

Exploiting Method Semantics in Client Cache Consistency Protocols
for Object-oriented Databases

By:
Johannes Dwiartanto

Supervisor:
Prof. Paul Watson

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy



University of Newcastle

School of Computing Science

2006

NEWCASTLE UNIVERSITY LIBRARY

204 26882 3

Thesis L 8253

**The following pages are missing from the
original thesis –**

Pg. ii

Pg. iv

Pg. viii

Pg. xii

Pg. 12

Pg. 108

Pg. 138

For Angela, Joan and my families

Abstract

Data-shipping systems are commonly used in client-server object-oriented databases. This is intended to utilise clients' resources and improve scalability by allowing clients to run transactions locally after fetching the required database items from the database server. A consequence of this is that a database item can be cached at more than one client. This therefore raises issues regarding client cache consistency and concurrency control. A number of client cache consistency protocols have been studied, and some approaches to concurrency control for object-oriented databases have been proposed. Existing client consistency protocols, however, do not consider method semantics in concurrency control. This study proposes a client cache consistency protocol where method semantic can be exploited in concurrency control. It identifies issues regarding the use of method semantics for the protocol and investigates the performance using simulation. The performance results show that this can result in performance gains when compared to existing protocols. The study also shows the potential benefits of asynchronous version of the protocol.

Acknowledgements

I would especially thank to my supervisor Prof. Paul Watson for his encouragement and persistence in supervising me, and for his help throughout my study life. I think that I will never forget this.

Also, I would like to thank Dr. Jim Smith, for his supervision and support whenever I needed to discuss my work. He may not even have realised that his sharing of expertise, especially on Linux and Databases, has contributed a lot to my knowledge and skills.

Also I would thank my examiners Prof. Ian Watson and Prof. Pete Lee for providing meaningful feedback on this thesis. I would also thank Prof. Cliff Jones for his feedback during thesis committee meetings.

I would not wish to forget friends, too numerous to mention their names, as they have given me happiness throughout my study life in the UK.

Certainly I would thank my family: my mother, father and sister, because without their support I would not have been able to study.

Conducting this research study has given me the opportunity to explore knowledge, and this exercise has often made me feel up and down. I would thank my wife Angela and my daughter Joan who are always great companions, especially in difficult moments.

Table of Contents

1	Introduction	1
1.1	Data-shipping	2
1.2	Semantic-Based Concurrency Control in OODB	4
1.2.1	Read-Write conflict examples	5
1.2.2	Write-write conflict examples	6
1.3	The goal and contribution of this thesis	8
1.4	The thesis outline	11
2	Background Studies	13
2.1	A brief introduction to object-oriented databases	13
2.2	Concurrency control	17
2.2.1	Serialisability of Transactions	18
2.2.2	Nested Transactions (NT)	18
2.2.3	Open Nested Transactions	21
2.3	Semantic-based concurrency control in OODB	23
2.3.1	Method and attribute locking	23
2.3.2	Using Direct Access Vectors	28
2.3.3	Attribute locking	31
2.3.4	Notes on the approaches	35
2.3.5	Summary	37
2.4	Existing client cache consistency protocols	38
2.4.1	Avoidance-based protocols	39
2.4.2	Detection-based protocols	44
2.4.3	Performance Issues	47
2.5	Summary	60

3	Protocol Design	61
3.1	The requirements	61
3.2	The Approach	63
3.2.1	Handling a read-write conflicts	64
3.2.2	Handling a write-write conflict	66
3.3	The Protocol Design	73
3.3.1	Synchronous Method-time Validation (SMV)	74
3.3.2	Commit-time Validation (CV)	82
3.3.3	Asynchronous Method-time Validation (AMV)	85
3.4	Possible implementation	90
3.4.1	A client requesting a database item	91
3.4.2	The method validation processor	91
3.5	Summary	94
4	The Simulation Design	95
4.1	The simulation package	95
4.2	The system model	97
4.3	The database and workload model	100
4.4	Correctness	104
4.5	The limitations of the model	106
4.6	Summary	107
5	Results and Analysis	109
5.1	The metrics	111
5.2	Our O2PL implementation	112
5.3	CV vs O2PL	115
5.3.1	The measurement	116
5.3.2	Summary	119
5.4	SMV, AMV and CV	119
5.4.1	With short-length transactions	119
5.4.2	With medium-length transactions	121
5.4.3	The effect of variable transaction lengths	128
5.4.4	Summary	131
5.5	SMV with a probability of commutativity	132
5.5.1	Under HotCold	132
5.5.2	Under high data contention	135
5.5.3	Summary	137

5.5.4 Chapter Summary	137
6 Conclusions and Further Work	139
6.1 Conclusions	139
6.2 Further work	143

List of Figures

1.1	Database Management System	3
1.2	Query-shipping vs Data-shipping system	4
1.3	Order schema	5
1.4	An example of interleaving	7
2.1	An example of a <i>has-a</i> relationship	14
2.2	A method hides the detail of operations on object	16
2.3	Swaps between transaction operations	19
2.4	Concurrency control in Closed Nested Transactions	20
2.5	An example of nested transactions	22
2.6	Scenario [MWBH93] when the open nested transaction protocol was adopted	25
2.7	Modified scenario [MWBH93] when open nested transaction was adopted	26
2.8	Scenario [MWBH93] re-considering closed nested transaction	27
2.9	Approach using DAV	29
2.10	Semantic-based CC protocol	31
2.11	Attribute Locking protocol	32
2.12	An example of correctness check	34
2.13	Locks acquisition using approach in [RAA94]	35
2.14	Locks acquisition using the approach in [MWBH93]	36
2.15	The avoidance-based scheme	40
2.16	The detection-based scheme	45
2.17	Concurrency control operations in Avoidance and Detection based schemes	48
2.18	Data locality	50
2.19	Factors that influence the performance	59
3.1	Releasing read-write conflict	64
3.2	Releasing read-write conflict	65
3.3	Scenario-0: serial, non-concurrent	67

3.4	Scenario 1: concurrent T1 and T2; both commits	68
3.5	Scenario 2: concurrent T1 and T2; T1 commits, T2 aborts	69
3.6	Scenario 3: concurrent T1 and T2; T1 aborts, T2 commits	70
3.7	Scenario 4: concurrent T1 and T2; T1 aborts, T2 aborts	71
3.8	Server's algorithm	72
3.9	SMV, CV and AMV - a brief illustration	73
3.10	The basic form of Synchronous Method-time Validation (SMV) protocol	75
3.11	Algorithm for handling a validation request	81
3.12	When a commutating transaction commits	82
3.13	When a commutating transaction commits	83
3.14	The Commit-time Validation (CV) protocol	84
3.15	The Asynchronous Method-time Validation (AMV) protocol	86
3.16	A case in the AMV protocol	90
3.17	A possible implementation of accessing an attribute in C++ -based pseudo-code	92
3.18	Method processor	92
3.19	An example of method processing	93
4.1	A screenshot of the animation of the simulator	96
4.2	Class diagram of the simulation package	98
4.3	Client-server system	99
4.4	Transaction Run	100
4.5	An example of an assertion	105
4.6	Example to illustrate the model limitation	107
5.1	O2PL vs CBL	114
5.2	Transaction in O2PL	115
5.3	Transaction in CV	116
5.4	CV vs O2PL	118
5.5	The average response time; HotCold; Short-length Tx	120
5.6	HotCold; Short-length Tx;	121
5.7	The average response time; HotCold; Medium-length; No method semantics	122
5.8	HotCold; Medium-length; No method semantics	123
5.9	Abort rate components. HotCold; Medium-length; No method semantics	126
5.10	The average response time; HotCold; Variable lengths; No method semantics	129
5.11	HotCold; Variable lengths	130
5.12	Under moderate data contention workload	133
5.13	A case in SMV	135

5.14 Under high data contention workload	136
--	-----

List of Tables

1.1	The client cache consistency protocols	10
2.1	Aspects of semantic-based concurrency control protocols	37
2.2	Probability values in HotCold and Sh/HotCold workloads	51
2.3	The protocols compared in the existing studies	52
2.4	The relative performance of the previously studied protocols	58
4.1	The System Parameters	100
4.2	Probability values in HotCold and Sh/HotCold workloads	102
4.3	Database and workload parameters	104
5.1	Parameter Values for O2PL vs CBL	113
5.2	The differences in the simulation settings	114
5.3	Parameters in CV vs O2PL	117
5.4	Workload and System Parameters	134
5.5	Workload and System Parameters	135

Chapter 1

Introduction

Object-oriented databases (OODB) are powerful because of their ability to handle complex applications, such as found in computer aided design (CAD), computer aided manufacturing (CAM), geographic information systems (GIS) and multimedia applications. This is because they support a data model that is capable of handling complex relationships between various types of objects. OODB also supports queries that are capable of traversing these data relationships and calling user-defined methods. Moreover, unlike in relational databases, updating a database schema in an OODB can be performed naturally, and this is particularly beneficial when a large number of objects are affected [CB02]. In addition, OODB is also known for its high performance, which is invaluable in some cases, such as those in mission-critical applications whose performance requirements cannot be met by relational databases [Gmb03][Cor03]. Examples include auction systems that are required to respond to thousands of bids daily [Gil01].

Because of their great potential, object-oriented databases have been studied extensively in the past decade.

One of the key issues in OODB is client data caching and concurrency control in a data-shipping, client-server environment. In a client-server environment, data-shipping is a paradigm, in which the client stores database objects in its cache and runs transactions locally, accessing the database objects from its cache. When many clients are connected to a server, multiple clients may run

transactions that share database objects, and consequently a copy of a database object may be held by more than one client. This raises the issue of client cache consistency for these copies, and so concurrency control is required to guarantee the correctness of transactions. This can be a key factor in system performance, as shown in earlier studies [Fra96][OVU98][ALM95].

The aim of this study is therefore to investigate the exploitation of semantic-based concurrency control in client cache consistency protocols, as a solution to this issue. Before discussing this in more detail, we will first describe the data-shipping paradigm, and introduce semantic-based concurrency control.

1.1 Data-shipping

Data-shipping is a paradigm for the client-server architecture that is commonly found in object-oriented database systems (ODBMS). It is different from the *query-shipping* used commonly in relational database system [SKS02]. The difference is in terms of where the database management system (DBMS) functions are located.

A DBMS contains a component called a database manager. This is an intermediate component situated between the user (i.e. the front end) and the physical data storage, as shown in Figure 1.1.

A database manager is responsible for: scheduling, transaction management, cache management and recovery management. Data-shipping and query-shipping paradigms are different in terms of where the database manager functionality is performed, which is illustrated in Figure 1.2 and described as follows:

- In query-shipping, database objects are stored only at the server. When a client transaction runs a database query, the query is sent by the client to the server. Upon receiving the query, the server processes the query and then sends the result back to the client. The server is therefore responsible for all the database manager functionality, whereas a client does not perform any of these functions.
- In data-shipping, database objects reside both at the clients and the server. When a client's

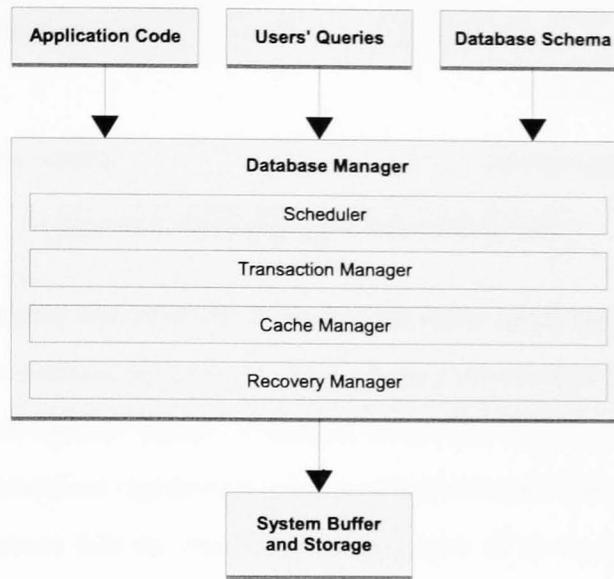


Figure 1.1: Database Management System

transaction runs a database query, first the client checks if the objects queried are in the client's cache. If the objects are not cached by the client, they are firstly fetched from the server and stored into the client's cache. Then the client continues with the transaction by reading or writing the objects from the local cache. Whenever an object is not found in the client's cache, the client firstly fetches the object from the server before continuing with the transaction. Because a client locally runs the transaction, a client is also responsible for the tasks of the database manager. Therefore data-shipping transferred some tasks from the server to clients.

As a result, data-shipping has the advantages that client machines, which are nowadays often very powerful, can be utilised to handle database manager tasks, so reducing the load on the server. This can make a server more responsive, which in turn can enhance the system's scalability in terms of the number of clients.

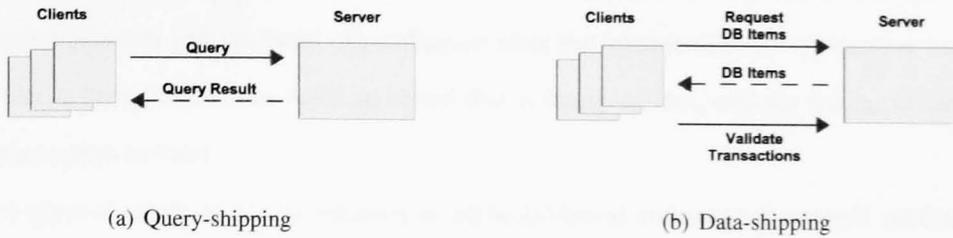


Figure 1.2: Query-shipping vs Data-shipping system

However, data-shipping also raises the issue of client cache consistency and concurrency control. In a client-server database environment, many clients can concurrently run database transactions that access shared database objects. Therefore, in data-shipping, because clients run transactions locally, copies of database objects may reside at multiple clients simultaneously. Consequently one needs to be concerned with the consistency of the copies of the database objects held at the clients and concurrency of access to the shared objects.

To address this issue, a number of client cache consistency and concurrency control protocols in data-shipping object-oriented database systems have been proposed in the past decade. Because transactions operate on the client's local objects, and each client communicates only with the server, in these protocols, the clients need to validate their local transactions at the server to check for cache consistency.

