

Exploiting Commutativity for Distributed Object Caching

John Eberhard and Anand Tripathi
Department of Computer Science
University of Minnesota
Minneapolis, MN 55431
{eberhard, tripathi}@cs.umn.edu

Abstract

In this paper, commutativity relationships are derived from the semantics of an object and are used to improve the performance of a distributed object system. First, we present a methodology to create a method commutativity specification from an object's semantic specification. Unlike previous approaches, which compare pairs of operations, our approach specifies the conditions under which the methods in a group of methods will commute. Second, since some applications tolerate weaker consistency, weakened semantics may be expressed in the semantic specification and enable greater commutativity. Third, we introduce method licenses, which permit a client to execute a method on a cached copy of an object with the guarantee that the executed method will commute with all other concurrent method invocations by other clients. Finally, we illustrate the performance benefits of this approach by using these concepts in a working prototype that uses Java RMI.¹

1 Introduction

In a wide area environment where high latencies are involved in communication, the caching of objects is necessary to assure reasonable performance of applications utilizing distributed objects. While it is possible to design an application such that distributed accesses are reduced, a better approach is to transparently add caching to the distributed object infrastructure.

Several researchers have investigated the use of semantics of object methods in the context of transactional object systems. While these approaches work well in a LAN environment, they do not perform well in a WAN environment. One problem is the la-

tency of acquiring a lock. Furthermore, there are non-transactional distributed object applications that also require caching. Existing transactional approaches [1] rely on the output of an object method invocation to determine concurrency control. This assumes the ability to abort or rollback the method invocations of a transaction.

In this paper we present an approach that exploits object semantics in a non-transactional distributed system. The main contributions of our work are the following.

- We present a methodology to extract commutativity from a semantic specification of an object and to create a *method commutativity specification* (MCS) that describes the conditions when sets (instead of pairs) of method invocations will commute.
- We extend this methodology such that weaker consistency requirements may be added to an object's semantic specification to increase the commutativity present in the resultant MCS.
- We introduce *method licenses*, derived from the MCS, that assure that clients access cached objects without violating the objects consistency.
- We use a working prototype to illustrate the performance benefits that can be obtained through the use of an MCS and method licenses.

This paper presents these contributions in the following manner. In Section 2, we review previous approaches that use semantics to improve concurrency. In Section 3, we provide an overview of the computation environment, as well as an introduction to method licenses. In Section 4, we provide further details about how commutativity information is extracted from the semantic specification and how the semantics specification can be weakened to provide greater concurrency. We also

¹This work was supported by NSF grant ITR 0082215.

describe how semantic information is used at runtime in the form of method license. In Section 5, we provide some experimental results of using these concepts in a distributed object application. We conclude in Section 6.

2 Previous Approaches

Our work builds upon prior work on exploiting object semantics in transactional environments. The seminal work by Weihl [12, 13, 14] identified commutativity relationships between pairs of “operations”. An operation was defined to include the parameters passed to the “operation” as well as the output of the “operation”. Weihl classified commutativity as two forms: forward and backward commutativity. Two operations “forward commute” if for all states of the object, if both operations are defined for a state, then applying the operations in either order results in the same state. Also implicit in this definition is that the output of the operation remains the same. Two methods “backward commute” if the operations can successfully execute in one order, then the method would have successfully executed in the opposite order and the final state of the object would remain the same. In both case, the definitions assume that the results of the operation are available and it is possible to abort or rollback the execution of the operation.

Other distributed object systems have attempted to exploit object semantics, but often do so by requiring “heavyweight” operations that include provisions for conflict resolution. For example, Bayou [11] “writes” contain merge procedures that are executed when conflicts are detected. The IceCube approach [7] involves a distributed object system where updates to a object at a client are logged. The updates are then sent to the server, which then has the responsibility of finding an acceptable order to all client updates.

Our work builds upon the “conits” of Yu and Vahdat [15]. A “conit” is a means of specifying the weakened consistency requirements of a cached object. Our work uses the weakening principles specified by conits in a systematic way to specify appropriate guarantees at the client.

In a general sense, our work applies the principle of semantic callbacks [9] to a working environment. In our work, the method license contains a predicate that must be true at each client. The server is aware of the method licenses and uses callbacks to adjust those licenses (and corresponding predicates) as needed to assure consistency. Not only does our work incorporate semantic callbacks, but it also provided a methodology that can be used to obtain the necessary predicates

from the semantic description of an object.

3 Overview

The essence of our approach is the use of method licenses to govern the execution of methods on clients in a distributed client server environment. The method licenses are created from information based on the semantic description of an object. In this section, we review the computational environment and describe the method licenses. In the next section we will describe how the commutativity information contained in the method licenses is obtained.

3.1 Computational Environment

The computational environment that we consider for our work is the distributed client-server environment. In this environment, distributed objects are provided by a server and accessed by clients using remote method invocation (RMI). Using this model, communication with the server is necessary for each method invocation.

To reduce the amount of communication in a client server system, we have implemented an infrastructure that transparently caches RMI objects at the client [4]. In our original approach, access to cached copies was controlled by a simple “multiple-readers/single-writer” protocol which performed poorly. In our new approach as presented here, methods are invoked on the local copy of the object and the method execution is logged so that it will eventually be executed at the server. However, when methods are invoked on a cached copied of an object, the global consistency of the object must now be managed and maintained. This paper explores the use of semantics to assure the consistency of cached objects. The primary mechanism to assure consistency is the method license, described below, which is derived from the semantic description of an object.

