensions should be tightly integrated with the system in order to
minimize the impact on the overall performance of the system.

This paper reports on our effort to augment Persistent Java (PJava), an al­
ternative platform for the Java language [Atkinson et al., 1996], with such ex­
tensible transaction management features. The paper specifically focuses on
the issues raised when designing the addition and the implementation of a cus­
tomizable locking mechanism for Persistent Java.

7.1.1 Overview of Persistent Java

The main goal of the Persistent Java (PJava) project is to leverage Java to sup­
port faster development and better maintenance of persistent and transactional
applications (e.g. [Jordan, 1996]) via provision of orthogonal properties. Pro­
viding properties such as persistence and transaction semantics orthogonally
has two benefits.

1. Application programming is not polluted with considerations unrelated
to the application logic itself, such as persistence or enforcement of some
transactional properties. In particular, programmers do not have to ex­
plicitly identify the data that may become persistent or may be used in a
transactional way. Similarly, the standard Java code that would operate
on transient data is used unchanged when it operates on persistent data
or in a transactional context. The addition of the desired property (e.g.,
persistence, persistence + transaction) is achieved by simply composing
the application code with some context-aware code that encapsulates the
particularities of the application requirement (e.g., management of roots
of persistence or monitoring of transaction execution).

2. Any Java classes can be used to build applications in a specific opera­
tional context (non-persistent Java, persistent Java, persistent and trans­
actional Java) without any change to either the sources or the compiled
form of these classes; no extra rewriting/pre-processing or code gener-