Highly Available Atomic Objects

Tony P. Ng
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract
This paper describes an approach to implement highly available atomic objects in a distributed system. Our approach uses the semantics of an object's type to increase availability and concurrency. The increase comes at the expense of writing a layer of type-specific code for concurrency control. The type-specific code can be written without making any assumption of any underlying replication or the serialization order used, allowing it to be re-used more easily than otherwise. We use the virtual partition algorithm[AT86] for replica control. It avoids reading multiple replicas during normal operation at the expense of synchronizing the replicas when they re-establish communication. This optimization is important in the context of semantics-based implementations because potentially no replica is more up-to-date than the others otherwise. In contrast to similar semantics-based approaches, ours assigns quorums to operations of the non-atomic objects used to implement the atomic objects, rather than to operations of the atomic objects.

1 Introduction
This paper describes an approach to implement highly available atomic objects in a distributed system. Transactions accessing atomic objects are recoverable and serialized. A recoverable transaction either commits or aborts eventually. It is never partially completed despite any failure. Serialized transactions appear to have executed serially at the atomic objects despite any concurrency. A large collection of recovery algorithms and concurrency control algorithms have been proposed to implement atomic objects[Gra78, BG81], some of which use the semantics of the object type to increase concurrency[Gaw85, Her90, Ng89, O'N86]. In order to provide high availability, an atomic object can be replicated. A replica control algorithm[Gi79, BG84] can be used to mask the existence of multiple physical copies.

Representing the state of a distributed system as a collection of atomic objects was proposed in [Wei84]. The partitioning into multiple objects allows a potentially different set of recovery, concurrency control, and replica control algorithms to be used in each atomic object, as long as all atomic objects follow a common serialization order to serialize transactions. For example, transactions are serialized in their order of commitment with two-phase locking[Gra78], and serialized in a timestamp order in timestamp-ordering algorithms[BG81]. Our approach to implement highly available atomic objects has the following features:

- The semantics of an object's type is used to increase the availability and concurrency of the implementation. The increase comes at the expense of writing a layer of type-specific code for concurrency control. The type-specific code can be written without making any assumption of any underlying replication or the serialization order used. The type-specific code does not have to be modified if replication is introduced, if the replica control algorithm is changed, or if a different serialization order is used system-wide.

- Our approach is a pessimistic state-based method in the terminology of [Her90]. It differs from other pessimistic state-based methods[Her87a] in that quorums are not assigned to operations of the atomic types, but rather to operations of the non-atomic types used to implement them. In certain atomic types, this may lead to higher availability because only the quorums for the non-atomic type operations that are actually executed are acquired. A quorum assigned to an operation of an atomic type must be large enough for any execution path within the operation.

Our approach is an extension of the work in [Ng89], which describes how to implement concurrent atomic objects. We extend that work to provide high availability by replication.

The rest of this paper is organized as follows. Section 2 describes the terminology and the background. Section 3 describes how to program a concurrent implementation of an atomic object[Ng89]. Section 4 gives an example. Section 5 describes how to increase availability by adding a replica control layer transparently. Section 6 discusses related work.

2 Background and terminology
In this paper, a system is modeled as a collection of objects. An object has a type, which defines a set of possible states and operations that can be invoked on the object. For example, the state of a bank account object may consist of its balance. The bank account object may support deposit and debit operations. Both operations take one argument: the amount to be deposited or debited. A deposit operation increments the balance by the amount deposited. A debit operation decrements the balance if there are sufficient
funds to cover the debit. Otherwise an error is returned and the balance remains unchanged. Borrowing from the notation in [Her87a], an operation is written as \( \langle op/args\rangle/\langle term/\{res\}\rangle \), where \( op \) is the name of the operation, \( args \) denotes the sequence of arguments, \( term \) is a termination condition, and \( res \) denotes the sequence of results. For example, the following are possible bank account operations:

\[
\langle \text{deposit}(30)/\text{ok}\rangle, \langle \text{debit}(20)/\text{ok}\rangle, \langle \text{debit}(40)/\text{no}\rangle
\]

None of these operations return any results. A debit operation has two different termination conditions, depending on whether there are sufficient funds.