3.2 Method Licenses

The method license is used to control access to cached copies of an object. We first define the method license and then present two scenarios to illustrate its use. In the first scenario, the method is allowed to execute at the client. In the second scenario, the server must be contacted before the method is allowed to execute.

A method license is associated with each cached object. A method license is the permission, granted by the server, for a client to invoke methods on a cached object. To accomplish this, a method license contains

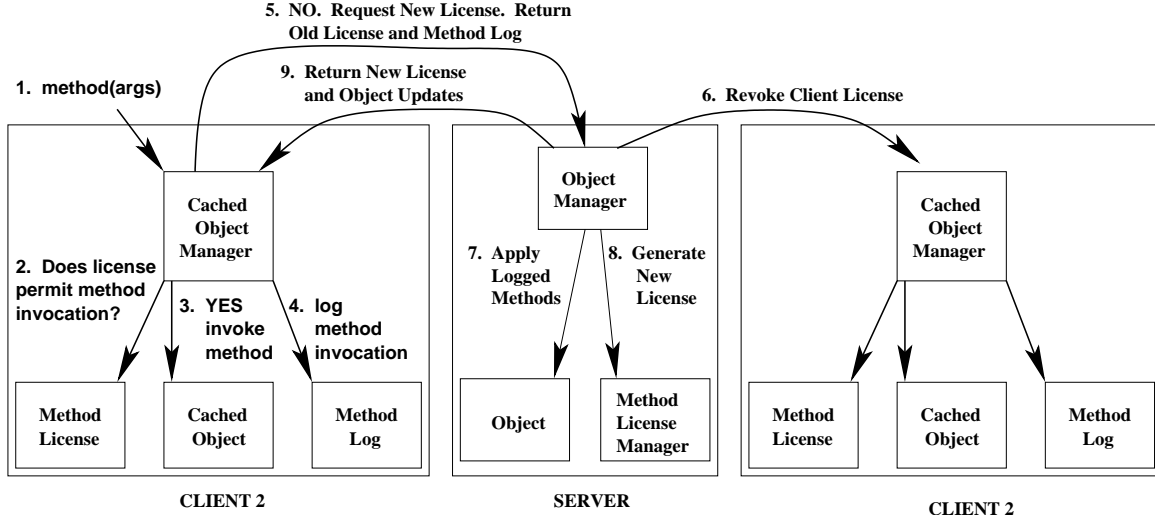


Figure 1. Method License Usage

a predicate describes the methods that a client is permitted to executed. The server manages the method licenses of all the clients to guarantee that all invoked methods will commute with each other. In some respects, method licenses may be considered a form of semantic guarantee about the state of the concurrently cached object. In our approach, each client is granted a method license that describes the methods that the client may execute (without contacting the server) with the guarantee that the object’s consistency will not be compromised.

When an object is cached, several other objects are also placed on the client. These objects are shown for Client 1 and Client 2 in Figure 1. These objects manage the invocation of methods on the cached object and assure the consistency of the object. A typical method invocation causes four steps to occur. In step 1, the cached object manager receives a method invocation. The cached object manager is responsible for assuring the consistency of the object, in particular it assures that any method invoked on the cached object will not violate any global consistency constraints. To do this, in step 2, the cached object manager checks the method license to see if the method is permitted to execute. In this case, the method license permits the method invocation. Consequently, in step 3, the method is invoked on the cached object. After the method is invoked, in step 4, the invocation of the method is recorded in the method log. This method log will eventually be sent to the server, in order to invoke the logged methods on the server copy of the object.

The second scenario, also shown in Figure 1, occurs

when a method license does not permit the method to be executed on the cached object. In step 1, the method invocation is given to the cached object manager. In step 2, the cached object manager asks the method license if the method may be executed. The method license responds that the method cannot be executed. The cached object manager then requests, in step 5, a new license from the object manager on the server and returns the old license and the method log. In this scenario, the object manager cannot grant a license to the client without revoking the method licenses held by other clients. Consequently, in step 6, the method license is then revoked from client 2. Client 2 returns its method license and method log. In Step 7, the object manager applies all the logged methods to the server’s copy of the object. It then obtains a new license for the client from the license manager in step 8. Finally, in step 9, the object manager returns the new license and object updates to the client. The object updates may either be a new version of the cached object or it may be a set of methods invocation to update the currently cached object.

4 Commutativity Relationships

Our work exploits the semantics of an object to increase concurrency in a distributed object system. The semantics of an object are described in a Method Specification Table (MST), where each row can be viewed as a Hoare Logic expression. Then using the MST, we use a definition of commutativity as well as commutativity rules to create a Method Commutativity Specifi-

cation (MCS). Recognizing that weakening consistency requirements may provide greater concurrency, a MST may be modified to provide weaker consistency. The MCS is used at runtime to create method licenses that govern the behavior of clients accessing a cached object.

4.1 Method Specification Table

Our approach begins with a Method Specification Table (MST), which is a semantic specification of an object’s methods. An MST consists of rows which describe the behavior of a method under specific preconditions. Each row specifies a method and its associated precondition and postconditions. The precondition (P) is a boolean predicate of the form $f(T, P_1, \dots, P_n)$, where T represents the state of the object and P_1, \dots, P_n represent the parameters of the method. One postcondition, the *state postcondition* (S), is a predicate describing the effect of the method upon the state of the object. The predicate is either specified as TRUE, if the object is not changed by the method; or it has the form $T' = g(T, P_1, \dots, P_n)$, meaning the new state T' of the object is a function that represents the new state of the object based on the current state of the object and the parameters of the method. The other postcondition, the *output postcondition* (R), is a predicate describing the value returned by the method. This predicate is either specified as TRUE, meaning that the method does not have a return value; or it has the form $V = h(T, P_1, \dots, P_n)$, where V is the output value. Because a method may have different behavior under different conditions, some methods may be described by more than one row.

