Developing a Class Hierarchy for Object-Oriented Transaction Processing

Daniel L. McCue

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK.

Abstract. This paper describes the development of a class hierarchy to support distributed transaction processing. Inheritance and polymorphism, key features of the object oriented programming model, have been used to develop a hierarchy of classes which convey to their subclasses the behaviours of persistence, concurrency-control, recoverability and identity necessary for distributed transaction processing. The development is traced from the requirements of distributed transaction processing to the definition of classes supporting these key properties. The system is interesting in both its development and its results. The development, not based on any rigourous design methodology, illustrates some of the design decisions unique to object-oriented systems. The resulting class hierarchy provides a flexible, object-oriented interface for reliable distributed programming. The paper includes a step-by-step description of the design of the classes and the class hierarchy.

1 Introduction

To perform object-level transaction processing, an application programmer must have a means for accessing and manipulating persistent objects through atomic operations (or operation sequences). This paper explains the development of a class hierarchy which provides such facilities. Starting from interface abstractions, concepts of recovery, persistence, concurrency control, the discussion continues with the development of a complete class hierarchy incorporating facilities for transaction management.

To simplify program access to permanent objects, a programming language facility is provided by which objects can be defined to be persistent (i.e., their state is maintained across program executions). To relieve programmers of the burden of considering the effects of concurrent access to objects or various kinds of system failure during execution, we present a language construct that directly supports atomic actions. Together, these tools allow the programmer to develop robust distributed applications within an object-oriented programming framework.

The remainder of this paper is divided into two parts followed by a brief concluding section. Section 2 describes the evolution of the class hierarchy from the desired properties, at the leaves of the hierarchy, through the necessary supporting classes to the root class, *Checkpointing*. Section 3 describes the class, *AtomicAction*, and its relationship to the property classes. The development of this class library is interesting in itself as an example of an approach to object-oriented design. The concluding section summarises the ideas presented and briefly describes the state of the current implementation of this system.

2 Objects and Atomic Actions

A computational model that has been widely advocated for constructing robust distributed applications uses *atomic actions* (*atomic transactions*) to provide fault-tolerant operations on objects. An object is an instance of some *class*. Each object consists of some variables (its instance variables) and a set of operations (its methods) that determine the externally visible behaviour of the object. The operations of an object have access to its instance variables and can thus modify the internal state. It is assumed that, in the absence of failures and concurrency, the invocation of any of these operations produces consistent (class specific) state changes to the object.

Operation invocations may be contained within *atomic actions* which have the properties:

- *serialisability*
- *failure atomicity*
- *permanence of effect*

The first property ensures that the concurrent execution of programs which access common objects is free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution). Some form of concurrency control policy, such as that enforced by two-phase locking, is also required to ensure the serialisability property of actions. The second property, *failure atomicity*, ensures that a computation either terminates normally (*commits*), producing the intended results (and intended state change to the objects involved) or it *aborts* producing no results and no state change to the object(s). Once a computation *commits*, the results produced are not destroyed by subsequent node crashes. This is ensured by the third property - *permanence of effect* - which requires that any state changes produced (i.e. new states of objects modified in the atomic action) are recorded on *stable storage*, a type of storage which can survive node crashes with high probability. A *commit protocol* is required during the termination of an atomic action to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the atomic action aborts, no updates get recorded [4].

The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. In our model, persistent objects are passive and normally reside in object stores which are designed to be stable. Atomic actions are used to ensure consistent state changes to objects, despite system failures.