Computation are modeled as serialized and recoverable transactions. A transaction may invoke operations on multiple objects. A transaction cannot invoke more than one operation on the same object concurrently. In other words, a transaction can be represented as a sequential process in each object. We call the transactions in a history.

The set of operations performed by an object is its history. Suppose \( T \) is a total order of the committed transactions in a history \( h \). A sequence of the committed operations in \( h \) arranged so that an operation \( o_0 \) appears before another operation \( o_2 \) if

- they are from the same transaction and \( o_1 \) is invoked before \( o_2 \), or
- they are from different transactions \( t_1 \) and \( t_2 \) and \( (t_1, t_2) \in T \)

is called a serialized sequence of \( h \) in the order \( T \).

An object provides a serial specification to define a set of valid sequences. For example, assuming an account has an initial balance of \( \$0 \), its serial specification can be defined to include only those sequences in which an \( \text{ok} \) response is returned to a \( \text{debit} \) operation if and only if the sum of all previous deposits covers the current and all previous successful debits. Consequently, the sequence of \( \text{deposit} \) and \( \text{debit} \) operations above is a valid sequence, but the not sequences

\[
\langle \text{deposit}(30)/\text{ok}\rangle, \langle \text{debit}(40)/\text{ok}\rangle
\]

\[
\langle \text{deposit}(30)/\text{ok}\rangle, \langle \text{debit}(20)/\text{no}\rangle
\]

Suppose \( T \) is a total order of the committed transactions in \( h \). The history \( h \) of operations processed by an atomic object is atomic in \( T \) if the serialized sequence of \( h \) in the order \( T \) is valid. Atomic objects guarantee that their histories are atomic in a common serialization order. A serialized sequence in the serialization order imposes a total order on any two operations. Thus the serialization order imposes a total order on operations as well as transactions.

In this paper, an atomic object is implemented using non-atomic objects. Operations invoked on non-atomic objects are synchronized by short-term mutual exclusion locks. We assume that when a transaction \( t \) is aborted, updates to the non-atomic objects by \( t \) are undone using standard recovery algorithms like shadow paging or write-ahead logging [Gra78]. However, those updates may have been observed by other transactions between the time the mutual exclusion lock is released and the time of abort (yet without violating the atomicity guarantee of the atomic object). Availability is increased by replicating the non-atomic objects transparently (in the sense that there appears to be only one copy of each non-atomic object).

3. Implementing atomic objects

In [Ng89], the state of an atomic object is represented by a non-atomic log object and a non-log component. The log object stores a set of operation records, each of which represents an operation processed by the atomic object. The following procedures are provided to access an operation record \( o \), invoked with a syntax of \( o.procedure<arguments> \):

- \( \text{arg1 = proc() returns(t):} \) Returns \( o \)'s first argument, which is of type \( t \).
- \( \text{match = proc(t: template) returns(bool):} \) Returns true if the operation record \( o \) matches the template record \( t \). A template record is similar to an operation record except that it consists only of the operation name, a termination condition, and, optionally, the status of the operation. Template records are written as \( <op/term> \) or \( <status/op/term> \).

We use a distinguished variable \( \text{this_op} \) in the code in this paper to refer to the operation record created for the current operation invoked on an atomic object.

In order to prevent the size of the log object from growing indefinitely, operation records are deleted when the corresponding operations are aborted. Committed ones are kept in the log object temporarily. Eventually, they are deleted in their serialization order and their effects are remembered more compactly in the non-log component. An atomic bank account object may keep a real number variable to store its balance. Each time a committed operation record is deleted from the log object, the variable can be incremented or decremented to reflect a deposit or debit. Both the log object and the non-log component are non-atomic. If a transaction aborts, recoverability guarantees that any updates on these objects are undone automatically. However, explicit synchronization is needed during an operation on the atomic object for concurrency control. The atomic object's history must remain atomic in the serialization order after the current operation is added. If such a guarantee cannot be made, the operation is rescheduled or aborted.