1.2 Semantic-Based Concurrency Control in OODB

Concurrency control is a mechanism to guarantee that transactions run correctly. It has been an extensive topic of research in databases for decades. Some studies of concurrency control have considered the use of method semantics to enhance concurrency in object-oriented databases as methods can be called on objects [JG98][RAA94][MWBH93]. These concurrency control schemes have been classified as Semantic-based Concurrency Control.

Essentially semantic-based concurrency control allows a conflict on a database object, either read-write or write-write conflict, to be released if the method of object run by one transaction

commutes with the method of an object run by the other transaction. When there is a commutativity relationship between two methods, depending on what the relationship is, a transaction can read an object that is being written, or write an object that is being written, without having to wait for the other transaction to finish.

Two types of conflicts can be released by semantic-based concurrency control: read-write conflict and write-write conflict. We will now describe examples of these type of conflicts. A detail description about semantic-based concurrency control is given in Section 2.3.

1.2.1 Read-Write conflict examples

The read-write conflict example is taken from the scenario of an order cancellation. Supposed the *Order* object has methods shown in Figure 1.3.

<pre> class Order { attribute String status; .. //method M1 void snapshotOrder() { read status .. } //method M2 void cancelOrder() { status = "CANCELLED" .. } } </pre>	<p>Compatibility Matrix</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th></th> <th>M1</th> <th>M2</th> </tr> </thead> <tbody> <tr> <th>M1</th> <td>X</td> <td>Commute</td> </tr> <tr> <th>M2</th> <td>Commute</td> <td>X</td> </tr> </tbody> </table>		M1	M2	M1	X	Commute	M2	Commute	X
	M1	M2								
M1	X	Commute								
M2	Commute	X								

Figure 1.3: Order schema

Supposed that all orders in an organisation are being recapitulated (read and summarised) in a transaction. While it is in progress, all *Order* objects are read and locked to prevent the objects from being written. Supposed a client has placed an order but then it is canceling the order while that order is being read. When canceling the order, the client will try to write the status of the order, but

because the Order object is being read, the client must wait for the transaction performing the read to end.

Suppose that the organisation has a flexible system that allows the customer to cancel an order while the order is being read. Semantic-based concurrency control can allow this by defining a semantic relationship, for example between `cancelOrder()` and `snapshotOrders()`, so that although the read on `status` in `snapshotOrder()` conflicts with the write on `status` in `cancelOrder()`, the conflict can be released (ignored). The relationship is defined as `commute` in a compatibility matrix as shown in Figure 1.3.

An illustration of the interleaving of the transactions in the above scenarios is shown in Figure 1.4. Transaction 1 reads the status of all the orders and Transaction 2 runs the order cancellation. Without semantic-based concurrency control the customer cannot cancel the order because the write on the `status` in `cancelOrder()` can proceed only when the read on the order has finished, which is at the end of the Transaction 1, so the customer must wait until the entire Transaction 1 has finished. By contrast, with semantic-based concurrency control, `cancelOrder()` can proceed, and the customer does not need to wait for the entire Transaction 1 to finish.

1.2.2 Write-write conflict examples

Another type of conflict that can be released using semantic-based concurrency control is a write-write conflict. Some examples of write-write conflict that can be released are as follows:

1. Two addition operations can be commutative. For example, operation A is $i = i + 1$, and operation B is $i = i + 1$. If operation B is independent from operation A, then operation A commutes with operation B by assuming that each operation is atomic and the intermediate result is not important so that whether operation A is performed before or after operation B does not matter.
2. Another example is a scenario in car rental in that the status of an order can be “shipped” or “paid” or “paid and shipped”, and a car can be shipped before or after the rent is paid

Tx-1 (Organisation)	Tx-2 (Client)
Starts (recapitulating orders)	
..	
for each order o {	
o.snapshotOrder() starting	Starts (cancelling order)
	o.cancelOrder() starting
	..delayed..
o.snapshotOrder() done	
}	
..	
End of Tx-1	
	o.cancelOrder() done
	End of Tx-2

(a) Without Semantic-based CC

Tx-1 (Organisation)	Tx-2 (Client)
Starts (recapitulating orders)	
..	
for each order o {	
o.snapshotOrder() starting	Starts (cancelling order)
	o.cancelOrder() starting
o.snapshotOrder() done	o.cancelOrder() done
}	End of Tx-2
..	
End of Tx-1	

(b) With Semantic-based CC

Figure 1.4: An example of interleaving

[MWBH93]. Thus, suppose an order is of type `Order` and `status` is its attribute, and suppose that the `pay()` method updates `status` to “paid”, and that the `ship()` method updates the `status` to “shipped”. A transaction `T1` that runs the `pay()` method will not conflict with another transaction `T2` that runs `ship()` method. When the two transactions run concurrently the value of the `status` will be “paid and shipped”. Therefore, here `T1` does not need to wait until `T2` finishes and vice versa.

3. Sometimes an updated result can be regarded as temporary. Consider an SVG-based (Scal-

able Vector Graphic) map showing objects whose information can be viewed. For instance, detailed information about a building in a street could be recorded and the information are represented as attributes of the street object. To update the map, a number of mobile agents traverse the streets. Because they can perform updates simultaneously, an update by one agent can overlap with an update by another agent, and the final value of an update will later be decided by a third party. Thus, here two simultaneous additions to the map can be allowed as their results can be regarded as temporary.

4. Another example comes from a study of real-time air traffic control [PLP97]. A radar reading is written into a record and shown on a radar display. However, because of the real-time requirement, the write operation can be overwritten by a new radar reading that arrives. Thus, here two methods that perform Write operations can be performed simultaneously.

In all of the preceding examples, it should be noticed that a write-write conflict between two operations can be released if the operations are independent from each other in that the result of one operation is not read by the other operation.

1.3 The goal and contribution of this thesis

As described in the preceding section, method semantics can be used to remove conflicts. Some approaches to semantic-based concurrency control have been previously studied, and are described in Section 2.3. However, semantic-based concurrency control has not been investigated in a data-shipping client-server environment. Moreover, in existing client cache consistency protocols for a data-shipping client-server environment, conflicts are handled on a page or object, and the potential of method semantics is not considered. Therefore our study fills this gap by investigating client cache consistency protocols that can exploit method semantics in concurrency control.

The following are our contributions of this thesis:

- We designed a new client cache consistency protocol that can exploit method semantics and described the issues this raised. We named the protocol Synchronous Method-time Validation

(SMV) as a client validates its transaction at the method level, at the end of each method call. The reason for this is to preserve the atomicity of a method, as required in semantic-based concurrency control*. Therefore SMV differs from the existing client cache consistency protocols in that the client does not validate at page or object level but at a method level. The protocol is synchronous as the client waits for the result of a validation before continuing its transaction. SMV also differs from the existing client consistency protocols in that the lock granularity is an object's attribute, not an object or page. Thus, our protocol differs from the existing protocols in terms of how a client validates transactions, and in terms of the lock granularity. To our knowledge SMV is the first client cache consistency protocol in object-oriented databases in which the client validates at the method level and uses an object's attribute lock granularity.

Moreover, in our design we investigate the use of method semantics to release read-write conflicts and write-write conflicts at both the client and server with Synchronous Method-time Validation (SMV).

In addition, we designed an asynchronous version of the protocol, named Asynchronous Method-time Validation (AMV). AMV is similar with SMV in that a client validates its transaction at the end of a method call. However, with AMV, the client does not wait for the result of a validation after sending a validation message, but instead continues its transaction until commit-time. The aim was to improve performance by reducing waiting. However, the implementation of the basic form of AMV, i.e. without method semantic-based commutativity support, in this thesis has met much more complexity than that of SMV, and so we believed it will require much more extra overhead if we implement AMV that uses method semantics in concurrency control. Therefore, in this thesis we will investigate the behaviour of AMV protocol compared to the other protocols, but only with their basic form.

The differences between SMV, AMV and the existing client cache consistency protocols are shown in Table 1.1.

*A description of method atomicity in semantic-based concurrency control is given in Section 2.2

	VALIDATION				
	On Write Access		Commit time	End of Method	
	Sync.:	Async.:		Sync.:	Async.:
	[Fra96] [LLOW91] [RW91]	[OVU98]	[Fra96] [CFLS91] [WN90]	[This thesis]	[This thesis]
USE SEMANTIC-BASED CONCURRENCY CONTROL	×	×	×	√	×

Table 1.1: The client cache consistency protocols

- We investigated the performance characteristics of these new protocols using simulation. Comparisons were made with an optimistic version of the protocols named Commit-time Validation (CV). Like SMV and AMV, CV uses attribute locking granularity, and is an optimistic avoidance-based protocol like O2PL (Optimistic Two-Phase Locking)[Fra96]. We wished to observe the performance differences between our new synchronous and asynchronous method-time validation protocols (SMV and AMV) (which are pessimistic) and Commit-time Validation (CV) (which is optimistic), because in a previous study the optimistic protocol (Optimistic Two Phase Locking - O2PL) has been found to be superior to a pessimistic protocol (Callback Locking - CBL) [Fra96]. Our results show that the SMV (pessimistic, synchronous) and the AMV (pessimistic, asynchronous) protocols can outperform CV (optimistic). We also investigate the performance sensitivity when the number of operations in a transaction varies, which was not addressed in the previous studies [Fra96][OVU98][ALM95]. Our results show that under higher number of operations in a transaction, the abort rate (i.e. the number of aborts per committed transaction) in the asynchronous protocol (i.e. AMV) is better than the abort rate of the other protocols.

To justify the optimistic behaviour of our CV, we compare our CV with the existing Optimistic Two-Phase Locking (O2PL). The result shows that despite some differences on their performance characteristics, their throughput are not different significantly.

Then, we investigate what could be gained if the support for method semantics is incorporated

in our protocol, by comparing between our synchronous method-time validation (SMV) that assumed no commutativity and the SMV that assumed a certain probability of method commutativity. The result shows that method semantics can give better performance in particular circumstances.

1.4 The thesis outline

This thesis is organised as follows. The next chapter (Chapter 2) firstly describes object-oriented databases and the semantic-based concurrency control approaches. Then the existing client cache consistency protocols are described, focusing on a review of their performance and the aspects that influence the performance.

Chapter 3 describes the design of our new protocols: Synchronous Method-time Validation (SMV) and Asynchronous Method-time Validation (AMV). Issues when method semantics are used in the protocols and aspects of the implementation of the protocols are explained.

Chapter 4 contains the design of the simulation to measure and compare the performance of the protocols. It includes the description of the model of the simulation, as well as the limitation imposed by the model.

Chapter 5 describes the results of the measurements. Each result is accompanied by an analysis to identify the characteristics of the protocols, and to compare their behaviour.

Finally, Chapter 6 contains the conclusions drawn from the study, and identifies options for further work.

Chapter 2

Background Studies

This chapter contains the background studies that are relevant to semantic-based concurrency control and client cache consistency protocols in object-oriented databases (OODB). It begins with a brief introduction to object-oriented databases, which particularly describes the abstractions supported by OODB to emphasise the difference between OODB and relational databases that are relevant to this thesis. This is followed by a review of the approaches to semantic-based concurrency control. However, beforehand some underlying background will be described that include concurrency control and nested transactions. Then, existing client cache consistency protocols will be reviewed. In particular, the description will focus on the identification of aspects that influence the performance characteristics of the protocols.

2.1 A brief introduction to object-oriented databases

The conceptual schema of a database is in general an entity-relationship model. An *Entity* is an object that is either real or abstract, and it has a number of attributes that define its properties. A *relationship* is the inter-relation between entities.

In a complex application, the database schema consists of a large number of entities and complex relationships. Object-oriented databases (OODB) are suitable for applications that contain complex

data relationships because the object-oriented database model has the advantage that is to be able to use abstractions to reduce the complexity. The following abstractions are supported by OODB:

- 1. Aggregation. Aggregation is an abstraction to define that an entity has some other entities. The relationship is generally known as a *has-a* relationship.

In object-oriented databases (OODB), a has-a relationship can be defined by having a has-a attribute that refers to another object. For example, as shown in Figure 2.1(b), to define a relationship that an Order involves a number of Items, the has-a attribute of object Order can be set as a reference to a list of Item objects. An update on an Item that is being referred by the has-a attribute of an Order, automatically updates the has-a attribute of the Order. Also in the case of deletion, if an Item, which is being referred by an Order, is deleted so that the list of Item is empty, the has-a attribute will automatically refer to an empty list. Thus, in OODB relationships are naturally constructed.

By comparison, in relational databases an entity or a relationship can be represented only as a relation, which is also called a table. Therefore, in relational databases, to define a has-a relationship one must define a separate has-a table.

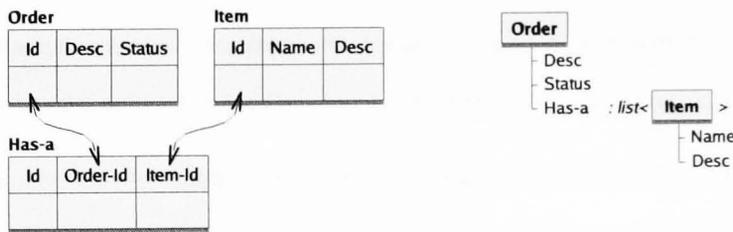


Figure 2.1: An example of a *has-a* relationship

The implication of requiring a separate relation to define a relationship in relational databases is that in application with many has-a relations, extra overhead is needed to ensure the integrity of the relations. In relational databases, a relation has attributes that define the prop-

erties of the relation*, and a foreign-key attribute is an attribute that refers to another key attribute in another table. Therefore, one must ensure that a foreign-key attribute has the same type as the attribute in its referred relation. As an example, consider “Order”, “Item” and “has-a” relations in Figure 2.1(a). The “has-a” relation should be made to define a relationship that an Order involves one or more Items. The Order-Id and the Item-Id in the has-a table are foreign-keys to the Id in the Order table and the Item table respectively. One must ensure that the type of the Order-Id attribute is the same as the type of the Id attribute in the Order table, and the type of Item-Id attribute in the has-a table is the same as the type of the Id attribute in Item table. Moreover, one must ensure that an update of one value of a key attribute in one table also updates the key attribute in the other table. As in the example, an update in the value in the foreign keys in has-a table must also update the key values in the Order and Item tables. Also a deletion on an entry in the Order or Item tables must also delete the values in the has-a table. Although some relational database products nowadays support integrity checking, the support is somewhat limited, such as in the current MySQL[AB04] where an integrity check is supported on update and deletion but not on attribute type definition. Therefore extra overhead in integrity checking is still needed.