Rows of the MST may be viewed as a restricted form of Hoare Logic[6] expressions. A Hoare Logic expression is a triple, $P \{Q\} R$, which describes the behavior of a program Q in terms of a precondition P and a postcondition R. A row of the MST may be considered a Hoare Logic expression where the precondition is P, the method is Q, and R is the combination of the state postcondition and the output postcondition. For the remainder of the paper, we will use Hoare Logic expressions of the form, $P_i \{M_i\} S_i \wedge R_i$, where P_i is the precondition, M_i is the method, S_i is the state postcondition, R_i is the output postcondition, and the subscript i indicates the row of the MST.

Examples of MSTs for a *bank account* object and for a *set* object are shown in Figures 2 and 3. The bank account has four methods whose behavior is described by five rows of the MST. In each row, the state of the object is represented by an integer, *bal*. The set object has also four methods that are described by five rows

	P	M	S	R
1	TRUE	balance()	TRUE	V=bal
2	$x > \text{bal}$	withdraw(x)	TRUE	V=FAIL
3	$y \leq \text{bal}$	withdraw(y)	$\text{bal}' = \text{bal} - y$	V=OK
4	TRUE	deposit(z)	$\text{bal}' = \text{bal} + z$	V=OK
5	TRUE	withdrawAll()	$\text{bal}' = 0$	V=bal

Figure 2. MST for Bank Account

	P	M	S	R
1	TRUE	Insert(x)	$S' = S \circ x$	V=OK
2	TRUE	Delete(y)	$S' = S - y$	V=OK
3	$z \in S$	Member(z)	TRUE	V=TRUE
4	$z \notin S$	Member(z)	TRUE	V=FALSE
5	TRUE	Clear()	$S' = \emptyset$	OK

Figure 3. MST for Set

of the MST. In each row, the state of the object is represented by a set, S.

4.2 Sequential Composition of Methods

Given that an MST describes method invocations, we want to use information from the method specification table to obtain the preconditions and postcondition for a method sequence. Consider the following Hoare expressions containing the information from two rows of the MST.

$$P_i \{M_i\} S_i \wedge R_i \text{ and } P_j \{M_j\} S_j \wedge R_j$$

We wish to create an expression of the following form, which represents the precondition and postcondition of executing the two methods as a sequence.

$$P_i \wedge P_{ij} \{M_i; M_j\} S_{ij} \wedge R_i \wedge R_{ij}$$

In this expression, $P_i \wedge P_{ij}$ represents the precondition for the method sequence, where P_{ij} represents the condition that must hold before M_i executes such that P_j is true after M_i executes. S_{ij} represents the state postcondition of the sequence. R_i and R_{ij} represent the output postconditions, one for each method. To obtain P_{ij} , S_{ij} , and R_{ij} , we substitute the state changes specified by S_i into P_j , S_j , and R_j , respectively.

First, consider the case where S_i is the TRUE constant. In this case, M_i does not change the object state. Consequently, $P_{ij} = P_i$, $S_{ij} = S_j$, and $R_{ij} = R_j$.

If S_i is not the TRUE constant, then it has the form $T' = g_i(T, \dots)$. Since P_j has the form $f_j(T, \dots)$,

Method Group: $withdraw(i)$ where $i < bal$ and $\sum i \leq 100$
$withdraw(1), withdraw(1), withdraw(3)$
$withdraw(50), withdraw(50)$
$withdraw(100)$

Figure 4. Sets of Methods in a Method Group

we can substitute T' from S_i into the T of P_j to create $f_j(g_i(T, \dots), \dots)$, which we denote as $S_i \triangleright P_j$, the substitution of S_i into P_j . Consequently, $P_{ij} = S_i \triangleright P_j$. Similarly, we can substitute S_i into S_j to create $S_i \triangleright S_j$, which is of the form $T' = g_j(g_i(T, \dots), \dots)$. We also substitute S_i into R_j to create $S_i \triangleright R_j$, which is of the form $V_j = h_j(g_i(T, \dots), \dots)$.

4.3 Commutativity

Once the method semantics are identified, we wish to identify the conditions under which groups of methods commute with each other. To do this, we first must have a definition of commutativity. While previous definitions currently exist [2, 8, 12], we present a definition based on our method specification described above. Then using this definition, we illustrate through the use of Hoare Logic how commutativity relationships can be determined. Since such a detailed analysis is not always necessary, we define rules to easily determine if groups of methods commute.

4.3.1 Commutativity Definition

For our commutativity definition, rather than considering pairs of methods that commute, commutativity is defined for a set of method invocations such that all method invocations in the set commute. Many sets of method invocations are identified by a *method group*. Each set in a method groups consists of method invocations that satisfy a group precondition, P , where the methods are taken from rows of the MST. In other words, a method group can be viewed as a generator of sets of method invocations from rows of the MST that satisfy the method group precondition. The group precondition usually defines a relationship between the parameters of the methods in the method group and the state of the object. For example, consider the method group consisting of methods from row 3 of Figure 2, that satisfy the precondition $\sum i \leq 100$. Some possible sets of methods in this method group are shown in Figure 4.