A log object supports several operations for determining atomicity and rescheduling. We assume a syntax of \( l.operation<arguments> \), where \( l \) is a variable of the type \( \log \). The only exception is the \( create \) procedure, which is invoked by \( \logcreate() \) and returns an empty log object. A log object \( l \) has the abstract value \( O \), where \( O \) is the set of operations inserted into \( l \). Notice that the interface of a log object does not assume any particular serialization order, so the atomic objects may be using the order of commitment or a timestamp order as the serialization order.
0 returns(array Coperation)

The set of committed or to-be-committed operations in \(O\), and \(T\) is the set of committed or to-be-committed operations in \(O\), and \(T\) is their serialization order. The set \(S\) of operations returned satisfies one of the following requirements, depending on \(c\) and \(p\):

1. \(c = \text{definite, } p = \text{after} : S \subseteq A\)
2. \(c = \text{definite, } p = \text{before} : S \subseteq B\)
3. \(c = \text{potential, } p = \text{after} : A \subseteq S \subseteq S'\)
4. \(c = \text{potential, } p = \text{before} : B \subseteq S \subseteq S'\)

The value of \(l\) is unchanged. A trivial but correct implementation may return an empty set for the first two combinations of \(c\) and \(p\), and \(S'\) for the other two combinations. A better implementation should return better approximations of \(A\) and \(B\). The \(\text{filter_match}\) operation is never blocked even if the status or serialization order of an operation \(o'\) in \(O\) is unknown.

\[
\text{filter_match} = \text{proc}(t: \text{template}, c: \text{certainty}, p: \text{position}, f: \text{proctype (operation)} \text{returns(boo1)}, o: \text{operation}) \text{returns(array(operati0n))}:
\]

Suppose \(S' = \{o' \in O : o' \text{ matches } t \text{ and either } f(o') \text{ returns true or } f = \text{nil}\}, A = \{o' \in C \cap S' : (o,o) \in T\}, B = \{o' \in C \cap S' : (o',o) \in T\},\) where \(C\) is the set of committed or to-be-committed operations in \(O\), and \(T\) is their serialization order. The set \(S\) of operations returned satisfies one of the following requirements, depending on \(c\) and \(p\):

1. \(c = \text{definite, } p = \text{after} : S \subseteq A\)
2. \(c = \text{definite, } p = \text{before} : S \subseteq B\)
3. \(c = \text{potential, } p = \text{after} : A \subseteq S \subseteq S'\)
4. \(c = \text{potential, } p = \text{before} : B \subseteq S \subseteq S'\)

The value of \(l\) is unchanged. A trivial but correct implementation may return an empty set for the first two combinations of \(c\) and \(p\), and \(S'\) for the other two combinations. A better implementation should return better approximations of \(A\) and \(B\). The \(\text{filter_match}\) operation is never blocked even if the status or serialization order of an operation \(o'\) in \(O\) is unknown.

**filter_exists** = \text{proc}(t: \text{template}, c: \text{certainty}, p: \text{position}, f: \text{proctype (operation)} \text{returns(boo1)}, o: \text{operation}) \text{returns(boo1)}, o: \text{operation}) \text{returns(boo1)}:

Equivalent to a procedure that tests whether the array returned by \(\text{filter_match}\) contains \(t\).

**delete_first** = \text{proc()} \text{returns(operation signals(unknown))}:

Deletes and returns an operation \(o\) such that \(\forall o' \in C \cap P - \{o\}, (o,o') \in T\), where \(P\) is the set of operations returned in previous invocations of \(\text{delete_first}(C\) defined above). The new value of \(l\) is \(O = \{o\}\). The exception \(\text{unknown}\) is raised and \(l\) is unchanged if the identity of such an operation is unknown.

**wait_delete_first** = \text{proc()}:

Returns when \(\text{delete_first}\) can be invoked successfully with a high probability. It can be used by the caller of \(\text{delete_first}\) to avoid busy waiting. Value of \(l\) is unchanged.