2. Method calls. Object-oriented databases (OODB) can support method calls. A method is an interface to execute a sequence of operations on an object, which is illustrated in Figure 2.2. Thus some details of operations can be performed by calling a method inside the database.

A method can be invoked from within a method, so forming nested methods. In Section 2.3, we will discuss how the semantic-based concurrency control approaches consider nested methods, and described the nested transaction model [Mos85][GR92] within concurrency control mechanisms.

Calling methods on objects is not found in relational databases. In relational databases, operations need to be performed outside the database. Operations are performed on data that must

*An attribute in a relation is a column in a table

```
Method()
{
    Camera camera;
    camera.setName("LEICA");
    cameraCollection.add(camera);
}
else {
    return false;
}
}
}
}
return new String("Not satisfied at all");
return new String("Fully satisfied");
}
```

Figure 2.2: A method hides the detail of operations on object

be firstly extracted from the database through queries.

3. Other abstractions. There are some other abstractions in object-oriented databases that are not found in relational databases, which include generalisation and polymorphism. However, they are not described further here as they are not particularly relevant with this thesis. Further details can be found here [KM94].

The preceding description contains abstractions in object-oriented databases (OODB) that are relevant for this thesis. Next we will describe how concurrency control can take advantage of the object-oriented database model.

2.2 Concurrency control

Semantic-based concurrency control is a concurrency control scheme in object-oriented databases that exploits method semantics. Before we describe semantic-based concurrency control, we will briefly describe concurrency control in general and explain terminologies such as transaction, schedule and conflict.

Transaction in databases is a way to perform database operations and it has the following properties: Atomicity, Consistency, Isolation and Durability [GR92].

- With the Atomicity property, a transaction finishes entirely (commits) or does not execute at all (aborts).
- With the Consistency property, any consistent database state is transformed to another consistent database state. Database state here is the value of a database item, and “consistent” here means that the value of a database item satisfies its constraints in the database schema.
- With the Isolation property, no interference to a transaction gives bad effect when the transaction is executing. In other words, when there are many transactions executing at the same time, this isolation property makes the final result to be as that if the transactions had executed one at a time.
- With the Durability property, the results of a transaction are not lost but recorded in a stable storage for future use.

A *schedule* is a sequence of transaction operations ordered by time. For example the following is a schedule containing Read (R) and Write (W) operations on database item X and Y, by two transactions T1 and T2:

$$R_1(X), W_1(X), R_2(X), W_2(X), R_1(Y), W_1(Y), R_2(Y), W_2(Y)$$

A *conflict* occurs if a schedule containing two consecutive operations (from different transactions) involve the same database item and one of the operations is a Write. For example, in the

preceding schedule, the following operations are in conflict because transaction T1 operates Read on X and then T2 consecutively operates Write on X

$$\dots, W_1(X), R_2(X), \dots$$

2.2.1 Serialisability of Transactions

When many transactions are executing concurrently, the schedule will probably contain an interleaving of the read and write operations of the transactions. A schedule that contains an interleaving of the read and write operations of transactions is called *serialisable* if the schedule can be transformed into one in which the transactions execute one at a time without interleaving.

A schedule is *correct* if it is serialisable.

A way to determine whether a schedule can be transformed into a serialisable (i.e. correct) schedule is by performing a series of swaps between two adjacent non-conflicting operations [GMUW00], and if the end result is that the transactions execute one at a time without interleaving, then the schedule is serialisable. For example, the following schedule has interleaving read and write operations but the schedule is serialisable because the swaps can be performed. The steps are illustrated in Figure 2.3.

$$R_1(X), W_1(X), R_2(X), W_2(X), R_1(Y), W_1(Y), R_2(Y), W_2(Y)$$

In reality, transactions execute dynamically in that we do not know when a transaction will end. Consequently, to achieve a correct (i.e. serialisable) schedule, it is not practical to wait for an entire schedule to complete and then check if the schedule is serialisable. To guarantee that a schedule is serialisable while the transactions are executing, a mechanism is required that is called *concurrency control*.

2.2.2 Nested Transactions (NT)

The executions of transactions in the preceding description are in a flat sequence. By comparison, Nested Transactions (NT) are transactions that are not in flat sequence. A nested transaction is a

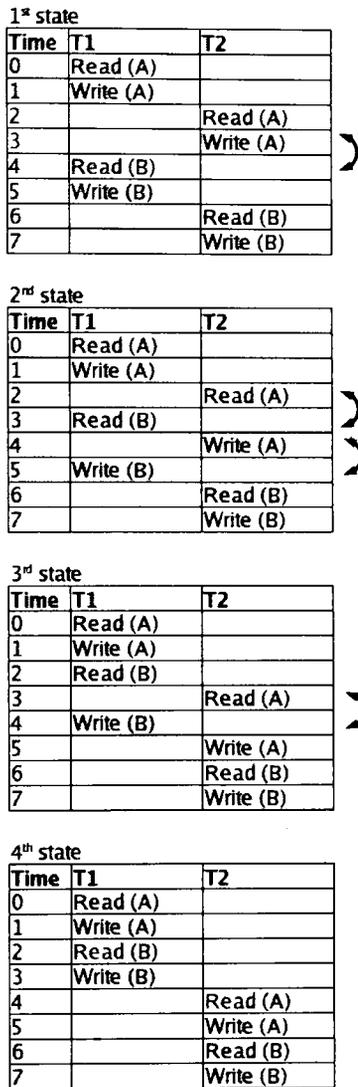


Figure 2.3: Swaps between transaction operations

transaction that executes within another transaction. The entire transaction executions in NT forms a tree, such that a transaction has some nested transactions as its branches, and a branch transaction can have other nested transactions as its sub-branches, as so on.

Firstly, we define some terminologies in a tree of transaction executions:

- The transaction at the top of the tree is called *top root transaction*.
- Transactions that are below a transaction T are called *descendants* of T. Transactions that are direct descendants of T are called sub-transactions of T.
- Conversely, transactions that are above a transaction T are called *ascendants* of T, or an ancestor to T. A transaction that is direct ascendant of a transaction T is called the *parent transaction* of T.

One of the advantages of NT is that sub-transactions can execute concurrently. However, if these transactions share a Read or Write on a database item X, the following algorithm can be applied for the concurrency control:

```

if a transaction T requires a lock on X
  if no lock conflict on X occurs,
  or if an ancestor to T has the lock on X
    the lock on X can be granted to T
  else
    T waits

if a transaction T finishes ,
  if T is top root transaction
    T and T's descendants can commit
  else
    give all T's locks to T's parent

if a transaction T aborts ,
  All T's locks are released
  Abort all T's descendants if any

```

Figure 2.4: Concurrency control in Closed Nested Transactions

The concurrency control algorithm in Figure 2.4 shows that a finished transaction T cannot commit by itself if it is not a top root transaction. All locks previously held by a finished non-root transaction are not released but given to its parent transaction. This protocol is for Closed Nested Transactions. The purpose of giving all locks to its parent transaction is to prevent its states (i.e. the values of the accessed database items) from being visible to other transactions. This is because

although a sub-transaction has finished, it can abort later if its parent transaction aborts, and then consequently its states are lost. Therefore to prevent inconsistency, all locks of a finished sub-transaction are not released but transferred to its parent transaction. Thus, if the top root transaction is the ancestor of all transactions within the tree, the preceding rule infers that when the top root transaction aborts, the entire transactions in the tree also abort.

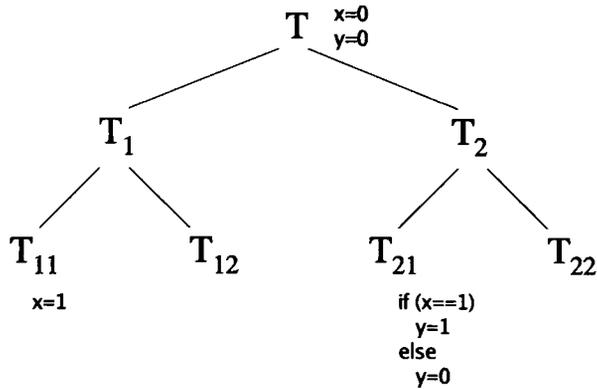
2.2.3 Open Nested Transactions

The preceding description of nested transactions, in which a finished sub-transaction cannot commit by itself, is known as Closed Nested Transactions. Another scheme for nested transactions called Open Nested Transactions [GR92], allows a finished sub-transaction to commit by itself, so after a sub-transaction finishes all locks are released and it can commit.

To illustrate, Figure 2.5 shows an example of a scenario in nested transactions that is handled using closed nested transactions and open nested transactions approaches. Supposed in transaction T, database items X and Y are initialised to 0, and T executes T1 and T2. Then T1 executes transactions T11 and T12, while T2 executes transactions T21 and T22. T11 and T21 share X: T11 performs Write on X, and T21 performs Read on X. Their executions ordered by time are shown in the table. Supposed T11 executes first at time 0 and then followed by T21 at time 1. At time 2 T11 writelocks X and at time 3 T11 updates X. Then at time 4 T21 is trying to readlock X, but a Read-Write conflict occurs so that T21 has to wait.

Supposed T11 has finished at time 5. Using the Closed nested transaction scheme, at that time T11 does not commit and the lock on X is given to T1. This is to prevent the value of X, which now equals to 1, from being visible to T21. The reason is that T11 may eventually aborts, because if T1 aborts T11 will abort as well, and when T11 aborts, the value of X is lost and become 0 again. Supposed that at time 6 T12 has not finished, so T1 cannot give the lock on X to the top root transaction T. At time 7 after T12 has finished, the lock on X is given to T (by T1), so T21 can now obtain readlock on X.

By comparison, using Open nested transactions, at time 5 when T11 has finished T11 commits,



Time	Closed Nested Transaction		Open Nested Transaction	
	T ₁₁	T ₂₁	T ₁₁	T ₂₁
0	Start		Start	
1		Start		Start
2	Writelock x		Writelock x	
3	x=1		x=1	
4		Request Readlock on x. Wait until lock on x is given to T		Request Readlock on x. Wait until lock on x is given to T ..
5	Finish. Lock on x is given to T ₁		Finish. Lock on x is given to T	
6		..wait for T ₁₂ to finish..		Readlock x
7		[T ₁₂ has finished, lock on x is given to T] Readlock x		Writelock y
8		Writelock y		y=1
9		y=1		Finish. Lock on x and y are given to T
10		Finish. Lock on x and y are given to T ₂		

Figure 2.5: An example of nested transactions

and the lock on X is released (given to the top root transaction T). Therefore, at time 6 T21 can obtain a readlock on X. Note that the readlock is obtained sooner here than using Closed nested

transactions where it is at time 7. However, if eventually T1 aborts, T11 will abort too, and then the value of X that T21 has read as 1 will become 0. Hence, open nested transactions give more concurrency but may endanger consistency.

2.3 Semantic-based concurrency control in OODB

The preceding description gives a brief background on concurrency control and that in nested transactions. Object-oriented databases (OODB) supports method calls, as mentioned in Section 2.1, and when a method calls another method, the entire execution forms nested methods. In semantic-based concurrency control, the way to perform concurrency control in nested methods has been adopted from that in nested transactions. This section describes the approaches that have been studied.

In general, three approaches to semantic-based concurrency control have been proposed, each of which was proposed in a different study.

- Method and attribute locking [MWBH93]
- Attribute locking [RAA94]
- Method or attribute locking using a Direct Access Vector [JG98]

The approaches differ in terms of the granularity of locking and how locks are represented. We will describe the approaches.

2.3.1 Method and attribute locking

This approach [MWBH93] applies the concurrency control protocols of nested transactions to the concurrency control of nested methods. The approach began with adopting “open nested transaction” to give better concurrency than “closed nested transaction”. However, then the study identified a limitation, and thereafter the approach adopted closed nested transaction.

In this semantic-based concurrency control approach [MWBH93], before method execution, a

lock needs to be acquired on all an object's attributes[†] accessed within the method. Throughout this section, we will abbreviate object's attribute to "attribute".

As an illustration of this approach, supposed two transactions T_1 and T_2 share an attribute X . When a transaction T_1 is about to execute a method, it tries to acquire lock on all items accessed in the method. When a lock conflict on item X occurs because X is already locked by the other transaction T_2 , transaction T_1 must wait until the lock on X is released by transaction T_2 . However, if a commutativity relationship occurs between the methods of T_1 and T_2 or their ancestor, that calls X , then T_1 can acquire the lock on X without waiting until the entire T_2 has finished. To define a commutativity relationship between methods, a method is associated with a "semantic lock", and in the object schema a matrix of relationships is constructed between the semantic locks (i.e. methods). A semantic commutativity relationship exists if the commutativity is defined between two semantic locks in the matrix.

As mentioned, the open nested transaction approach was initially adopted for handling nested methods. In this, when a method has finished, locks on items operated within the method are released. As an illustration, consider the example in Figure 2.6. There are two transactions, T_1 and T_2 , which share an item $o_2.X$. Supposed there is a semantic commutativity relationship between $method_{11}()$ and $method_{21}()$ of object o_1 . In stage-1, transaction T_1 is holding a Writelock on $o_2.X$; thus transaction T_1 is also holding semantic locks on $o_1.method_{11}()$ and on $o_2.method_{12}()$. Meanwhile, transaction T_2 is requesting a Readlock on $o_2.X$. Due to read-write conflict on $o_2.X$, T_2 needs to wait until the Writelock on $o_2.X$ is released. In stage-2, T_1 has finished accessing $o_2.X$ as well as methods $o_2.method_{12}()$ and $o_1.method_{11}()$. By adopting the open nested transaction approach, the lock on $o_2.X$ is released, and the semantic lock on $o_2.method_{12}()$ is released, and because $o_1.method_{11}()$ has finished too the semantic lock on it is also released. But as T_1 has not entirely finished, T_1 retains the semantic lock on $o_1.method_{11}()$. At this time, because $o_1.method_{11}()$ and $o_1.method_{21}()$ commute and both of the methods are ancestors of $o_2.X$, the Readlock on $o_2.X$ can be acquired by T_2 . Notice that if the commutativity relationship does not exist, T_2 has to wait until

[†]Object's attribute refers to attribute that is atomic.