Given this definition of a method group, we now wish to determine for each possible set identified by a

method group, that, for each set, the method invocations commute with each other. If they do, then the method group is a **commutative method group**. A method group is a **commutative method group** if for every state of the object and every set of methods, M , with elements m_1, m_2, \dots, m_n , that satisfy the group precondition P , and for all possible orderings of m_1, m_2, \dots, m_n , the following conditions are true.

- CC1: Commutativity Condition 1
The precondition for each method, m_1, m_2, \dots, m_n , is always true before the method's execution, regardless of the order in which it is executed.
- CC2: Commutativity Condition 2
The final state of the object is the same. That is, the sequential composition of the method postconditions after the methods have executed is identical for all possible orderings of m_1, m_2, \dots, m_n .
- CC3: Commutativity Condition 3
Each method's return value remains the same, regardless of the order in which it is executed.

These three commutativity conditions form the basis of our definition of a commutative method group.

4.3.2 Commutativity and Hoare Logic

Suppose we have methods M_i and M_j which are described by rows of the MST. The preconditions and postconditions for the two possible method sequences are the following:

$$P_i \wedge P_{ij} \{M_i; M_j\} S_{ij} \wedge R_i \wedge R_{ij}$$

$$P_j \wedge P_{ji} \{M_j; M_i\} S_{ji} \wedge R_j \wedge R_{ji}$$

The two methods commute if they satisfy the three commutativity conditions. First, because of the definition of P_{ij} and P_{ji} , CC1 will always be satisfied when $P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$ is true. Second, to satisfy CC2, the final state of the object for the two sequence must be the same. If S_{ij} is of the form, $T' = g_j(g_i(T, \dots), \dots)$, and S is of the form, $T' = g_i(g_j(T, \dots), \dots)$, then the states are the same when $g_j(g_i(T, \dots), \dots) = g_i(g_j(T, \dots), \dots)$. Consequently, $S_{ij} \wedge S_{ji}$ must always be true when $P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$. Third, to satisfy CC3, the return values must be equal. If R_i is of the form, $V_i = h_i(T, \dots)$, and R_{ji} is of the form, $V_i = h_i(g_j(T, \dots))$, then the return values are equal when $h_i(T, \dots) = h_i(g_j(T, \dots))$. In other terms, $R_i \wedge R_{ji}$ is true. Similarly, $R_j \wedge R_{ij}$ must also be true.

If any of the three conditions, $R_i \wedge R_{ji}$ or $R_j \wedge R_{ij}$ or $S_{ij} \wedge S_{ji}$, is not always true when the precondition

$(P_i \wedge P_{ij} \wedge P_j \wedge P_{ji})$ is true, then the methods do not commute. If this is the case, it may be necessary to add an additional precondition to assure that $R_i \wedge R_{ji}$ and $R_j \wedge R_{ij}$ and $S_{ij} \wedge S_{ji}$ are always true.

This analysis technique is easily extended to the consideration of larger groups of method invocations. For example, a group of three methods would result in six possible orders, instead of two. The preconditions and postcondition of the six possible method sequences could be analyzed as shown above.

4.3.3 Commutativity Analysis

To determine commutativity of a pair of methods, the evaluation of the preconditions and postcondition as described in the previous section must be done. During analysis, it may be evident that the additional invocations of the same methods may also commute. This section illustrates the analysis of the *withdraw()* method as described by row 2 of Figure 2.

We begin by labeling each method with a subscript, and choosing an appropriate variable for the parameter. We then evaluate the preconditions and postconditions for our analysis. These values are as follows.

$$P_i : x < bal; S_i : bal' = bal - x; R_i : V = OK$$

$$P_j : y < bal; S_j : bal' = bal - y; R_j : V = OK$$

$$P_{ij} : y < bal - x; S_{ij} : bal' = (bal - x) - y$$

$$P_{ji} : x < bal - y; S_{ji} : bal' = (bal - y) - x$$

First, consider the precondition that must be true. Due to the commutative properties of arithmetic, $P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$ can be simplified to $x + y < bal$. Similarly, both S_{ij} and S_{ji} simplify to $bal' = bal - (x + y)$, which are always true. Since R_i and R_j do not depend on the state of the object, both $R_i \wedge R_{ji}$ and $R_j \wedge R_{ij}$ are always true.

This analysis can be extended to more than two methods. By induction, one can show that invocations of *withdraw* method will always commute as long as the sum of the parameters is less than *bal*. Similarly, for all orderings, the final postcondition will be $bal' = bal - \sum x$.

As mentioned earlier, it may be necessary to add additional preconditions to assure that the methods commute. Consider rows 1 and 2 of the set object. When we execute the methods *insert(x)* and *delete(y)* in different orders, we obtain the following post conditions: $S' = (S \circ x) - y$ AND $S' = (S - x) \circ y$. These postconditions contradict when $x = y$. To compensate, we must add the precondition that $x \neq y$. By induction, these methods commute as long as the following condition holds: $\forall x \forall y, x \neq y$.

4.3.4 Efficient Rules for Commutativity Testing

The full commutativity analysis of the previous section is not always necessary when determining commutativity. Using the commutativity definition, we have identified three simple rules to determine when methods (with their associated preconditions) do not commute. These rules must be true for all states of the object and for all parameters of the methods that satisfy the preconditions of the methods.

- RETURN OVERWRITE RULE

Methods do not commute if one method always changes the return value of another method. For example, the *deposit* and *balance* methods of bank account do not commute since deposit always changes the return value of the balance method.

In terms of Hoare logic, for all combinations of M_i and M_j , the methods do not commute if any one of the following two conditions is always false: $R_i \wedge R_{ji}$ or $R_j \wedge R_{ij}$.