**retry** = \text{proc}(c: \text{retry_condition} signals (abort)}:

Returns when \(c\) is true with a high probability. If \(c\) is unlikely to ever become true, the exception \(\text{abort}\) is raised, which, if left unhandled, will cause the invoking transaction to be aborted. The retry procedure can be used by a program to reschedule an operation that cannot proceed immediately. The readers are referred to [Ng89] for the specification of retry conditions.

How to implement an atomic object using a log object is described in [Ng89]. In general, each atomic object operation has to perform backward validation, in which an operation checks that its results are valid regardless of which operations are serialized before itself, and forward validation, in which an operation checks that the results of operations serialized after itself are not invalidated by its own effects. A similar distinction is made in [H84] in the context of optimistic algorithms, although only one of the two has to be performed.

4 An Example

```plaintext
account = module exports deposit, debit

var 1: log := log.create() \% init'ed to empty
    snapshot: real := 0
    lock: mutex := mutex.init()

while true do \% background process
    1. wait_delete_first() \% wait to delete
    clean_up() \% delete and merge
    end \% while

    clean_up = proc()
        lock.acquire(); begin_transaction()
        o: operation := l.delete_first()
        lock.release();
        return end clean-up
    while true do \% background process
        1. insert(this-op); lock.release();
        end clean-up

    deposit = proc(x: real)
        lock.acquire()
        if l.filter_exists(debit/no, potential, after, bind(exceed-by-less, x), this_op) then 1. insert(this_op); lock.release();
        return end % return ok
        lock.release()
        l.retry(...) \% see [Ng89]
        deposit(x) \% try again
        end deposit

    debit = proc(x: real signals(no)
        lock.acquire()
        if l.filter_exists(debit/ok, potential, after, bind(exceed-by-less, x), this_op) and low_balance_observed(this_op) \% x then 1. insert(this_op); lock.release();
        return end % return ok
        if high_balance_observed(this_op) \% x then 1. insert(this_op); lock.release();
        signal no end \% return no
        lock.release()
        l.retry(...) \% debit(x) \% try again
```
end debit

bind = proc(p: proctype(real, operation))
    returns(bool), x: real)
    returns(proc(ctype(operation) returns(bool)))
q = proc(o: operation) returns(bool)
    return(p(x, o))
end q

return(q) ? q = p with first arg = x
end bind

short_by_less = proc(x: real, o: operation)
    returns(boo1)
end short_by_less

return(low_balance_observed(o) - o.arg1 <= x)
end low_balance_observed

high_balance_observed = proc(o: operation)
    returns(real)
end high_balance_observed

return(accum(potential, <deposit/ok>, o) +
    snapshot - accum(definite, <debit/ok>, o))
end high_balance_observed

return(accum(potential, <deposit/ok>, o) +
    snapshot - accum(definite, <debit/ok>, o))
end high_balance_observed

accum = proc(c: certainty, t: template, o: operation) returns(real)
    value := real := 0
    for each s: operation in
        l.filter_match(t, c, before, nil, o) do
            value := value + s.arg1
    end
end return(value)
end accum

The code in the example implements an atomic
bank account object using a log object and a real num-
er object. In this implementation, a deposit opera-
tion returns ok if there are no (debit/no) operations that may be serialized after it.
self. The test is necessary because a (debit/no) operation assumes that the
balance is less than the amount it was trying to debit.
A (deposit/ok) operation may invalidate that assump-
tion. Concurrency is improved by distinguishing be-
tween large and small (unsuccessful) debits. More pre-
cisely, if the current deposit plus the largest balance
that can be observed by a (debit/no) operation is less
than the amount being debited, the addition of the
current (deposit/ok) operation would not have made
any difference for the debit. Consequently, the cur-
rent deposit operation can be processed immediately
without being blocked. The largest balance that can
be observed by an operation o is computed by adding
the deposits that are potentially serialized before o
and subtracting the debits that are definitely serialized
before o. This upper bound is valid regardless of the
serialization order or the eventual status of the un-
committed operations in the log object. If a deposit
operation is blocked, it is retried later.