T₁ has finished to have the request for a Readlock on o₂ granted.

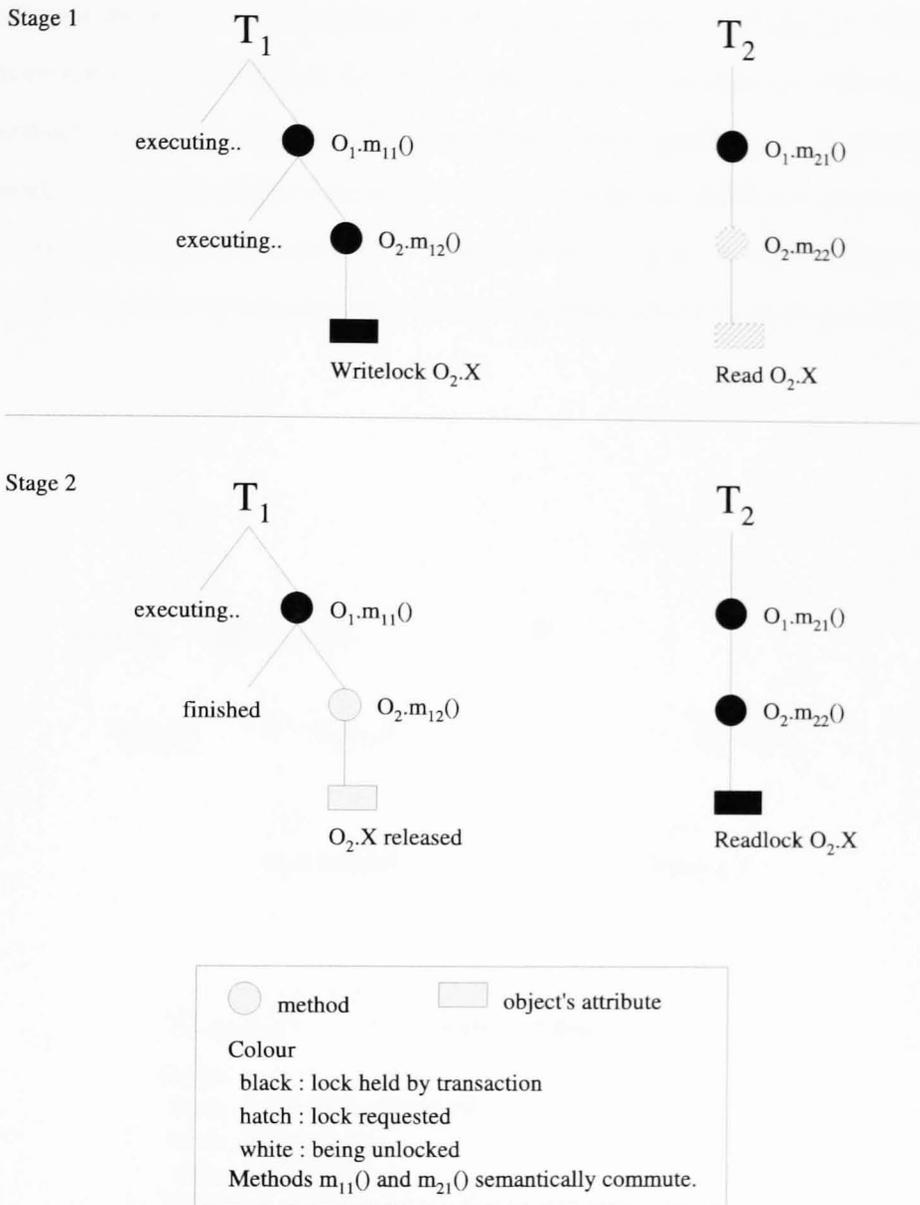


Figure 2.6: Scenario [MWBH93] when the open nested transaction protocol was adopted

However, a limitation was found when the open nested transaction approach was adopted. Con-

consider a modified scenario from the above example in which T_2 directly executes $o_2.method_{22}()$ (i.e. without intermediate method $o_1.method_{21}()$ in T_2), and supposed there is a semantic commutativity relationship between $method_{12}()$ and $method_{22}()$ of object o_2 , as shown in Figure 2.7. The lock on $o_2.X$ has been released and T_1 retains only its semantic lock on $o_1.method_{11}()$, while T_2 is requesting a Readlock on $o_2.X$. At this point, T_2 cannot detect a lock conflict on $o_2.X$, but because T_1 has not entirely finished, granting the Readlock to T_2 must be based on detected commutativity relationship between $method_{12}$ and $method_{22}$ of object o_2 . Thus, using the open nested transaction approach, a conflict can only be identified between two top-level methods operating on the same object.

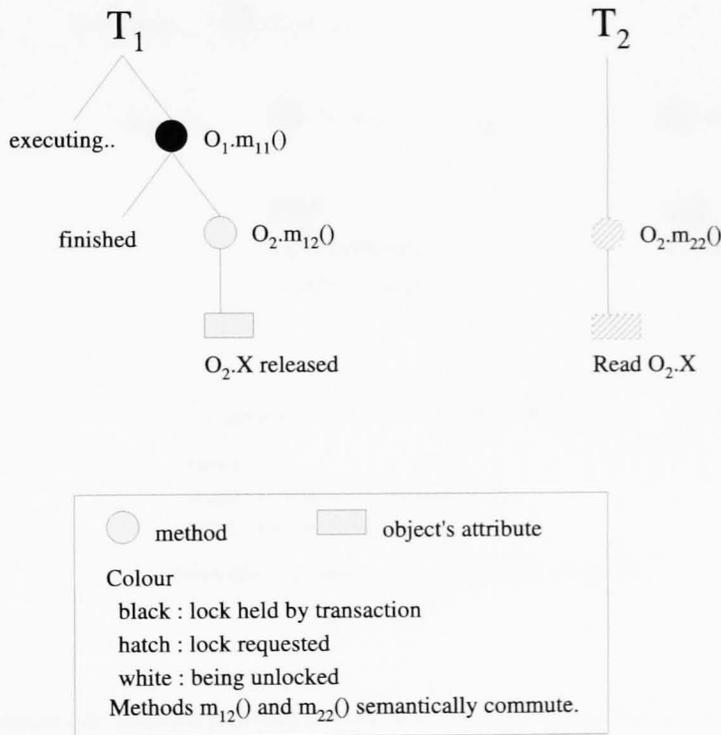


Figure 2.7: Modified scenario [MWBH93] when open nested transaction was adopted

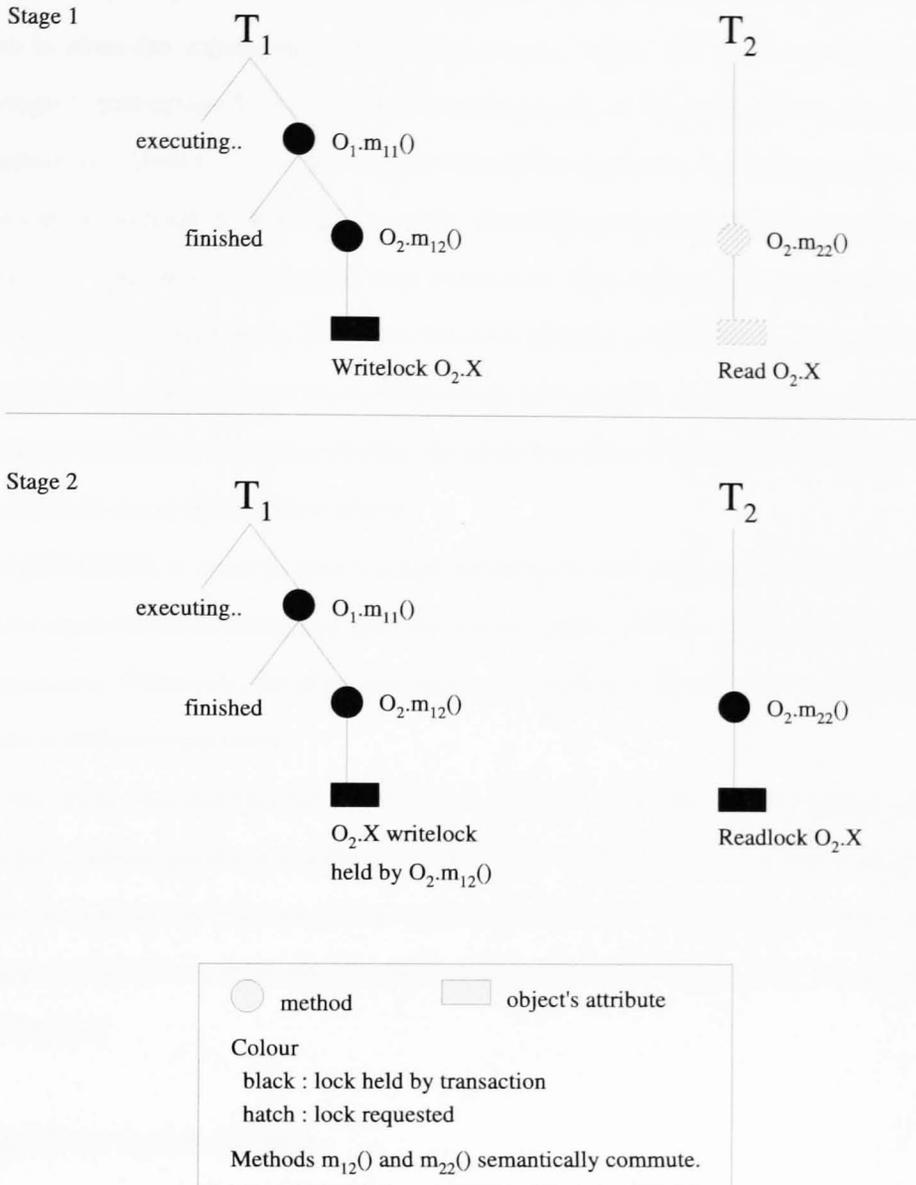


Figure 2.8: Scenario [MWBH93] re-considering closed nested transaction

In order to eliminate the above limitation, this approach [MWBH93] considered the closed nested transaction approach, in which the locks of a finished sub-transaction are not released but

are instead retained by its parent. This allows a transaction to identify conflicts, from wherever the transaction is when the transaction is requesting a lock. Figure 2.8 shows an example of a scenario. In stage-1, transaction T_1 has finished accessing $o_2.X$. At this time, T_1 has also finished method $o_2.method_{12}()$. Unlike in the open nested transaction approach, the Writelock on $o_2.X$ is given to its parent i.e. method $o_2.method_{12}()$, and T_1 also retains the semantic lock on the method (i.e. $o_2.method_{12}()$). Therefore, when at this time transaction T_2 is requesting a Readlock on $o_2.X$, it detects the read-write conflict on $o_2.X$. T_2 can however acquire a Readlock on $o_2.X$, due to the semantic commutativity relationship between $method_{12}()$ and $method_{22}()$ of object o_2 (at stage-2). Thus, T_2 can detect the lock conflict and resolve the commutativity relationship, although the two commuting methods are at different tree levels.

The study [MWBH93] is novel in showing that method semantics can be exploited by concurrency control in object-oriented databases, and identifying issues with the open and closed nested transaction approaches. Moreover, the proposed approach was able to handle the case where nested transactions run at different tree levels.

However, the study assumed that the trees of the concurrent transactions are disjoint, and one issue that was left uninvestigated was handling non-disjoint complex objects in dynamic executions of transactions. A situation in that transactions execute methods of different objects that access a shared subobject is defined as a referentially shared object (RSO) [JG98][RAA94], as described in the study by [RAA94].

2.3.2 Using Direct Access Vectors

This semantic-based concurrency control scheme[JG98] also uses the closed nested transaction approach. The novelty, however, is that the granularity of a lock can change during runtime from method level to object's attribute level in order to obtain more concurrency.

To allow changes in lock granularity, a variable called Direct Access Vector (DAV) is introduced here. DAV is a collection of access modes of all of an object's attributes accessed within a method, and the role of a DAV is to represent a lock. The DAV is generated automatically by the system. At

```

object Car
(
  attributes:
    id
    price-to-rent (p)
    quantity-on-hand (qoh)
  methods:
    [M1] adjust_price()
      [A] if QOH>10 then
        [A1] price-to-rent *= 0.9;
    [M2] check_out_rent()
      [B] if o.test_status=new then
        [B1] o.change_status( granted );
            qoh=qoh-1;
    [M3] pay_rent
      [C] read price-to-rent;
            read qoh;
            o.change_status( paid );
)

object Order
(
  attributes:
    id
    status (s)
  methods:
    [N1] test_status()
      [D] read status;
    [N2] change_status( value )
      [E] status=value;
)
    
```

DAV

[id,p,qoh]

[R,W,R]

[R,N,R]

[R,W,N]

[R,N,W]

[R,N,N]

[R,N,W]

[R,R,R]

[R,R,R]

[id,s]

[R,R]

[R,R]

[R,W]

[R,W]

DAV: R=Read,W=Write,N=Null

Car		Lock Holder							
		D(M1)	D(A)	D(A1)	D(M2)	D(B)	D(B1)	D(M3)	
Lock Requester	D(M1)	N	Y	N	N	Y	N	N	
	D(M2)	N	N	Y	N	Y	N	S	
	D(M3)	N	Y	N	S	S	S	Y	

Order		Lock Holder	
		D(N1)	D(N2)
Lock Requester	D(N1)	Y	N
	D(N2)	N	N

Note: Y=commute; N=not commute; S=semantically commute

Figure 2.9: Approach using DAV

the start of a method it contains the access modes of all the object's attributes that are accessed within the method. During the execution of the method, at a point when an attribute is to be accessed, the system changes the DAV to contain only that attribute.

