Distrib Comput (1992) 6:1–3

DISTRIBUTED COMPUTING

Introduction

Fred B. Schneider *

Fred B. Schneider, Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

This issue of *Distributed Computing* contains four papers concerning specification of concurrent systems. These papers were motivated by the Twenty-sixth Lake Arrowhead Workshop, "How Will We Specify Concurrent Systems in the Year 2000?", held in September 1987 and sponsored by the Western Committee of the IEEE Computer Society. The general chair of the workshop was Leslie Lamport; the program chair was Brent Hailpern.

The purpose of this workshop was to allow comparison and evaluation of different approaches to specifying concurrent systems. Some months before the meeting, participants were provided with an informal description of a serializable database interface and invited to present a formal specification for this system at the meeting. Upon arriving at Lake Arrowhead, participants were surprised with a proposed implementation of the system, and each was asked to indicate how one might prove that this implementation satisfied the formal specification being advocated. Being confronted with a "surprise implementation" helped in determining whether a formal specification precluded correct implementations. The original description of the problem and the addendum describing the surprise implementation are both reproduced below; they were written by Leslie Lamport and revised based on comments provided by Brent Hailpern, Mohamed Gouda, and me.

Three papers giving formal specifications for the serializable database problem were submitted to *Distributed Computing*. These were (repeatedly) reviewed and revised, resulting in the papers published in this issue. Leslie Lamport then studied the final versions of the papers and wrote a critique. Lamport's critique was reviewed by the authors of the papers it discusses and revised in accordance with their comments.

It has taken quite some time for these papers to appear in print. Reading and written formal specifications is apparently rather difficult, and many rounds of revision and review were required before the reviewers believed that the formal specifications corresponded to the informal problem statement. And, as Lamport discusses, there still remain difficulties with the final specifications! Formal specifications may be unambiguous, but unless they are easy to read and write they will not become effective tools for communication. Publication of exercises like this one is intended to promote progress in that direction.

1 Informal description of a serializable database interface

A database consists of a collection of *objects* that can be read and written by a collection of *client programs*. Each client program performs a sequence of *transactions*, where a transaction consists of a sequence of reads and writes to database objects. Several client programs may concurrently execute transactions. Transactions may be aborted either by the client program or by the database system. The result of aborting a transaction is as if none of the reads and writes of the transactions were executed. Serializability means that the values returned by all the read operations from nonaborted transactions are ones that could be obtained by executing these transactions in some sequential order: an order in which all the read and write operations of one transaction are performed before any operations of the next transaction are performed. See Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System" (CACM, November 1976) for more details.

Below is an informal specification of an interface for a database and a sketch of an implementation. You should:

- Write a formal specification that captures the intuitive meaning of the specification.
- Do the same for the implementation.
- Sketch how one proves that the implementation satisfies the specification.

* The photograph and autobiography of Professor Fred B. Schneider were published in Volume 2, Issue No 3, 1987 on page 117

The specification

A client program accesses the serializable database through the following procedure calls, whose informal meanings are given. A single client must wait for the return from a procedure call before issuing another call. However, different client programs may issue concurrent procedure calls.

Begin-Transaction( ): *key* or "failed"

Called to initiate a transaction. There are no arguments, and the returned value is some sort of key that identifies the transaction. The "failed" return indicates that the transaction was not begun. It should occur only if the system has run out of resources – for example, if all keys have been allocated.

Read(*key*, *object*): *value* or "abort"

Called to issue a read operation of the indicated object as part of the transaction specified by *key*. *Read* returns the current value of the object or else the value "abort", indicating that the transaction has been aborted by the database system.

Write(*key*, *object*, *value*): "ok" or "abort"

Writes *value* to *object* as part of the transaction indicated by *key*. The value "ok" is returned if the operation is successfully completed, and the value "abort" is returned if the system has aborted the transaction.

End-Transaction(*key*): "ok" or "abort"

Ends the specified transaction. The value returned indicates if the transaction was completed successfully or was aborted. (This procedure should not be called if the transaction has been aborted by the system.)

Abort-Transaction(*key*):

Aborts the specified transaction. It always succeeds in aborting the transaction.

The last four of these procedures (*Read*, *Write*, *End-Transaction*, and *Abort-Transaction*) can be called only with a key of an active transaction – that is, a key returned by a *Begin-Transaction* call that was not subsequently aborted during a *Read* or *Write* procedure or ended by an *End-Transaction* or *Abort-Transaction* call. When a procedure call results in the completion or abortion of a transaction, the database system may reuse its key. The specification should leave undetermined what happens if a procedure is called with an incorrect key.

The *Read*, *Write*, and *End-Transaction* procedures may return "abort" only if the transaction accesses an object that is accessed by some other, concurrently-executed transaction. (Note that this is *only if*, not *if*; such concurrent access does not necessarily cause the abortion of any transaction).

If is assumed that if a transaction is not aborted, then the client program will terminate it by a call to *End-Transaction* after issuing only a finite number of operations for that transaction. Under this assumption, the system guarantees that control eventually returns from each procedure call.

The implementation

The implementation is based upon the two-phase locking scheme of Eswaran et al. Each object is locked by the transaction as it is needed, and all of a transaction's locks are released when the transaction aborts or completes.

The implementation is a multiprocess program. You may assume that the client program and the implementation are written in a language in which each client program's call to a database procedure initiates a new process. The implementation processes interact with the physical database through the following procedures.