- STATE OVERWRITE RULE

Methods do not commute if the state postconditions always conflict, i.e. executing the methods in a different order results in a different final state of the object. An example of methods that do not commute for this reason are the *Insert* and *Clear* methods of the *Set* object.

In terms of Hoare logic, for all combinations of M_i and M_j with state postconditions S_i and S_j , $S_{ji} \wedge S_{ij}$ is always false.

- PRECONDITION MODIFICATION RULE

Methods do not commute if the postcondition of one method always changes the state of the object such that the precondition of another method is no longer true. An example of methods that do not commute for this reason are the *withdraw* and *withdrawAll* methods of the Bank Account object, as described by rows 3 and 5 of Figure 2.

In terms of Hoare logic, for all combinations of M_i and M_j , either $P_i \wedge P_{ij}$ is always false or $P_j \wedge P_{ji}$ is always false.

We have identified the following simple rule to determine if methods commute.

- NON UPDATING RULE

Methods which do not change the state will always commute with each other when their preconditions are satisfied. For example, a *balance()* method will

Row	From Rows	Methods	Conditions
1	1	{balance()*}	TRUE
2	2	{withdraw(x)*}	$\forall x \ x > \text{bal}$
3	3	{withdraw(y)*}	$\forall y \ \sum y < \text{bal}$
4	4	{deposit(z)*}	TRUE
5	1,2	{balance(), withdraw(x)*}	$\forall x \ x > \text{bal}$
6	1,3	{balance(), withdraw(y)*}	FALSE
7	1,4	{balance(), deposit(y)*}	FALSE
8	2,3	{withdraw(x)*}	$\forall x \ (x > \text{bal} \text{ OR when } x \leq \text{bal}, \sum x \leq \text{bal})$
9	2,4	{withdraw(x), deposit(z)*}	$\forall x \forall z \ (x > \text{bal} + \sum z)$
10	3,4	{withdraw(y), deposit(z)*}	$\sum y \leq \text{bal}$
11	2,3,4	{withdraw(x), deposit(z)*}	$\forall x \forall z \ (x > \text{bal} + \sum z) \text{ OR (when } x \leq \text{bal}, \sum x \leq \text{bal})$

Figure 5. MCS for Bank Account

always commute with a *withdraw()* method when the parameter of the *withdraw()* method is greater than the balance.

In terms of Hoare logic, for all combinations of M_i and M_j , S_i and S_j are the TRUE constant and the methods commute when $P_i \wedge P_j$ is true.

4.4 Method Commutativity Specification

Unlike previous approaches which create a matrix to specify which pairs of methods commute, we create a method commutativity specification (MCS) which identifies groups of methods that will always commute as long as an associated condition is satisfied. Each row of the MCS identifies the group of methods that commute, the rows of the MST which the methods come from, and the condition under which the methods commute. We use the following notation to indicate the grouping of a number of method invocations of methods m_i and m_j : $\{m_i, m_j\}^*$. Figure 5 shows the MCS for a bank account. The methodology for creating the MCS is described below.

The MCS is built incrementally. First, method groups consisting of methods from one row of the MST are considered. For the bank account object, these are

shown in rows 1 through 4 in Figure 5. Then method groups consisting of methods from two rows of the MST are considered, as shown by rows 5 through 10 of Figure 5. The process is continued until all possible grouping have been considered.

If n is the number of rows in the MST, then at most $2^n - 1$ combination of method must be considered to generate a complete MCS. However, if it is determined that a group of methods do not commute, other method groups containing that method group do not need to be considered any further. For example, the method group consisting of rows 1,2,3 of the MST of the bank account object is not considered because rows 1,2 of the MST do not commute, as shown by row 6 of Figure 5.

When generating each row of the MCS, the following steps are taken is followed.

First, determine if the rows satisfy any of the NON COMMUTING rules. For example, rows 1 and 3 in Figure 2 do not commute as shown by row 6 of Figure 5.

Second, determine if the rows satisfy the NON UPDATING rule. If so, then the methods will commute for the specified preconditions. For example, the *balance()* method in row number 1 of Figure 2 will always commute with itself. The resultant rule is shown as row 1 in Figure 5.

If at this point, it is still unknown whether the methods commute, then the analysis process described earlier must be used. For example, the entry in the MCS corresponding to the analysis of the withdraw method is shown in row 3 of Figure 5.

Using these techniques, an MCS can be created for the set object. The MCS for a set object is shown in Figure 6.

4.5 Weakening Consistency

While the use of the MCS may expose commutativity in a distributed environment, there may be cases where we wish to permit more concurrency by weakening the consistency requirements. In particular, the consistency conditions given in Section 4.3.1 may be weakened. One manner to weaken consistency is to weaken the consistency of the value returned by a method execution. Another manner to increase concurrency is to weaken the requirements concerning the final state of the object.