Figure 2.9 shows some examples of a DAV in a car rental scenario [JG98]. Supposed the application has two classes: Car and Order. The DAVs in classes Car and Order consist of three elements and two elements respectively, referring to the number of attributes in each class. In class Car, the execution of method M1 visits three breakpoints: [M1], [A] and [A1]. During runtime at each breakpoint the DAV changes. The DAV at [M1] is [R,W,R] that refers to Read, Write and Read on the attributes: "id", "price to rent" and "quantity on hand" respectively, which correspond to their access mode. At breakpoint [A] the DAV changes to [R,N,R] that refers to Read on "id" and Read on "quantity on hand" ("price to rent" is not applicable i.e. N), and then at breakpoint [A1] the DAV becomes [R,W,N] that refers to Read on "id" and Write on "price to rent" ("quantity on hand" is not applicable).

In the preceding example, before the execution of method `adjust-price`, a lock is obtained on the method and on all attributes to be accessed within the method. Then, during the method execution, the lock changes to become less restrictive, containing only on the accessed attributes. The purpose of changing the lock into finer granularity (i.e. attribute) during the method runtime is to allow more concurrency.

The commutativity relationship table, which is based on the DAV, is automatically generated by the system. Then a user can explicitly set "semantically commute" relationship between DAVs into the commutativity table.

The protocol derived from this approach is as shown in Figure 2.10. Rule 2-1 requires locking on the method to be called before a method execution. Rule 2-2 and Rule 2-3 provide semantic-based concurrency control by adopting the closed nested transaction approach.

Rule 2-1:

A lock is required only for method execution,
and is granted before method execution.
During method execution, the lock changes in accordance
with the breakpoints.

Rule 2-2:

A method cannot terminate until all its children terminate.
When a method *m* terminates,
if *m* has a parent and *m* commits, then
 the lock on *m* is retained by its parent
If *m* has no parent or if *m* has a parent but *m* aborts, then
 the lock is released

Rule 2-3:

A lock is granted if one of the following conditions is satisfied:

1. When no other methods hold or retain a conflicting lock,
 if conflicting locks are held,
 such locks are retained by the ancestors of the requesting method
2. For semantic commutativity,
 if conflicting locks are retained by non-ancestors,
 then when one of the ancestors of the retainer
 not including the retainer itself
 and an ancestor of the requester commute.

Figure 2.10: Semantic-based CC protocol

2.3.3 Attribute locking

In this approach [RAA94], unlike in the preceding approaches ([MWBH93], [JG98]), a lock needs to be acquired only on an object's attribute, whereas it is not necessary to take a lock on a method.

The protocol is as shown in Figure 2.11. Rule 3-1 requires that a lock is acquired on an object's attribute, which we will abbreviate as "attribute" in this description. The "atomic operation" in Rule 3-1 means attribute. Before an attribute is accessed (i.e. for read or write), a lock on the attribute needs to be acquired. A lock request on an attribute should also include the method that is accessing the attribute, and the ancestors of the method, if any. When a lock is granted on an attribute, the system records the locked attribute and the method (and its ancestors if any). When a lock request arrives, the system consults the record and also consults the commutativity relationship table in the object schema, to check for a lock conflict, as well as whether a conflict can be released due to a

semantic commutativity relationship between the methods.

Rule 3-1:

A method execution can execute an atomic operation t
if and only if $\text{lock}(t)$ is requested and is granted

Rule 3-2:

A method execution cannot terminate (i.e. commit or abort)
until all its children have terminated.

When a method execution terminates:

- If it is not top-level and it commits,
its locks are inherited (transferred) to its parent.
- If it is not top-level and it aborts,
its locks are discarded.
- If it is top-level,
its locks are discarded.

Rule 3-3:

A $\text{lock}(t)$ can be granted to a method execution, only if:
no other method execution holds a conflicting lock,
and
for all other non-ancestor methods x'' that retain a conflicting $\text{lock}(x)$,
some element x' (between x'' and x) and some ancestor t' of t commute.

Figure 2.11: Attribute Locking protocol

Rule 3-2 shows that the protocol adopts the closed nested transaction approach. A method is regarded as a sub-transaction within a nested transaction. When a method finishes, the locks are not released but retained by its parent. Recall from the preceding description about nested transactions that lock retention prevents the results in a finished method from being visible to other transactions, because the method may later be aborted when its parent aborts. Also, as in the method locking approach [MWBH93], the retained lock in a closed nested transaction allows a conflict to be identified by a transaction from wherever in a tree the transaction is when the transaction is requesting a lock.

Finally, rule 3-3 of the protocol shows that a lock conflict on an attribute can be released due to method commutativity relationships between ancestors of the attribute.

One issue claimed to have been addressed in this approach [RAA94] is the handling of referentially shared-objects (RSO), which was claimed in the previous study [MWBH93] to have been

left uninvestigated. In RSO, methods of different objects, in different transactions, share an object, resulting in a situation where contention on the shared object causes a check on method commutativity to be made against methods of different objects. It is in contrast with the case where a check for method commutativity is made against a method of the same object. RSO might occur dynamically during transaction execution, so that the idea of statically defining commutativity relations across objects was found to be inflexible and difficult to enforce. This study [RAA94] claimed that RSO was addressed during the execution of methods by determining conflicts and commutativity relationships.

The study provided a proof of correctness (informally) of the approach. It is based on the “semantic serialisability” concept for the nested transaction model [CB89]: an execution of nested transactions is considered correct if it is serialisable as viewed by the top-level transactions. A serial view of top-level transactions will be obtained if it is possible to perform a series of “reductions” and “substitutions” against subtransactions in the trees.

As an example, Figure 2.12 shows two transactions T_1 and T_2 , whose execution time is ordered from left to right. Supposed that a commutativity relationship occurs between $o_5.m()$ and $o_5.m()$ in that the operation on $o_5.a$ is a Write. After T_1 finishes accessing $o_5.a$, the lock on $o_5.a$ is held by method $o_5.m()$. Due to the commutativity relationship between $o_5.m()$ and $o_5.m()$, T_2 can acquire a writelock on $o_5.a$ after method $o_5.m()$ in T_1 finishes. A proof is required to check that these accesses are correct.

The proof should show that the top-level transactions T_1 and T_2 are serialisable. A series of reductions and substitutions are performed as follows. Stage 1 shows an interleaving between two transactions T_1 and T_2 in that $o_3.m()$ interleaves with $o_2.m()$. These two methods have a RSO (referentially shared object) in object o_5 . Firstly, reductions are made and the result is stage 2. Then, because $o_2.()$ and $o_3.()$ are methods of different objects, no conflict occurs and a substitution can be made, resulting in stage 3. Because a commutativity relationship exists between $o_5.m()$ and $o_5.m()$, a substitution can be made, and the result is stage 4. The commutativity relationship releases the conflict between the writelock on $o_5.a$ held by $o_5.m()$ in T_1 and the writelock on $o_5.a$ requested

by T_2 . Afterward, reductions are made in stages 4 and 5, and result is the serial execution of the top-level transactions T_1 and T_2 in stage 6. This proves that the transaction execution is correct.

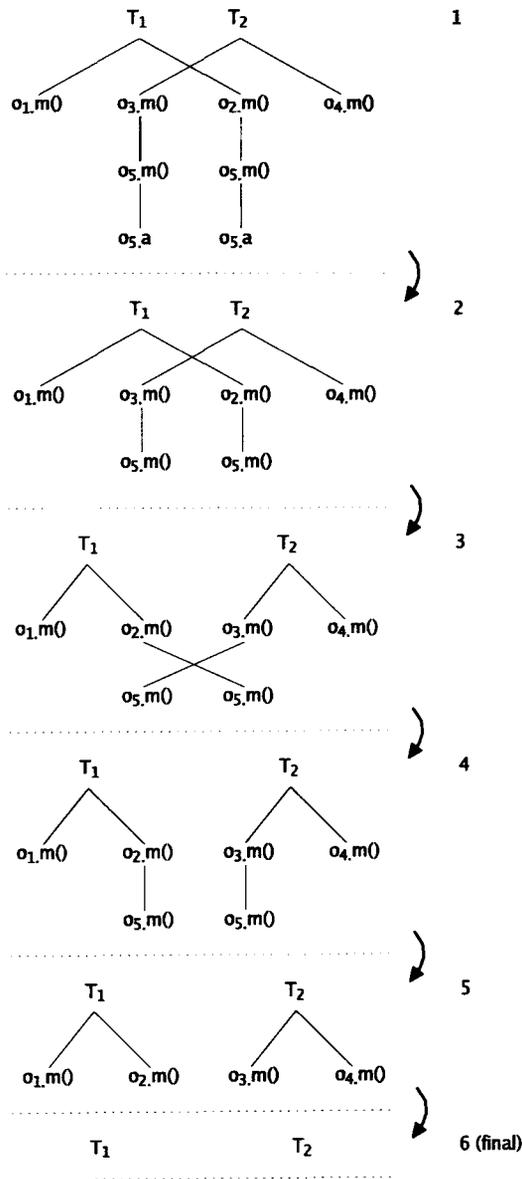


Figure 2.12: An example of correctness check

2.3.4 Notes on the approaches

It can be observed that the difference between this approach [RAA94] and the previous approach [MWBH93] is in terms of lock acquisition. Figure 2.13 illustrates lock acquisition in [RAA94]. Suppose that a transaction include nested methods calls as illustrated in Figure 2.13. Method $o_1.m()$ accesses attribute $o_1.a$ and calls method $o_2.m()$. Then, method $o_2.m()$ accesses attribute $o_2.a$ and executes method $o_3.m()$, and then method $o_3.m()$ accesses attribute $o_3.a$. Before the access on each attribute, i.e. $o_1.a$ in stage 1, $o_2.a$ in stage 2 and $o_2.3$ in stage 3, a lock is requested on each attribute, so that finally in stage 4 all attributes are locked. Information about a lock is associated with information about the method, and its ancestors if any.

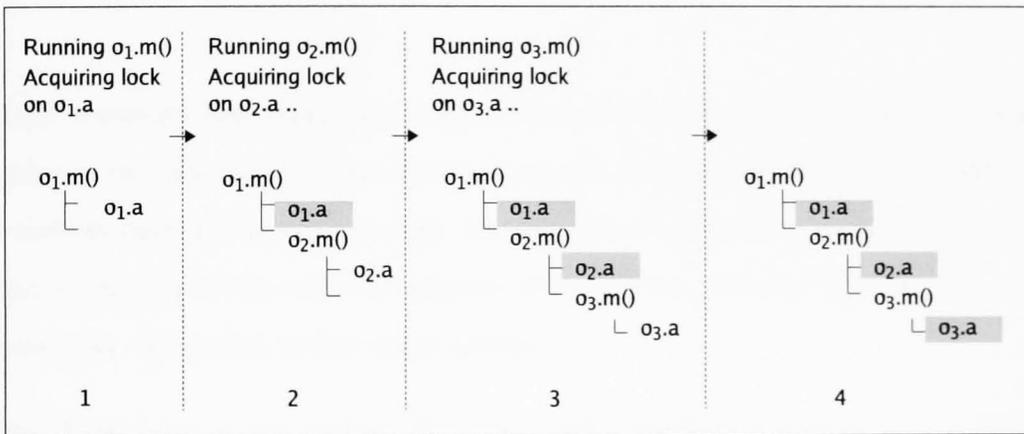


Figure 2.13: Locks acquisition using approach in [RAA94]

By comparison, the protocol in the previous study [MWBH93] requires a lock to be acquired before a method invocation. Thus, before a method is called, a lock is requested for the method and all the attributes to be accessed within the method. Using the same example as in the preceding description, Figure 2.14 illustrates the lock acquisition in this approach. At stage 1, before the execution of method $o_1.m()$, a lock on the method is acquired. When the lock is obtained at stage 2, a lock is taken on attribute $o_1.a$ and method $o_1.m()$. The lock acquisition goes on until the attribute $o_3.a$ is locked, at which point there we also lock on attributes $o_2.a$ and $o_1.a$ as well as their ancestors.

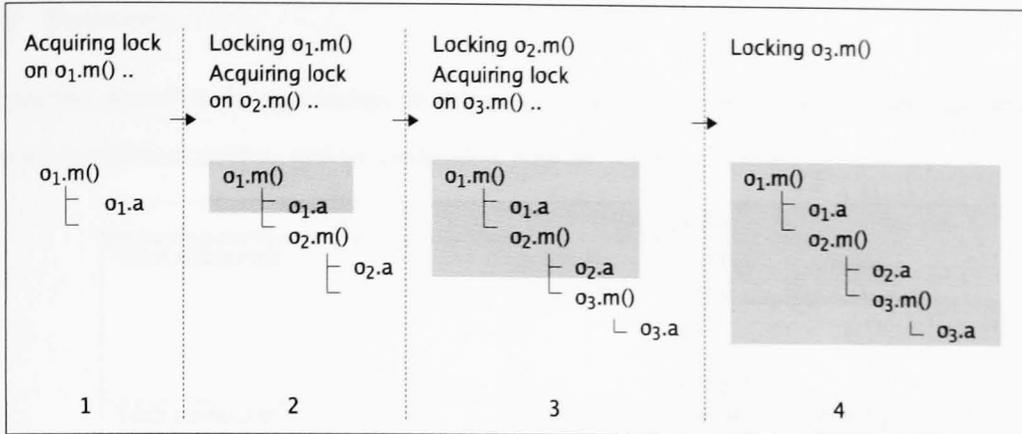


Figure 2.14: Locks acquisition using the approach in [MWBH93]

Thus, essentially both approaches ([MWBH93] and [RAA94]) are similar in terms of the information in the lock record, in that their lock records contain information about attributes and their methods (and their ancestors if any). The lock record will be used to check whether a lock conflict can be released, by using the semantics of the methods consulted from the corresponding commutativity relationships in their object schema.