The debit operation is more complicated because of
two different termination conditions. A no response is
returned if the highest balance observed by this debit
is less than the amount being debited. An ok response
is returned if the following are satisfied:

- There are no large (debit/ok) operations that may
  be serialized after itself.
- The smallest balance observed by this debit is not
  less than the amount being debited.

5 Highly available atomic objects

Availability of an atomic object can be increased
by replicating the non-atomic objects used in its imple-
mentation. The type-specific code remains un-
changed. In this section we will describe how to repli-
cate a non-atomic log object and the non-log com-
ponent. The non-log component is modeled as non-
read/write objects, which support only read
and write operations (for example, the real number
object used in the example in section 4). We also
prove that the replication is transparent in the sense
that only one copy of each non-atomic object appears
to exist.

5.1 Virtual Partition Algorithm

The replica control algorithm we propose is based on
the virtual partition algorithm described in [AT86].
Each replica of a replicated object is assigned a non-
negative vote. A transaction running in a network
partition can access the replicated object only if the
sum of the votes of the replicas in the partition reaches
a quorum. For a read/write object, read and write
quorums are selected such that their sum exceeds the
total number of votes.

In the virtual partition algorithm, all replicas in the
same network partition are guaranteed to have the
same value1. Consequently, only one replica has to
be read during a read access and all replicas in the
network partition are written during a write access (if
there are at least a read and write quorum of votes
in the network partition respectively). Each node n
maintains a view, consisting of the nodes with which
it believes it can communicate. The view of a node is
updated over time when failures occur or disappear.
Different nodes can have different views. A transac-
tion is aborted if it encounters a view different from
the one possessed by the node on which it is initiated
at the time of the initiation. When a new view is
formed, a two-stepped view update transaction is exe-
cuted. In the first step, a unique view identifier for
the new view is chosen; the readers are referred to [AT86]
for details. The new view identifier is guaranteed to
be larger than the existing view identifiers of the mem-
bers in the new view. In the second step, the value of
the most up-to-date replica of a replicated read/write
object is copied to other replicas in the new view.

1 The virtual partition algorithm described in [AT86] is actu-
ally more general than the one described here. The specialized
version is sufficient for our purposes.
In this paper, the virtual partition algorithm is used to replicate non-atomic objects rather than atomic read/write objects. The algorithm remains unchanged except for the following modifications:

- Since mutual exclusion locks rather than traditional concurrency control algorithms (e.g., read/write two-phase locks) are used to guard access to non-atomic objects, each mutual exclusion lock is replaced by multiple mutual exclusion locks, one on each node that stores replicas of the non-atomic objects it guards. Acquiring/releasing the original lock has to be performed by acquiring/releasing all the replacement locks in the current view.

- Our modified algorithm allows different quorums to be assigned for different classes of log operations. When a log operation $o$ is invoked, the quorum for $o$'s class is looked up and compared with the number of votes in the current view. If there are sufficient votes, the operation can be processed on any replica. Any change to the state of the log object will be written to all replicas in the view. In there are insufficient votes, an exception is signaled, which, if unhandled, will cause the invoking transaction to be aborted. The second step of the view update transaction has to be changed also because a single most up-to-date replica may not exist.

How to assign quorums for the non-atomic objects and create a single most up-to-date replica during a view update transaction is described in the next section.

5.2 Assigning quorums

We assign a read quorum of one to the read operation of a non-atomic read/write object (e.g., the real number object in the example in section 4), and a write quorum of $V_0$ to the write operation, where $V_0$ is the total number of votes. Each replica is given one vote. This assignment provides high availability for the operations of the atomic type (e.g., the deposit and debit procedures in the example in section 4), since they only read the non-log component of the atomic object state. A non-atomic read/write object is written only when an operation record is deleted from a log object and its effects merged into the non-log component (e.g., the clean_up procedure in the example in section 4). Using a write quorum of $V_0$ implies that the deletion and merging can be executed only when all replicas of the read/write object are accessible. We assume that some form of retry is applied when inaccessibility of a replica causes the current transaction to be aborted. Other than increasing the average number of operation records in the log object, the write-all requirement of a read/write object does not contribute to any degradation of the atomic object's availability from the user's point of view.