Acquire-Lock(*object*, *key*): "granted" or "rejected"

Release-Lock(*object*)

When an *Acquire-Lock* call has returned "granted", no other *Acquire-Lock* call on the same object can return "granted" until there has been a *Release-Lock* call for that object. Until such a *Release-Lock* call is received, we say that the object is owned by the identifier *key*. An *Acquire-Lock* call returns "rejected" only if deadlock exists – i.e., if the call results in a cycle of calls to

Acquire-Lock(*object*[*i*], *key*[*i*])

that have not yet returned such that *object*[*i*] is owned by *key*[$i+1$], where addition is modulo the cycle length.

A *Release-Lock* call may be issued for an object only after the lock has been granted for that object, in which case the call will eventually return. The *Acquire-Lock* procedure has the property that if every granted lock is eventually released, then every call to *Acquire-Lock* will eventually return. (In other words, individual processes are not starved.)

Read(*object*): *value*

Write(*object*, *value*)

These are the obvious database operations. They always return.

The *Acquire-Lock*/*Release-Lock* procedures do all the necessary deadlock detection, so the implementation is pretty simple, needing to keep track only of the set of objects that have been locked by each transaction.

In sketching the proof that this correctly implements the specification, you should point out where the assumption that the implementation's procedure calls eventually return is used.

2 Addendum to a serializable database interface

To make sure that the specification is not tailored to the particular implementation (two-phase locking) in the original problem, another implementation is given below. Anyone presenting a specification should indicate how they would prove that it is satisfied by this implementation. (Obviously, details are not expected.)

Another implementation

This implementation is based upon the Multiversion Timestamp Ordering algorithm on page 153 of *Concurrency Control and Recovery in Database Systems* by Bernstein, Hadzilacos, and Goodman (Addison-Wesley, 1987). It is described informally, but the description should suffice to enable any competent programmer to code it in the idealized language of his or her choice. (An idealized language is required because, to allow the finiteness of real computers to be ignored, the program uses arbitrarily large integers and sets.)

The implementation uses *timestamps*, which are non-negative integers, as the keys. Timestamps/keys are issued in increasing order, starting from 1.

An *object version* is a record with the following three components:

value: a value
written: a timestamp
latest-read: a timestamp

where $written \leq latest\text{-}read$. The usual record terminology is employed, so "*ov.value*" denotes the *value* component of object version *ov*. For an object version *ov*, let *Interval*(*ov*) denote the timestamp interval [*ov.written*, *ov.latest-read*]. (*ov.written* is the timestamp of the transaction that wrote the object version, and *ov.latest-read* is the timestamp of the latest transaction to read that version.)

For each object *o*, the implementation maintains a set *Versions*(*o*) of object versions *ov*. If *ov* and *ov'* are two distinct elements of *Versions*(*o*), then *Interval*(*ov*) and *Interval*(*ov'*) will be disjoint timestamp intervals. We write $ov < ov'$ if $ov.written < ov'.written$. Initially, *Versions*(*o*) consists of a single object version *ov* with *ov.value* equal to the object's initial value, and $ov.written = ov.latest\text{-}read = 0$.

For each timestamp/key *t* of an active transaction, the implementation maintains a set *DependsUpon*(*t*) of timestamp keys. The elements of *DependsUpon*(*t*) are all less than *t*. (An element *t'* is in *DependsUpon*(*t*) if transaction *t* reads a value written by transaction *t'*.)

The implementation also maintains the following variables:

Aborting: A set of timestamps, initially empty.
LastStarted: A timestamp, initially 0.
LastDone: A timestamp, initially 0.

The action that implements each procedure call is described below. Angle brackets ("⟨" and "⟩") enclose atomic operations. The descriptions employ the *Abort*(*k*) operation, which is defined as follows:

```
Abort(k):
    Aborting := Aborting ∪ {k};
    For each object o:
        For each element ov ∈ Versions(o):
            if ov.written = k
              then Versions(o) := Versions(o) − {o}
            fi;
    return Abort

Begin-Transaction( ):
    ⟨LastStarted := LastStarted + 1;
     DependsUpon(LastStarted) := Φ;
     return LastStarted⟩

Read(k, o):
    ⟨if DependsUpon(k) ∩ Aborting ≠ Φ
        then Abort(k)
        else ov := the largest element of Versions(o) such
                   that ov.written ≤ k;
             ov.latest-read := max(ov.latest-read, k);
             if ov.written ≠ t
                 then DependsUpon(t)
                      := DependsUpon(t) ∪ {ov.written}
             fi;
             return ov.value
     fi⟩

Write(k, o, v):
    ⟨if DependsUpon(k) ∩ Aborting ≠ Φ
        or there exists an element ov ∈ Versions(o) such that
           ov.written ≤ k < latest-read
        then Abort(k)
        else if there is an element ov in Versions(o) with
                ov.written = k
                then ov.written := v
                else Versions(o) := Versions(o) ∪ {ov}
                                    where ov.value = v
                                    and ov.written = k
                                    and ov. latest-read = k
             fi
     fi⟩

End-Transaction(k):
    ⟨wait until DependsUpon(k) ∩ Aborting ≠ Φ
        or
        LastDone ≥ all elements of DependsUpon(k)⟩;
    ⟨if LastDone ≥ all elements of DependsUpon(k)
        then LastDone := k;
             return ok
        else Abort(k)
     fi⟩

    Abort-Transaction(k):
        ⟨Abort(k)⟩
```