One disadvantage in this approach [RAA94] is that a deadlock may occur when two transactions are holding readlocks on a shared attribute and then they are simultaneously try to acquire a writelock on the attribute. Supposed the attribute is X , and two transaction $T1$ and $T2$ are already holding a readlock on X . Then both transactions are trying to acquire writelock simultaneously, and a commutativity relationship exists between the method that read X and the method that wrote X in both transactions. So, first $T1$ acquires a writelock on X , and because of the commutativity relationship $T1$ needs to wait until the method in $T2$ has finished. At the same time, similarly $T2$ is acquiring a writelock on X and needs to wait until the method called by $T1$ has finished. Thus, this scenario ends up with $T1$ and $T2$ waiting for each other to finish each other's method. The deadlock in this situation will not occur in the previous approaches (in [MWBH93] and [JG98]), because in the previous approaches a lock needs to be acquired before method execution.

2.3.5 Summary

This section describes the approaches to semantic-based concurrency control. There are three approaches in different studies, and we summarise how they differ in Table 2.1.

	[MWBH93]	[JG98]	[RAA94]
LOCK ACQUISITION	method execution	method execution, attribute operation	attribute access
LOCK GRANULARITY	method, attribute	method, change to attribute	attribute
LOCK RECORD	method and its ancestor(s), attribute	direct access vectors (DAV)	attribute and its ancestor(s)
COMMUTATIVITY RELATIONSHIP	explicitly defined	explicitly and automatically defined	explicitly defined
REFERENTIALLY SHARED OBJECT	unexplored	explored	explored
CORRECTNESS PROOF	not detailed	not detailed	detailed

Table 2.1: Aspects of semantic-based concurrency control protocols

The studies, however, did not address the database environment on which they are implemented, such as either in data-shipping or query-shipping. The impression is that it assumes a centralised database system (i.e. query-shipping) as the environment. In the next section we will describe the existing studies of client cache consistency protocols, which are in data-shipping environments. We will include description on how concurrency control is handled in the protocols, in order to identify how semantic-based concurrency control can be used in the protocols.

2.4 Existing client cache consistency protocols

The preceding section described semantic-based concurrency control. This section will review the existing client cache consistency protocols. We will describe where read-write and write-write conflicts are addressed in order to identify where semantic-based concurrency control can be incorporated in client cache consistency protocols. In addition, we will describe issues that affected the performance of the protocols.

As mentioned in Section 1.1, a number of client cache consistency protocols have been proposed for data-shipping object-oriented database system. Based on how the server checks the consistency of clients' caches, the protocols have been categorised into two families: *avoidance-based* and *detection-based* [Fra96].

- In avoidance-based protocols, the server records which copies of objects are cached at which clients. When a client wishes to update an object, it sends a write intention notification message to the server. If the server identifies the set of database objects updated in the transaction that are cached at other clients, the server contacts the other clients to tell them to remove the stale copies. Before continuing with the transactions, the server waits until this has been done. Thus, stale copies of objects are avoided, hence the name is "avoidance-based".
- In detection-based protocols, the server records which stale copies of objects are cached at which clients. When a client validates a transaction at commit time, the server detects whether the objects being validated by the transaction are stale, and if so the server will reject the transaction. Thus, stale copies of objects are allowed to be cached by clients while a transaction runs, but this is later detected by the server at validation time, and the transaction is aborted.

Based on how client validates a transaction to the server, each of these families is further categorised into pessimistic, optimistic and asynchronous:

- In the pessimistic scheme a client validates its local transactions on every update on an object. After the client sends a validation message to the server, the client waits for the result from the server before continuing the transaction.

- In the optimistic scheme a client validates its local transactions only at the end of the transaction. During the transaction the client optimistically reads and writes objects locally.
- In the asynchronous scheme a client validates its local transactions at every object update, but unlike in pessimistic scheme, after the client sends a validation message to the server the client does not wait, but continues with the transaction. The result of whether or not the validations are successful is given at the end of the transaction.

We now discuss the protocols in more detail.

2.4.1 Avoidance-based protocols

Avoidance-based protocols prevent a client cache from caching stale (out-of-date) database items[‡]. This is illustrated in Figure 2.15.

When a client intends to update an item in its local cache, it notifies the server and waits for the response. When the server receives the notification, if other remote clients have also cached a copy of the item, the server sends cache consistency messages to those clients (point 1a). This cache consistency message is to prevent a client from caching a stale (out-of-date) copy of the item. The server will only allow clients to proceed with the write once it is sure that no other clients are caching a stale copy of the item. Therefore, before the server approves the client's intention to write, the server needs to receive an acknowledgement as to whether the cache consistency actions from all the other clients have been satisfied. While the server waits for responses from the other clients, it acquires an exclusive lock, i.e. a writelock, on the item in order to prevent interference with it. If a cache consistency action cannot be satisfied, the server aborts the transaction (point 2b). Otherwise, the server allows the client to proceed with the write, and the client locally acquires a writelock on the item. Thus, in an avoidance-based scheme, when a client holds a local writelock on an item, no other clients can have a stale copy of the item.

As mentioned above, the server locks the item while waiting for cache consistency action responses from clients. This is a short-term lock, to prevent the item from being read or written during

[‡]“Item” here can be in any granularity. Page is generally the granularity in the existing studies.

the waiting period. If during this period a write request for the item arrives from another transaction, the transaction will be aborted. Also if during this period a request to read the item arrives from a client, the client must wait until the lock has been released by the server. The server will give the clients the most recent version of the item.

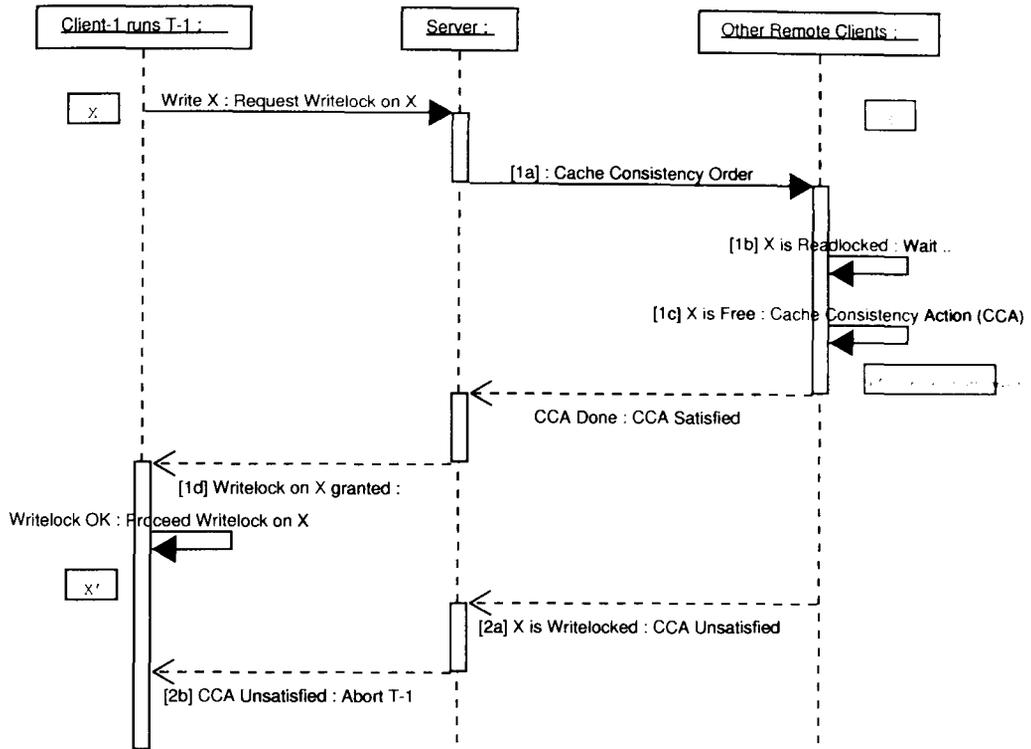


Figure 2.15: The avoidance-based scheme

When a cache consistency message is received by a client, it must perform a cache consistency action. This is either an invalidation (removal) of the item from the cache, or a propagation (replacement) of the item with the updated value. However firstly the client checks whether or not the item is locked. If the item is not locked, the cache consistency action can be performed (point 1c). If the item is currently readlocked, the client defers the cache consistency action until the readlock is released (point 1b). If the item is currently writelocked, the client informs the server that the cache consistency action cannot be satisfied (point 2a). When a cache consistency action can be

performed, it is acknowledged in a message from the client to the server.

Next we review how read-write and write-write conflicts are resolved in avoidance-based protocols. This will be useful to identify how method semantics can be used in resolving read-write and write-write conflicts.

- Read-write conflicts

A read-write conflict is detected and resolved at the client and at the server. A client locally records readlocks and writelocks on accessed items. A client checks for a read-write conflict on an item when it receives a cache consistency message from the server, which means that another client intends to update the item. Recall that when client A is updating an item it sends a write intention notification to the server, and if the item is also cached at client B a cache consistency message is sent to client B by the server. A read-write conflict occurs when client B has a readlock on the item when it receives the cache consistency message. The conflict is handled by Client B by deferring the cache consistency action until the item is unlocked that is after client B has ended his transaction.

A read-write conflict is also detected at the server, that is when the server handles a read request for an item from a client. When an item is requested by a client, but has been updated by another uncommitted transaction, the server will postpone sending the item until the item is unlocked i.e. after the updating transaction has ended. Thus, the server takes exclusive locks (similar to writelocks) on items that have been updated by uncommitted transactions. However, the server does not record readlocks on items. It is the client that holds readlocks on items it receives from the server.

- Write-write conflicts

A write-write conflict is detected and resolved at the client. A write-write conflict occurs when a cache consistency message on an item arrives at a client that has a writelock on the item. In such cases, the conflict is resolved by the client informing the server that the cache consistency action cannot be performed.

However, a write-write conflict can also be detected at the server. Consider a scenario where at the server there is a queue of two requests from client 1 and from client 2. The first request is a write intention notification on item X from client 1 and the next request is a write intention notification on the same item X from client 2. The server firstly serves the request from client 1. Because X is also cached by client 2, the server sends a cache consistency message to client 2 and acquires a writelock on X on behalf of client 1. While the server is waiting for a response from client 2, the server continues with the next request from the queue, that is the request from client 2, which is also a write intention notification on X. At this point the server detects a write-write conflicts on X, because client 2 intends to write X, while X is writelocked on behalf of client 1. Thus, the write-write conflict will be detected at the server when a client has sent a write intention notification on an item before a cache consistency message on the item is received.

Depending on how lock requests are made by a client, the existing avoidance-based protocols are further categorised into: *synchronous*, *deferred* and *asynchronous* [Fra96]

- Synchronous

In the synchronous protocol, every time a client intends to write an item, the client sends a write intention notification to the server and waits for a response. The client can proceed with its transaction once the server has registered write intention on the item on behalf of the client. Callback Locking Protocol (CBL) [Fra96] is a protocol that falls into this synchronous category. A *callback* message is a cache consistency message, sent by server to the client to perform a cache consistency action. Recall that when the server receives a write intention on an item from a client, if other remote clients are caching a copy of the item, the server will send a callback message to each of the other remote clients, and will allow the write to proceed once all callback messages have been satisfied.

- Deferred

In the deferred category, a client sends a write intention notification only at the end of a trans-

action. During a transaction a client locally performs updates without notifying the server. At commit time, the client notifies the server of all updated items in a commit message, and then waits for a response. The server, having received the notification, sends a cache consistency message to other remote clients that are caching the copies of the items, and will commit the transaction if all cache consistency actions have been satisfied. Sending a write intention notification only at the end of a transaction reduces the message overhead between clients and servers, and it allows fewer write intention notification messages from a clients per transaction, and therefore fewer cache consistency messages from the server. A protocol in this category is Optimistic Two-Phase Locking Protocol (O2PL) and its variants: O2PL by invalidation (O2PL-i) and O2PL by propagation (O2PL-p) [Fra96]. The difference between O2PL-i and O2PL-p is in terms of a client's cache consistency action. In O2PL-i, a client's cache consistency action is to remove the item from the client's local cache, whereas in O2PL-p a client's cache consistency action is to replace the item in the client's local cache with the updated version sent by the server.

- Asynchronous

In the asynchronous protocol a write intention notification is sent by the client on each item update (i.e. as with the synchronous scheme). However, in the asynchronous protocol, after sending a write intention notification a client does not wait for a response from the server, but instead continues with its transaction. At commit time, the client sends a commit message and waits for a response from the server. The server, during a transaction, receives all the write intention notifications from clients, and sends cache consistency messages to other clients. However, upon a successful consistency action, the server does not send a response to the client. Only if a transaction must be aborted would the server sends an abort message to the client.

The advantage of the asynchronous scheme is to reduce a client's blocking time, and frequent validations of a transaction between the client and server. In the synchronous scheme, a

client validates the transaction frequently and the client experiences some blocking time on every validation, whereas in the deferred scheme the client does not experience any blocking time, except at commit time. The asynchronous scheme is a compromise between these two protocols by keeping blocking time low but requiring frequent validations of the transaction. Asynchronous Avoidance-based Concurrency Control (AACC) [OVU98] is an avoidance-based protocol which is asynchronous. The protocol also makes an effort to reduce the number of messages sent by a client, by tagging pages that reside only at one client as "private-read", so that if a client knows that it is the only client holding the copy of a page, it does not send a write intention notification to the server.

2.4.2 Detection-based protocols

In detection-based protocols, consistency is maintained by having the server reject transactions that attempt to validate stale items. A client can cache stale items in its local cache, and when a client updates an item it validates the item with the server. If the server detects that the item is stale then the transaction is aborted. Thus, the server must maintain records that allows it to detect whether or not an item being validated is stale. The record of such items is kept until the corresponding transaction has ended. Figure 2.16 illustrates the detection-based scheme.

To check for consistency, the server consults a record of the version number of the items. For example, in the Caching Two Phase Locking (C2PL) protocol [Fra96], each item is associated with a sequence number. The server maintains a record of the most recent sequence number for every item, so that an item with sequence number lower than that in the server's record is regarded as stale. In the Adaptive Optimistic Cache Consistency (AOCC) protocol [ALM95], the server maintains a record of stale items in every client named "Invalid Set". When the validation of an item is successful, all other copies of the item in the other clients are added into the Invalid Set, so any further validation on an item registered in the Invalid Set is rejected (because the item is stale).