A different quorum is assigned for each operation of the log object. In addition, a different quorum is assigned for different combinations of arguments for the insert, filter_match, and filter_exists operations. In particular, it is possible that several network partitions may perform insert operations independently, with none of the partitions having processed all the insertions. Consequently, the insertions in different partitions should be "merged" during a view update transaction. In order to simplify the "merge" operation, we set $q(delete_first()) = V_0$, where $q(apply(args))$ denotes the quorum assigned for an operation $op$ invoked with the arguments $args$, and $V_0$ is the total number of votes. Consequently, a deletion must be processed by all replicas but insertions may be performed by multiple network partitions independently. A merge can be implemented by writing the union of all the operation records in all the log replicas as the new log value in each replica in the new view.

The quorums assigned for the insert and filter_match operations satisfy the following relation:

$$V_0 : q(insert(o)) + q(filter_match(t, potential, ...)) > V_0$$

where $t$ is a template that matches $o$. The retry and wait_delete_first operations are assigned a quorum of $0^*$, which means that they can be executed if at least one replica is available. Their semantics provides at most a probabilistic guarantee. The quorum $q(filter_match(t, definite, ...))$ for any $t$ is also set to $0^*$, since returning any subset of the matching records is acceptable according to the semantics. The quorum for a filter_exists operation is set to be equal to the corresponding filter_match operation, since the former can be treated as a shorthand for the latter and they are ignored in the rest of this paper.

We omit the proof that the quorums provide one-copy equivalence. The readers are referred to [Ng92].

5.3 Availability trade-offs

The quorum assignment described in section 5.2 is sufficient to guarantee correctness. However, choosing an appropriate assignment for an application can be simplified by observing the following trade-offs and constraints in a typical atomic type implementation. Due to backward and forward validation, each atomic type operation has to be able to observe other operations whose effects may invalidate its result, or whose results may be invalidated its own effects. This notion of operation conflicts is formally defined using a serial dependency relation in [Her90]. Informally, $(o, o')$ is in a serial dependency relation $serial$ if the existence of $o'$ before $o$ in a sequence may invalidate $o$. If $(o, o') \notin serial$, whether $o'$ exists before $o$ does not affect the validity of $o$. For example, the tuples $((deposit(\ldots)/ok()), (deposit(\ldots)/ok())$, and $(deposit(\ldots)/ok()), (deposit(\ldots)/ok())$ form a serial dependency relation for the bank account type.

In our atomic object implementations, an atomic type operation observes other (conflicting) operations through the use of the filter_match operation of the log object. If both backward and forward validation
succeed, a record for an operation o is inserted into the log object. Consequently, a quorum to observe conflicting operations using the filter-match operation would in general have to be acquired before an operation o is successfully processed (and inserted into the log object). In other words,

\[ q(\text{insert}(o)) \geq q(\text{filter-match}(t', \ldots)) \]

(2)

where \( t' \) is a template that matches an operation \( o' \) and either \((o, o')\) or \((o', o)\) is in serial. Inequalities 1 and 2 imply that in general

\[ q(\text{insert}(o)) + q(\text{insert}(o')) > V \]

(3)

where \((o, o')\) is a pair of conflicting operations in serial. This in turn implies that there is an availability trade-off between a pair of conflicting operations. Exceptions are described in the next section.