4.5.1 Weakening a Method's Return Value

The first aspect we can weaken is the visibility of the current state of the object. To accomplish this, the MST is changed so that the return value is based on

ROW	ROWS	Methods	Conditions
1	1	{insert(i)}*	TRUE
2	2	{delete(i)}*	TRUE
3	3	{member(i)}*	$\forall i, i \in S$
4	4	{member(i)}*	$\forall i, i \notin S$
5	1,2	{insert(i), delete(j)}*	$\forall i, j, i \neq j$
6	1,3	{insert(i), member(j)}*	$\forall j, j \in S$
7	1,4	{insert(i), member(j)}*	$\forall j, j \notin S$ AND $\forall i, j, i \neq j$
8	2,3	{delete(i), member(j)}*	$\forall j, j \in S$ AND $\forall i, j, i \neq j$
9	2,4	{delete(i), member(j)}*	$\forall j, j \notin S$
10	3,4	{member(i)}*	TRUE
11	1,2,3	{insert(i), delete(j), member(k)}*	$\forall i, j, i \neq j$ AND $\forall k, k \in S$ AND $\forall j, k, j \neq k$
12	1,2,4	{insert(i), delete(j), member(k)}*	$\forall i, j, i \neq j$ AND $\forall k, k \in S$ AND $\forall i, k, i \neq k$
13	1,3,4	{insert(i), member(j)}*	$\forall i, j, i \neq j$
14	2,3,4	{delete(i), member(j)}*	$\forall i, j, i \neq j$
15	1,2,3,4	{insert(i), delete(j), member(k)}*	$\forall i, j, k, i \neq j$ AND $j \neq k$ AND $i \neq k$

Figure 6. MCS for Set Object

the state of the cached object (instead of the global state). In the MST, we annotate cached state using a C in the subscript. For example, the locally cached balance of a bank account is designated bal_C . Because the returned value is now the locally cached value, the return value remains the same, no matter what order the method is executed in. However, returning a value based on the cached object's state is not very valuable unless there are guarantees about the consistency of the current state of the object. To specify the consistency conditions of the cached object, we can use conditions similar to the conditions used by Yu and Vahdat's *conit* approach [15]. For example, we can use a $TIME(fromRow, toRow, time)$ condition which indicates that invocation of a method in the *fromRow* of the MST row may not be visible to the method in the *toRow* of the MST until *time* seconds later. These conditions are added to the precondition of each method and are added to the MCS as a condition.

	M	P	S	R
1	balance()	$TIME(3,1,15) \wedge$ $TIME(4,1,15)$	TRUE	$V = bal_C$

Figure 7. Weakened Balance Row of Bank Account MST

ROW	FROM	Methods	Conditions
12	1,2,3,4	{balance(), withdraw(i), deposit(j)}*	$\sum i \leq bal$ AND $TIME(3,1,15)$ AND $TIME(4,1,15)$

Figure 8. Modified Row of Bank Account MCS

For example, we can use read weakening techniques to provide greater concurrency with respect to the use of the bank account's *balance()* method. Figure 7 shows the weakening change made to row 1 of the MST for the bank account. In this row, the return value was changed to indicate that the cached value should be returned. Preconditions were also added to enforce the requirement that any effect of a deposit or withdraw method is visible within 15 seconds.

Using this changed definition we are able to generate a new MCS for the bank account. Because more commutativity is now possible, more rows are added to the MCS for the bank account. For example, the MCS now contains a row which includes all the rows from the MST. This new row is shown in Figure 8.

While read weakening uses a *conit-like* approach, the advantage to our approach is that the weakened read consistency is placed in the semantic definition such that its use will only affect the values read, but will not influence the other concurrency conditions, CC1 or CC2.

4.5.2 Weakening Final State of Object

The second condition that can be weakened is CC2, which states the the final state of the object will be the same. While we have not yet implemented this form of weakening in our prototype, we briefly explore how this form of weakening may be accomplished. The most trivial approach to weakening this condition is to ignore differences in the final state and assume that the final state of either order is acceptable. A better approach is to provide mechanisms to assure that the final state is the same, regardless of the order in which the methods are executed. However, when weakening in this manner, it must be remembered that the weaken-

	P	M	S	R
1	TRUE	Insert(x)	$S' = S \circ x$ AFTER(1,2,x=y)	V=OK
2	TRUE	Delete(y)	$S' = S - y$	V=OK
.

Figure 9. Weakened Rows for Set MST

ROW	ROWS	Methods	Conditions
6	1,3	{ insert(x), member(y)}*	AFTER(1,2,x=y)

Figure 10. MCS for Set Object

ing only affects the $S_i \triangleright S_j$ condition, not the $S_i \triangleright P_j$ condition, during the commutativity analysis.

We specify final state weakening by adding additional qualifiers to the postcondition of the MST. These qualifiers are used during the commutativity analysis as well as when the object is used. When used during commutativity analysis, the qualifiers only apply to operations that reflect the final state of the object. The caching infrastructure is responsible for assuring that mechanisms corresponding to the qualifiers are used at runtime.

In the most basic approach, methods are executed concurrently. When the log is returned to the server, the server tracks the potentially concurrent operations that have not yet been returned from the clients. The application of the methods from a client’s log is delayed until it not possible for concurrent operations to be in progress. The client log is then examined, and the qualifiers are used to determine the correct ordering of the methods. The qualifiers are used to enforce an ordering between concurrent methods. This is done by annotating a priority relationship between methods. An example is the AFTER annotation. The AFTER annotation has the form, AFTER(*laterRow*, *earlierRow*, *condition*), where the *laterRow* will be executed after the *earlierRow* when the concurrent method invocations are returned to the server. For example, the MST of a set object can state that a concurrent *insert()* will always follow a concurrent *delete()* when their parameters are the same. Figure 9 shows rows of the modified MST. With this change, the MCS be weakened as shown in Figure 10.

It may also be possible to use more advanced mechanisms. The use of these mechanisms are beyond the scope of this paper. However mechanisms based on operational transformation [10] or data patch [3] could be used. In this approach, the MST would contain a qual-

ifier for the *operational transform* or *data patch* that would cause the appropriate mechanism to be used at runtime.