Therefore, the check for lock conflict in detection based protocols is made at the server rather than at the client. We now consider conflicts in more detail:

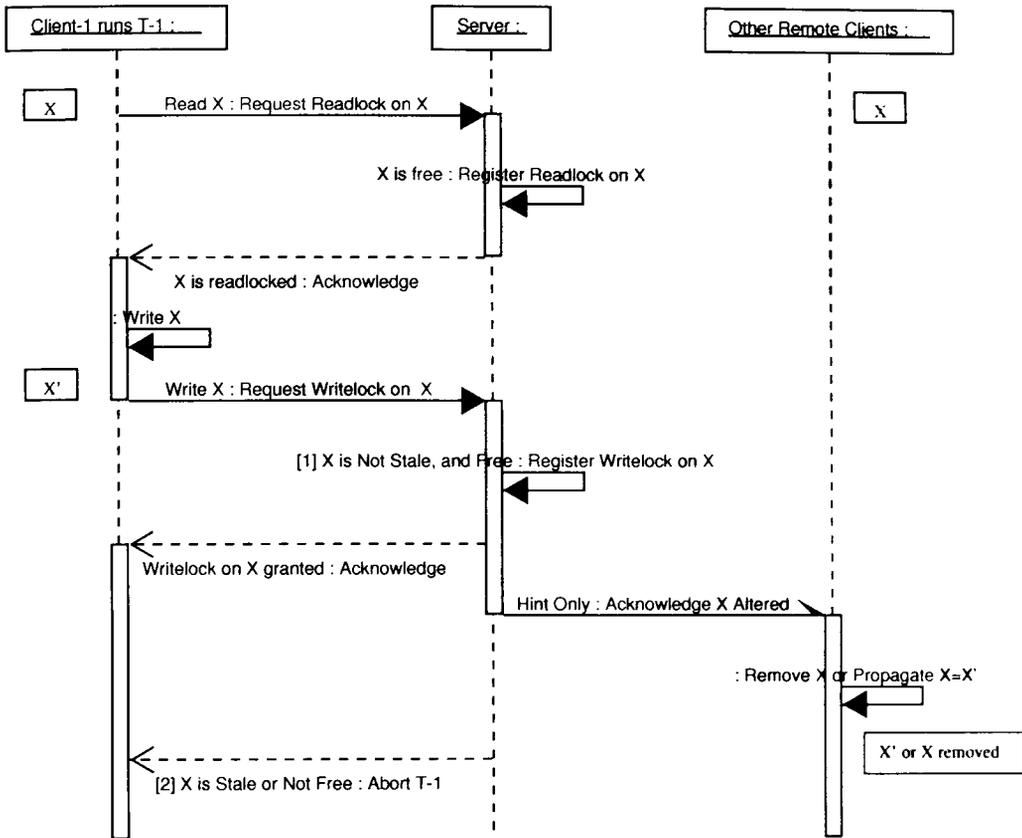


Figure 2.16: The detection-based scheme

- Read-Write conflict

A read-write conflict is detected when the server is processing a fetch request for an item from a client. When a client intends to read an item that is not locally cached, it requests the item from the server. If the item is being recorded on behalf of an unfinished transaction, a read-write conflict is detected and the server defers sending the item until the transaction has ended. Otherwise the server sends the item to the requesting client. When the requesting client receives the item, it acquires a readlock on it.

- Write-Write conflict

A write-write conflict is detected at the server when it receives a validation request from a client. An item that has been successfully validated on behalf of a transaction is recorded at the server, and the record is kept until the transaction ends. During the transaction period, if another transaction validates an item currently in the record, a write-write conflict occurs. The server resolves this conflict by aborting the other transaction to preserve the correctness of transaction execution.

The detection-based protocols are divided into *synchronous*, *deferred* and *asynchronous* in accordance with how client validates accesses to items.

- Synchronous

In the synchronous scheme, a client validates on every update on an item. After sending a validation message the client waits for the result.

The caching Two-Phase Locking (C2PL) protocol [Fra96] is a synchronous protocol. In C2PL, a page is tagged with a sequence number. The server maintains a record of the most recent sequence number on every page that is cached by a client. The successful validation of a page updates the page's sequence number. Therefore a page, which is subsequently validated by another transaction, that has sequence number lower than that recorded at the server, is regarded as stale.

- Deferred

In the deferred scheme, in order to reduce the message overhead between the client and server, a client does not validate items with the server during a transaction. It is only at the end of the transaction at commit time that a client validates all items updated in the transaction. The server checks whether any item being validated is stale. If one of the items is stale, the transaction is aborted. Adaptive Optimistic Concurrency Control (AOCC) [ALM95] is a deferred protocol. In this protocol, "adaptive" refers to the lock granularity that can change from page level to object level whenever false sharing is detected (False sharing occurs when one transaction is trying to lock an object, but the page containing of the object is being

locked by another transaction because the other transaction is accessing another object in that page). In AOCC, the server maintains a record named “Invalid Set” that contains stale pages currently cached by clients. When a transaction validates a page that is in the Invalid Set, the transaction is aborted.

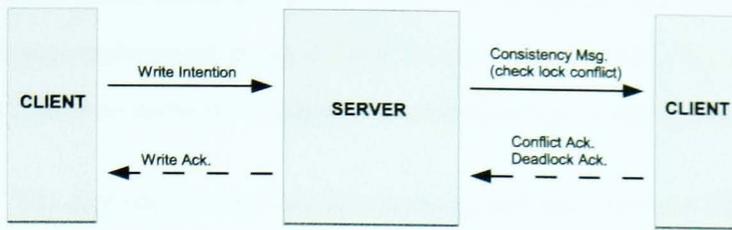
- **Asynchronous**

In an asynchronous scheme, a client sends a validation message on every intention to update an item but the client does not wait for the result from the server, but instead it continues the transaction locally. At commit time, a client sends a commit message and waits for the result of all validations previously sent. At the server, when a validation from a client is received, if there is no conflict the server does not acknowledge the successful validation. But if a conflict is detected then the server will send a message to the client to abort the transaction. If there is no conflict throughout a transaction, the server sends an acknowledgement to the client at the end of the transaction. The asynchronous protocol results in low blocking time at the client but frequent communications between the client and server.

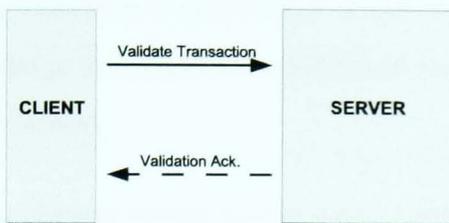
A summary of where lock conflict is detected in avoidance-based and detection-based schemes is shown in Figure 2.17. It can be seen from the figure that in avoidance-based protocols the performing of conflict checks is passed by the server to clients, because concurrency control and deadlock detection is performed not only at the server but also at the clients. In detection-based protocols, concurrency control is performed only at the server.

2.4.3 Performance Issues

Here, we will describe issues with the performance of existing client cache consistency protocols. In the literature these protocols have had their performance investigated using simulation, as this allows the various parameters to be changed without having to modify an actual system. Firstly, we will describe the model generally used in measuring the performance of the existing client cache consistency protocols. Then we will summarise the issues that influence their performance. The



(a) In avoidance-based scheme



(b) In detection-based scheme

Figure 2.17: Concurrency control operations in Avoidance and Detection based schemes

aim of the description is to understand the characteristics of the protocol.

The Structure of models

The architecture modeled is client-server. The modeled system consists of the following components:

- **CPU (central processing unit).** CPUs processes machine instructions at both the clients and the server, with processing speed measured in millions of instructions per second (MIPS).
- **Disk.** Disk is used for stable database storage. Disk was present only at the server. The cost to read from, or write to, disk was represented as an average disk access cost[§].
- **Cache.** Cache here is an area of memory used to temporarily store database items fetched from disk. Cache is used because it has much lower access cost than does disk. A cache

[§]The actual components of disk access cost includes: seek time, settle time and latency

is present at the client and at the server. It has a fixed capacity measured by the number of pages. The page replacement policy is Least Recently Used (LRU). The cost to read or write a database item from cache is assumed to be a fixed number of instructions.

- **Network.** The network is a medium for conveying messages between the client and server. The cost of the network consist of a *fixed* cost and a *variable* cost. The fixed cost is the processing cost of the CPU and the network controller at the client and at the server. The value is assumed to be a fixed number of CPU instructions. The variable cost is the cost per byte of message transferred and is calculated based on the network speed in millions of bits per second (Mbps).

A database is physical storage holding a set of database pages¹. It is modeled as a collection of page identifiers. During a transaction, a client accesses the database by reading or writing page identifiers representing the page in the database.

In a transaction, each client has interest in accessing a particular set of database pages or objects. In addition, a particular set of pages or objects in the database might be shared by a set of clients. The pattern of the access to the database is dictated by the workload, which models data locality and sharing. The following describes in detail how the workload is modeled.

Pages in the database were separated into regions [ALM95] [OVU98], as illustrated in Figure 2.18.

- **Private region:** a region that is private to a client. It contains pages that are accessed most of the time by a particular client.
- **Shared region:** a region containing pages that are accessed by all the clients (i.e. they are shared by all the clients).
- **Other region:** a region outside the Private and Shared regions.

¹Note that the granularity of a database item can be page or object. In our description we use page.



Figure 2.18: Data locality

The above regions are used in some studies [ALM95] [OVU98]. In contrast, in another study [Fra96] there were simply *hot* and *cold* regions, without a Shared region. The Hot region was the same as The Private region, and the Cold region covered the Shared and Other regions.

Thus, each client was allocated a Hot region in the database, and each client was also allocated a Cold region outside the Hot region. In addition, the Hot region belonging to a client overlapped with the Cold region belonging to other clients.

How frequently each region is accessed during a transaction was determined by a probability value. In addition, how likely a transaction performs a Write on pages in a particular region was determined by a probability value. These settings are encapsulated in a *workload*. Thus, a workload is identified by an access probability value for each region, and a write probability value for each region.

Some workloads were then defined based on the access probability and the write probability for each region. The workload that was claimed to represent general database applications is HotCold [Fra96], in which Hot and Cold regions were defined. The HotCold workload was later altered to become Sh/HotCold [ALM95] [OVU98] to include a Shared region. The values of the access probability and the write probability for each region is shown in Table 2.2.

Study	General workload	P(Access Hot Region)	P(Access Cold Region)	P(Write)
[Fra96]	Hotcold	80%	20%	20%
[ALM95]	Sh/Hotcold	80%	10% on <i>Shared</i> , 10% on <i>Other</i>	5%
[OVU98]	Sh/Hotcold	80%	10% on <i>Shared</i> , 10% on <i>Other</i>	varied

Table 2.2: Probability values in HotCold and Sh/HotCold workloads

From the preceding description, it can be seen that workloads model data locality and data sharing by defining probability values for accessing pages and writing pages in the Private, Shared and Other regions in the database.

Aspects that Influence Performance

The performance was measured for variable levels of data contention. One study [Fra96] use the number of clients as the variable of data contention, with a fixed Write probability. Other studies [OVU98] [ALM95] use the write probability values as the variable of data contention, with a fixed number of clients.

The performance of the existing client cache consistency protocols are also affected by the database size, measured as the number of pages in the database. If the number of pages in the database is reduced, with the same number of clients and the same transaction length, the data contention rises. For example, if there are a smaller number of pages in the Shared region, the probability of sharing pages is higher. In all the existing studies, the database size was fixed.

We now describe the results of previous investigations into the performance of the following client cache consistency protocols:

- Caching Two-phase Locking (C2PL) [Fra96]
- Callback locking protocols (CBL) [Fra96]
- Optimistic Two-phase Locking (O2PL) [Fra96]
- Adaptive Optimistic Concurrency Control (AOCC) [ALM95]
- Asynchronous Avoidance-based Consistency Protocol (AACC) [OVU98]

No.	Avoidance-based					Detection-based			Ref.
	Sync.	Def.			Async.	Sync.	Def.	Async.	
	CBL	O2PL-i	O2PL-p	O2PL-ND	AACC	C2PL	AOCC	NWL	
1		Δ	Δ	Δ		Δ			[Fra96]
2	×			×		×			[Fra96]
3	○						○		[ALM95]
4	▽				▽		▽		[OVU98]

Table 2.3: The protocols compared in the existing studies

The performance comparisons will not be based on the quantitative results, but on whether one protocol performs better or worse than the others, and the identification of parameters that affect the performance. This is because the protocols compared in one study were different from those in the other studies, as seen in Table 2.3, and each study used different parameter values in the simulations.

First, we describe the following terminology used in the performance analysis:

- **Throughput** is the number of committed transactions per unit of time. It is the main metric that indicates whether or not a protocol is better than the others.
- **Average Response Time** is the average time measured from the start of a transaction until the commit of the transaction.
- **Abort rate** is the average number of aborts experienced by a transaction measured from the start of the transaction until the commit of the transaction.

In general, the relative performance of the above protocols under HotCold or Sh/HotCold workload were as follows:

- O2PL outperforms CBL, and CBL outperforms C2PL [Fra96]. The version of O2PL referred to here is O2PL-i.
- AOCC outperforms CBL [ALM95]
- AACC outperforms AOCC [OVU98]

The following are the description about aspects that influenced the performance:

- Disk utilisation

In the previous studies, disk is only at the server (as mentioned in the preceding description). Disk accesses occur when the server receives a request for a page from a client and the page is currently not in the server's cache. When receiving a request, the server reads the page from disk and writes it into the cache, before sending the page to the client. The cost of accessing the disk is much higher than any other costs. It takes milliseconds to access the disk, but only nanoseconds to access the cache.