5.4 An Example

Using the serial dependency relation

\[
\{((\text{deposit}(\ldots)/\text{no})), (\text{deposit}(\ldots)/\text{ok})),
((\text{debit}(\ldots)/\text{ok})), (\text{debit}(\ldots)/\text{ok}))\}
\]

for the bank account type, the following constraints are imposed in addition to the ones described in section 5.2:

\[ q(\text{insert}(\text{debit}(\ldots)/\text{no}))) + q(\text{insert}(\text{deposit}(\ldots)/\text{ok}))) > V \]
\[ q(\text{insert}(\text{debit}(\ldots)/\text{ok}))) + q(\text{insert}(\text{debit}(\ldots)/\text{ok}))) > V \]

Suppose the log object in the example in section 4 has three replicas with one vote each, the three possible quorum assignments are shown in figure 1. In the first assignment, a successful debit requires two replicas to complete and a deposit requires only one replica to complete in most cases. The exception occurs when a \((\text{debit}/\text{no})\) operation that may be serialized after the deposit exists in the log. In that case, the operation \(\text{filter-match}(\text{deposit}/\text{ok}, \text{potential}, \ldots)\) has to be executed. Executing this particular operation requires all three replicas. Fortunately, \((\text{debit}/\text{no})\) operations are quite rare in a typical bank account, and even if the account is low in funds, a \((\text{debit}/\text{no})\) operation can only be inserted into the log before any network partition.

6 Related work

A large number of replica control algorithms have been proposed to manage replicated data objects. Most of these algorithms model a system as a collection of read/write objects. The read/write model limits concurrency as well as availability, since at most one transaction executing in at most one distinguished network partition is allowed to update an object at any time. Representatives of these algorithms include

![Figure 1: Quorum Assignments for Account Type](image-url)

\begin{align*}
q(\text{insert}(\text{deposit}(\ldots)/\text{ok}))) & \geq 2 \text{ or } 3 \\
q(\text{filter-match}(\text{deposit}/\text{ok}, \text{potential}, \ldots)) & \geq 2 \text{ or } 3 \\
q(\text{insert}(\text{debit}(\ldots)/\text{no}))) & \geq 2 \text{ or } 3 \\
q(\text{filter-match}(\text{debit}/\text{no}, \text{potential}, \ldots)) & \geq 2 \text{ or } 3 \\
q(\text{insert}(\text{debit}(\ldots)/\text{ok}))) & \geq 2 \text{ or } 3 \\
q(\text{filter-match}(\text{debit}/\text{ok}, \text{potential}, \ldots)) & \geq 2 \text{ or } 3
\end{align*}
the available copies algorithm [BG84], the true-copy token algorithm [Min82], and the quorum consensus method [Gif79]. A large number of extensions to quorum-consensus have also been proposed [DB85, AT86, Her87b, JM87, Par86].

Using the semantics of abstract data types to improve availability is discussed in [BDS87, Her87a, Her90]. Three algorithms: consensus locking [Her87a], consensus scheduling [Her87a], and optimistic state-based backward validation [Her90], are considered below. These algorithms are similar to ours in their use of a replicated log of atomic type operations.

6.1 Consensus locking

Consensus locking [Her87a] is a lock-based algorithm in which each atomic type operation acquires a lock in a quorum of repositories before it is processed. Each operation has its own lock type and quorum, and whether two operation locks conflict and two operation quorums overlap are determined by a lock conflict relation and a quorum intersection relation respectively. The intersection of the two relations has to be a serial dependency relation.

Consensus locking and the algorithm described in this paper provide similar degrees of availability. Using the bank account example, consensus locking provides a higher degree of availability during a deposit operation if an unsuccessful debit operation may be serialized after the deposit. Our approach requires other deposit operation records to be collected to determine whether the debit operation may be invalidated, whereas consensus locking disallows concurrent deposit and unsuccessful debit operations. Consequently, consensus scheduling may provide higher availability at the expense of lower concurrency. If not all deposit operation records can be collected, the availability of our approach can be improved (to match consensus locking) by handling this exception with a retry statement that retries the deposit either when more replicas are available or when no unsuccessful debit operations may be serialized after the deposit. If the order of commitment is used as the serialization order (which is the case in consensus locking), the second condition is equivalent to waiting until concurrent unsuccessful debit operations are committed or aborted, which is required in consensus locking anyway.