4.6 Method Licenses and Method License Manager

The MCS is used in a distributed caching system to guide the management of method licenses that are granted to clients. A method license grants a client permission to execute a group of methods on a cached copied of an object with the assurance that the methods will eventually execute on the server’s copy of the object and that the execution of those methods will commute with other methods that are concurrently executing at other clients. The method license may specify conditions based on the current cached state of the object as well as the parameters of the method invocations. These conditions specify whether the client is able to continue executing methods at a client.

A method license manager (MLM) is responsible for granting method licenses to clients. The MLM uses the MCS as the basis for assigning method licenses. It does this by designating a row of the MCS as the active row. The active row indicates which row of the MCS is currently being used to grant method licenses to clients.

Since there are cases where a row of the MCS may not contain a method that the client wishes to execute, the MLM may grant an exclusive method license to a client. An exclusive license indicates that only the current client is able to execute methods on an object.

The MLM receives a request to grant a client a method license in one of two scenarios. The first case is when the object is cached for the first time. The second case is when the client attempts to execute a method on the object, but does not have the appropriate method license. In both cases, the MLM is asked to provide a client with a method license. Besides indicating which client is requesting the method license, the request also indicates which methods, with associated conditions, that the client is likely to execute.

When the MLM receives a method license request, it uses the following steps to grant that request.

1. If any client has an exclusive method license, then that license is revoked using the callback from the server to the client.
2. If the MCS does not have an active row, select a row as the active row.

The MLM uses the client’s information about the method it will execute in order to select the active

row of the MCS. It will attempt to select the row of the MCS with largest possible group of methods.

3. If the request can be satisfied by the active row, grant a method license to the client.

A method license is granted to the client with appropriate conditions which may limit the methods a client is allowed to execute. The method license manager must assure that the conditions specified in the active row are satisfied by all clients using the object. This is done by using a technique called *license division*, which is discussed below.

4. Otherwise, if the request can be satisfied by another row that contains the methods of the active row, then set that row as the active row. A method license is granted to the requesting client and compatible licenses are sent to existing clients through the callback from the server to the clients.
5. Otherwise, the active row must be changed. The steps to change the active row are the following.
 - (a) Revoke the licenses for the active row using the callback.
 - (b) Select a new active row.
 - (c) If a new active row can be selected, grant a method license to the client.
 - (d) Otherwise, grant the client an exclusive license.

The MLM uses a row of the MCS to grant method licenses to clients. When granting a method license to a client, the MLM must assure that the MCS row condition, the condition for the active row of the MCS, remains true. It does this by dividing the condition among the clients with those license in a technique we call *license division*. In other words, a set of predicates is created where each predicate is placed in a method license and the conjunction of the predicates is equivalent to the MCS row condition. The manner in which the condition is divided is dependent upon the manner in which the condition is represented as well as some method of predicting which methods invocations are likely to be done by each client.

For example, consider row 10 of Figure 5. When providing a license to a client, the MLM must assure that the sum of all parameters passed to the invocations of the *withdraw()* method is less than the current balance. To do this, it limits the sum of parameters in each method license. For example, if 3 clients have method license, and the current balance is 300, then the condition could be divided up among the clients so

that each client could execute methods as long as the sum of the withdraw parameters is less than 300.

Because the MCS row condition is shared among clients, it may be necessary to dynamically adjust the conditions. This may done transparently in the background by the MLM. While not implemented in our current prototype, these adjustments may be done asynchronously in order to minimize the impact to the latency of the client's method invocations. To adjust the licenses, techniques such as those used by commodity tokens [5] can be used to make these adjustments.

5 Experimental Results

To evaluate the advantages and disadvantages of the method license approach, we have integrated these ideas into a working prototype. This prototype uses the framework we developed for caching Java RMI objects [4]. In particular, we evaluate the use of these ideas in conjunction with a Bank Account object that is accessed by several clients. To evaluate the impact of concurrency, we tailored our experiments using parameters which included the number of accounts shared by the clients and the number of deposit and withdraw methods executed by the client.

5.1 Experimental Setup

To evaluate the client's performance, we run a server process that creates a number of the bank account objects and starts client processes on remote clients that access those objects. These client processes then use a pregenerated script to determine which methods to execute. Each clients runs its script and then reports statistics to the server when it completes.

A script generator is used to generate the scripts run by each client. Our script generator permits us to set the parameters of the experiments. These parameters include the number of clients accessing the set of bank account objects, the percentage of shared objects that are accessed by all clients, and the percentage of method invocations that are *update* (either *deposit* or *withdraw*) methods. A random number generator is used in conjunction with these parameters in order to generate the scripts execute by the clients. The script generator is designed such the when the number of client is changed, the methods executed remain the same. This permits us to verify at the end of a test run that the results are correct.

Our experiments used 8 Sun computers with SPARC microprocessors for the server and clients. The server had a processor speed of 450 MHz and a memory size of 2 gigabytes. The client's processor speeds were between

Table 1. Response Time (in μSec)

Configuration	1	2	4	8
RMI 20	949	1079	1332	2017
LICENSE 20	427	1012	1896	3740
LICENSE 80	1325	3432	6202	12435
LICENSE 100	34	45	52	84
LICENSE-WEAK 20	47	53	58	92
LICENSE-WEAK 80	275	118	150	238
LICENSE-WEAK 100	352	199	246	481

296 and 520 MHz and their memory sizes were between 128 megabytes and 1280 megabytes. The systems were connected via a switched Ethernet network.