Consensus locking, when compared to state-based approaches such as consensus scheduling or the one described in this paper, has the advantage that it is relatively easy to program once the lock conflict and quorum intersection relations have been determined. The repositories use a simple table lookup to determine whether an operation should be blocked, after which no further explicit synchronization is needed. A lock-based algorithm, however, has several disadvantages. First, it does not take the state of an object into consideration. For example, two debits are not allowed to proceed concurrently, even when there are sufficient funds to cover both of them. Without considering how much is being debited relative to the current balance, at most one debit can be allowed to proceed. Second, since locking algorithms imply that transactions are serialized in the order they release and acquire a conflicting lock (i.e., the order of commitment), it is not clear how lock-based algorithms can be modified to allow other serialization orders.

6.2 Consensus scheduling

Consensus scheduling [Her87a], like the approach described in this paper, is state-based. A quorum of log object replicas are collected and merged, and any changes to the log are written to a quorum of the replicas also. Synchronization is performed by examining the state of the merged log replicas. Consensus scheduling overcomes the concurrency problem of consensus locking mentioned above. The difference between consensus scheduling and our approach lies in quorum assignment. In consensus scheduling, quorums are assigned to the atomic type operations, whereas quorums are assigned to log operations in our approach, leading to a difference in availability.

In [Her87a], Herlihy shows that the quorum intersection relation used by consensus locking is always a subset of the one used by consensus scheduling. Consequently, consensus scheduling offers more concurrency at the expense of less availability. In the bank account example, the serial dependency relation

\[
\{((\text{deposit}(. . .)/\text{ok}(), (\text{deposit}(. . .)/\text{no}(), \\
(\text{debit}(. . .)/\text{ok}()), (\text{debit}(. . .)/\text{no}()))
\]

is not a correct quorum intersection relation for consensus scheduling. An extra operation pair \(((\text{deposit}(. . .)/\text{ok}()), (\text{deposit}(. . .)/\text{ok}())\) is needed, as a deposit operation may have to collect other deposit operation records to determine whether together they may invalidate an unsuccessful debit operation. In our approach, an operation can wait for either more replicas to be available (if there are too few currently) or the conflicting transactions to be terminated, whichever occurs first. In addition, the waiting is necessary only when a conflicting operation exists. In the bank account example, an unsuccessful debit is unlikely to exist in a typical account, meaning that deposits are unlikely to be required to have intersecting quorums. In that sense, our “quorum intersection relation” for the operations of an atomic type is dynamic, because it depends on which log operations are executed at run time. Adding this dynamicity to consensus scheduling is possible but would require the application to specify which quorums to use under which conditions, instead of specifying a single quorum for each atomic type operation.

6.3 Optimistic state-based approaches

Optimistic state-based approaches [Her90] are similar to pessimistic approaches except that validation is performed immediately before commitment. Failure to validate causes a transaction to be restarted. Either backward or forward validation are performed, instead of both. Backward validation has the advantage that the validation overhead is independent of the number of concurrent transactions. Optimistic state-based backward validation overcomes the availability and concurrency trade-off between consensus locking and consensus scheduling. A serial dependency relation can be used as the quorum intersection relation.

174
7 Conclusion

This paper describes an approach to implement highly available and concurrent atomic objects. The state of an atomic object is represented using non-atomic objects. Type-specific synchronization is used to guarantee serialization in a transaction order. Our approach offers high availability by replicating the non-atomic objects. Since the log object hides the serialization order being used by the atomic objects, our replica control algorithm can also be used with different serialization orders without change.

The replica control algorithm we propose provides higher availability than algorithms that use a read/write object model. Update operations may be processed in more than one network partition as long as the object's history remains atomic. The overhead of reading multiple object replicas is eliminated by using the virtual partition algorithm. In our approach, quorums are assigned to log operations instead of atomic object operations. An object can determine accurately at run time which log operations have to be executed, and it can also decide at run time between waiting for a conflicting transaction to be terminated or for network connectivity to improve.

References