5.2 Scenarios Tested

We evaluated three different techniques for our experiments. The first technique, RMI, used normal Java RMI. The second technique, LICENSE, used method licenses without weakening the *balance()* method. The third technique, LICENSE-WEAK, used method licenses which weakened the *balance()* method using a TIME condition that stated that the effects of a deposit or withdraw may not be available for at most four seconds.

For each technique we tested 3 configurations. In the first configuration, labeled as 20, only 20 percent of the accounts were shared and only 20 percent of operations were update methods. In the second configuration, labeled as 80, 80 percent of the accounts were shared and 80 percent of the methods were update methods. In the third configuration, labeled as 100, all the accounts were shared and all the methods were deposit or withdraw methods.

Then for each technique and configuration, we measured the average number of microseconds taken for each method invocation as we changed the number of clients. We measured the response time for the cases of 1, 2, 4, and 8 clients. These results are show in Table 1.

For the RMI technique, the table only shows the response time for configuration 20. While we also ran configurations 80 and 100, the response times were nearly identical. For the RMI case, it can be seen that the response time more than doubles as the number of clients changes from 1 to 8.

For the LICENSE technique, the best performance was achieved for configuration 100, when all the accounts were shared and the methods only consisted of update operations. In this case, all clients could obtain a method license for executing the deposit and

withdraw methods and could execute the operations for the duration of the test without the need to contact the server. In this case, the response time ranged from between 3.6 and 4.2 percent of the RMI time. This configuration shows that the method license performs very well when the client invokes methods that are known to commute.

On the other hand, the other LICENSE configurations performed poorly. In these cases, each client executed a mix of update and balance methods, where each type of method required a different method license. Because of this mix, frequent license changes were required. For example, our infrastructure measured that the 2 client case of the LICENSE 20 configuration required more than 5000 requests for a valid license while the 2 client case of the LICENSE 100 configuration only required one request license request per object. These frequent license changes severely impacted the response time. As seen in the table, the performance becomes increasingly worse as more sharing takes place. More sharing takes place when the number of clients is increased or when the number of shared objects is higher. These configurations show that the method license approach will not perform better if the application uses methods that do not commute.

However, when consistency is relaxed for the *balance* method, the LICENSE-WEAK technique performed much better than the standard method license approach. In this case, the response times range from between 4.4 and 29 percent of the corresponding RMI response time.

For the LICENSE-WEAK technique and configuration 100, the overhead of the asynchronous updates to the server impacted the response time, causing it to perform worse than the LICENSE technique. However, even in the worse case, the response time is 37 percent of the corresponding RMI time.

6 Conclusions

In this paper, we have introduced the *method commutative specification* (MCS) which defines when a group of method invocations commute. This new approach identifies sets of method invocations that commute, in contrast to existing approaches that only consider pairs of method invocations. We presented a methodology, based on Hoare Logic, to create the MCS from the semantic specification of an object. We have also illustrated how weakened consistency requirements may be added to object’s semantic specification which then results in an MCS that permits greater commutativity.

We have also defined *method licenses*, derived from

the MCS, to assure that clients access cached objects without violating the consistency of the object.

We evaluated the use of these concepts in a working prototype. We show that if the application is able to run without frequent method license changes, then the response time is at most 4.2 percent of the RMI response time. However, if the application causes a large number of changes to the method license, the performance can be much worse than RMI. Yet, this limitation may be remedied through the use of weaker consistency requirements. By using weaker consistency requirements, the best response time was 4.4 percents of the RMI response time and the worst response time was 29 percent of the RMI response time.

This paper serves as a basis for further work to improve the performance of a distributed client-server environment. In particular, the following enhancements may be made. First, the process of creating the MCS must be automated. The automation of the MCS is needed when the number of methods of an object becomes large. Second, it should also be possible for the commutativity analysis to suggest ways in which commutativity may be weakened. Third, more work should be done to permit the weakening of the CC1 and CC2 conditions. Not only should existing ideas, like operational transformation, be explored, but new ways to resolve ordering conflicts may be identified.

References

- [1] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, Mar. 1992.
- [2] C. Beeri, P. A. Bernstein, N. Goodman, M. Lai, and D. E. Shasha. Concurrency control theory for nested transactions. In *Proc. of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 45–62, Montreal, Aug. 1983.
- [3] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):341–370, Sept. 1985.
- [4] J. Eberhard and A. Tripathi. Efficient object caching for distributed Java RMI applications. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 15–35, Heidelberg, Germany, Nov. 2001.
- [5] U. Çetintemel, B. Özden, M. J. Franklin, and A. Silberchatz. Design and evaluation of redistribution strategies for wide-area commodity distribution. In *proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, Mesa, Arizona, Apr. 2001.
- [6] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, Dec. 1969.
- [7] A.-M. Kermarrec, A. Rowston, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, USA, Aug. 2001.
- [8] H. F. Korth. Locking primitives in a database system. *J. ACM*, 30(1):55–79, Jan. 1983.
- [9] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 1–7, Philadelphia, PA, May 1996.
- [10] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proc. of 1998 ACM Conference on Computer-Supported Cooperative Work*, pages 59–68, Seattle, USA, Nov. 1998.
- [11] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995.
- [12] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, Dec. 1988.
- [13] W. E. Weihl. The impact of recovery on concurrency control. In *Proc. of the Eighth ACM Symposium on Principles of Database Systems*, pages 259–269, Mar. 1989.
- [14] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions and Programming Languages and Systems*, 11(2), Apr. 1989.
- [15] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, Aug. 2002.