Disk utilisation is influenced by the size of the client caches [Fra96]. In the Optimistic Two Phase Locking (O2PL) and Callback Locking (CBL) protocols, higher disk utilisation occurs when clients have larger caches. The reason is described as follows. In the O2PL and CBL protocols, the server receives all dirty pages (i.e. pages modified during a transaction) at the end of the transaction. If clients have had larger caches, more dirty pages are received by the server. More dirty pages at the server causes a higher number of pages to be replaced in the server's cache (with the Least Recently Used page replacement policy) because the server's cache has a limited size, and dirty pages that were rejected from the server's cache needed to be written to the disk. Thus with larger clients' caches, the disk utilisation becomes higher.

By comparison, in Caching Two Phase Locking (C2PL) which is a detection-based protocol, a validation was not associated with dirty pages but instead with the sequence number of the

updated page. Consequently C2PL protocol did not take up the server's cache's space, and the disk utilisation in C2PL was less than that in O2PL and CBL when the number of clients was increased.

With a fast network, the influence of disk utilisation could become even greater, and the performance of O2PL and CBL could be disk bound (i.e. almost entirely dictated by disk).

Another study [OVU98] that compared the asynchronous avoidance-based protocol (AACC), the optimistic detection-based protocol (AOCC) and the pessimistic avoidance-based protocol (CBL), showed that the disk utilisation in the AOCC protocol (optimistic, detection-based) was greater than that in CBL (callback) and AACC (asynchronous; avoidance-based) protocols. This was due to the fact that in AOCC transactions ran more quickly than in CBL and AACC. In AOCC transactions ran optimistically without blocking, whereas in CBL and AACC transactions could be ended when a conflict was found.

Thus, in summary disk utilisation will be higher when the client's cache is larger and when transactions run more quickly, which can be due to a fast network or the optimistic behaviour of the protocol. As a consequence, because a disk access is the highest cost component of the performance, the system can become disk bound.

- Message overhead

Message overhead is also a factor that influences the performance. Message overhead usually refers to the *frequency* as well as the volume, of messages transferred between the clients and server.

A study [Fra96] showed that the large reduction of the number of messages traveling from the clients to the server made O2PL (optimistic, avoidance-based) outperform C2PL (pessimistic, detection-based) under the HotCold workload.

The study also showed the reduction in message overhead due to sending only writelock requests, rather than both readlock and writelock requests made O2PL (optimistic, avoidance-

based) and CBL (pessimistic, avoidance-based) outperformed C2PL (pessimistic, detection-based) in low condition workloads.

The study [Fra96] also showed that the message overhead was related to the size of the client's caches. When the clients' caches were larger, O2PL (which is avoidance-based) caused the server to send more consistency messages to the clients. By contrast, in C2PL (which is detection-based) the client cache size did not significantly affect the message overhead.

In the same study, the message overhead also influenced the scalability measured in terms of the number of clients. Under a high data contention workload (HICON), the reduction in messages from server to clients allowed C2PL (pessimistic, detection-based) to be more scalable than O2PL (optimistic; avoidance-based) and CBL (callback). Being avoidance-based, O2PL and CBL required the server to send consistency messages to clients, whereas being detection-based, C2PL did not need consistency messages to be sent. Consequently, when the number of clients increased, in the avoidance-based protocols the server performance suffered from having to send more consistency messages to the clients, which caused their performance to drop. In comparison, the performance of C2PL remained stable with increasing number of clients.

In O2PL-p protocol (optimistic, avoidance-based by propagation) the performance drop was even more significant, because in O2PL-p consistency messages also carried the actual pages to be propagated to the clients, and so O2PL-p generated larger message volumes than O2PL-i (avoidance-based by invalidation). As a consequence, the performance of O2PL-p was even below the performance of C2PL (pessimistic, detection-based) for a high number of clients.

Again, with regards to the scalability, a study [ALM95] showed that when the number of clients increased, the increase in the number of messages in AOCC (optimistic, detection-based) was not as large as that in CBL (pessimistic, avoidance-based). This is because CBL is an avoidance-based protocol that requires the server to send consistency messages to the clients, whereas AOCC does not require the server to send consistency messages. Thus, with

regards to increasing the number of clients, the study showed that AOCC scaled better than CBL.

The influence of message overhead on performance was also investigated for asynchronous protocols [OVU98]. The study showed that under the HotCold workload, AACC (asynchronous, avoidance-based) outperformed CBL (pessimistic, avoidance-based) one of the reasons for this was that the number of messages transferred in AACC is lower than those in CBL. This is due to the behaviour of AACC that allows a client not to wait for the result of every validation, but instead to continue until commit time.

Thus in summary, message overhead is an important factor that influences the performance. Studies have shown that message overhead could be incurred by the behaviour of the protocol (eg. avoidance-based, detection-based, asynchronous), by the size of client cache and by the workload.

- Cache effectiveness

Cache effectiveness is also another important factor that influences the performance. The study in [Fra96] showed that, due to a more effective use of the client cache, O2PL (optimistic, avoidance-based) outperformed C2PL (pessimistic, detection-based) under the HotCold workload, despite the fact that O2PL was disk bound (as described in the disk utilisation section above). In C2PL, a client's cache can hold stale pages, whereas in O2PL it contains only non-stale pages (recall that avoidance-based protocols do not allow stale items to reside in a client's cache). Consequently, in O2PL a client's cache could be filled with more pages that could be used in the future.

Again, O2PL-i (optimistic, avoidance-based by invalidation) outperformed O2PL-p (optimistic, avoidance-based by propagation) under the HotCold workload because a client's cache in O2PL-i had more non-stale pages than that in O2PL-p. In O2PL-p, client cache consistency is achieved by propagating (replacing) stale pages in a client's cache with updated pages, whereas in the O2PL-i it is achieved by invalidating (removing) stale pages from client's

cache. Therefore, in O2PL-p if updated pages are not used in the future by the client, the client's cache can be filled with pages that are not required so wasting the cost of propagation. When the cache size is limited, the LRU page replacement policy for the cache in O2PL-p might cause useful pages to be ejected from the cache, and in turn increased the disk utilisation in getting the useful pages back from the disk. In contrast, in O2PL-i, client cache consistency is achieved by removing stale pages from the the client's cache, and as a result O2PL-i leaving more space for useful pages in the client's cache under the HotCold workload. However, under the FEED workload, which is when one client acts as data generator (writer) whereas the other clients are the readers of the data, O2PL-p made effective use of the client cache, outperforming O2PL-i [Fra96]. Because no conflict occurred among the readers, a reader client with a large cache could hold all its hot pages, and any propagation of the pages into the client cache on behalf of the writer client is not wasted.

The preceding description shows that client cache effectiveness is an important influence on performance. Cache effectiveness is determined by the behaviour of the protocol (whether avoidance-based or detection-based, and whether the consistency action is by invalidation or by propagation) and by the workload.

- CPU cost. CPU cost was also an influence because it is a component of the message overhead and the disk access cost.

For example, the study [Fra96] showed that C2PL (pessimistic, detection-based) became CPU bound because the protocol required validation not only on a write access to a page but also on a read access.

Another study [OVU98] also showed the influence of CPU speed on performance when comparing the CBL (pessimistic, avoidance-based) and AOCC (optimistic, detection-based) protocols). The study showed that CBL outperforms AOCC under the HotCold workload, but it contradicted another study [ALM95] in that AOCC outperformed CBL under the same workload. This was because the CPU used in [OVU98] is faster than the CPU used in [ALM95].

An increase in CPU speed reduced the blocking time ^{||} of a validation in CBL (pessimistic, avoidance-based), because when a transaction needs to wait because of a read-write conflict detected at the server, a faster CPU at the server allows shorter blocking time in resolving the conflict. Also, faster CPUs at the clients reduces the message processing time in CBL. Thus, with faster CPUs, the shorter blocking time and the reduced message processing time allow CBL (pessimistic, avoidance-based) to outperform AOCC (optimistic, detection-based) under this workload.

Aspects	Relative performance				Reference	
1. Disk I/O - Workload: HotCold * Small client cache, slow networks * Large client cache, fast networks high no of clients - Fast CPU - Slow GPU	high				low	[Fra96] [Fra96] [OVU98] [ALM95]
2. Message Overhead - Workload: HotCold - Workload: Private - Workload: HiCon, Uniform	high				low	[Fra96] [OVU98] [ALM95] [Fra96] [Fra96]
3. Cache Effectiveness - Workload: HotCold - Workload: Feed	low				high	[Fra96] [Fra96]
4. CPU Requirement - Workload: HotCold	high				low	[OVU98]

Table 2.4: The relative performance of the previously studied protocols

^{||}The blocking time is the time needed by a client since sending a validation message until receiving the result of the validation.

A summary of the above influencing factors and their effect on performance can be seen in Table 2.4.

- Workloads factor

When a transaction restarts after it is aborted, it can access pages that are the same or different from the previous transaction. When a transaction restarts by accessing the same pages as in the previous transaction, O2PL (optimistic, avoidance-based) outperforms C2PL (pessimistic, detection-based) although O2PL incurs a higher abort rate than C2PL. This is due to the fact that the avoidance-based O2PL could have better cache effectiveness than C2PL which is detection-based **, and so in O2PL the client's cache contains more useful pages than in C2PL. This leads to higher cache hit rate when the restarted client accesses the same pages as in the previous transaction.

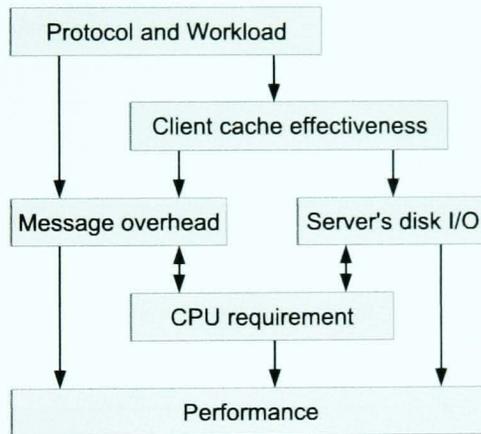


Figure 2.19: Factors that influence the performance

The preceding description has tried to identify the key factors that influence the overall performance, it can be seen that one aspect influenced the others. As a summary, Figure 2.19 shows dependencies among the factors and their influence on the performance.

**Mentioned in the preceding description about cache effectiveness

2.5 Summary

In this chapter we reviewed the advantages of object-oriented databases (OODB), and one of the advantages is that OODB supports method call. Then we reviewed the studies about semantic-based concurrency control that exploited the semantics of methods in order to enhance concurrency. Some background about nested transactions that are relevant to the semantic-based concurrency control studies was explained. Finally we reviewed the existing client cache consistency protocols and identified aspects that influence the performance of the protocols. These are important for the design of our protocols, which will be described next.

Chapter 3

Protocol Design

This chapter describes the design of a new client cache consistency protocol that exploits method semantics. Firstly we detail the requirements and define the criteria, based on our review of semantics-based concurrency control and client cache consistency protocols (Chapter 2). Afterward we will describe a new approach that allows semantic-based concurrency control to be exploited in a client cache consistency protocol. Finally we will describe the implementation of the protocols.

3.1 The requirements

Recall that the requirement is to allow semantic-based concurrency control, which can exploit method semantics for concurrency control, in client cache consistency protocols. It is important to note that the focus is on the concurrency control, therefore firstly we need to identify where lock conflicts are detected in existing client cache consistency protocols.

In avoidance-based protocols, the following are where lock conflicts on database item are detected. Here, a database item can be of any granularity, such as page, object or attribute.

1. At the server, when the server receives a request for a database item from a client.
2. At the server, when the server receives a request to validate a transaction from a client.

3. At a client, when it receives a consistency message from the server.

First, we investigate point 1 in order to identify how a lock conflict is detected at the server when the server receives a request for a database item from a client. This occurs because the client wants to readlock an item that is not in its cache. When a request for a database item arrives at the server, if the database item is already writelocked at the server, then *read-write* conflict occurs and the server will block the request until the item is unlocked. However if the database item is not writelocked at the server then it can be sent to the client. A writelock is held at the server when it is sending consistency messages to clients and is waiting for responses regarding the result of consistency actions. The writelock is kept by the server until the client ends its transaction (commits or aborts). For example in the Optimistic Two Phase Locking (O2PL) protocol (Optimistic, Avoidance-based) [Fra96], the writelock will be kept by the server only during commit time until the server receives responses regarding the consistency messages from all the clients. By comparison in Callback Locking protocol (Pessimistic, Avoidance-based) [Fra96], the writelock will be kept by the server for the duration of the transaction.

Next, we describe point 2, in which lock conflict is detected at the server when the server receives from a client a request to validate a transaction. In avoidance-based schemes, a client validates a transaction when it is going to writelock a database item. Because it is the case that when a client gets a writelock on a database item no other clients have a stale copy of the item, the client sends a write intention message to the server. When the server receives a write intention on a database item, if the database item is being writelocked at the server (for the same reason as in the above description), then a *write-write* conflict occurs and the write intention cannot be granted. However if the database item is not writelocked at the server then the write intention can be granted.

Finally, in point 3, a lock conflict is at client when it receives a consistency message from the server. When a client receives a consistency message from the server, the client is asked by the server to invalidate (remove) or propagate (update) the database item from the client's cache. Recall that the server sends a consistency message to a client because the database item is intended to be writelocked by another transaction, and therefore a consistency message is equivalent to a

writelock request that conflicts with a readlock or writelock. Consequently when a client receives a consistency message, if the item is being readlocked by the client then a *read-write* conflict occurs, which leads the client to block the consistency action until the readlock is released. If the item is being writelocked by the client then a *write-write* conflict occurs, and this leads the client to rejecting the consistency action.

Given the identification of where lock conflicts are detected, we set our requirement, which is to enable semantic-based concurrency control in resolving a lock conflict at the location where the lock conflict is detected.

The approach to the requirement is described in the next section.

3.2 The Approach

To enable semantic-based concurrency control to be used in resolving conflict, we consider a protocol in which a client validates its transaction at the end of a method. This aims to allow a method to finish and so method semantics can be used to check whether a lock conflict can be released. Section 2.3.3 noted that atomicity of a method call must be preserved in semantic-based concurrency control. Therefore by checking at the end of a method, the requirement for the atomicity of the method is fulfilled, as the method has entirely finished. Therefore at this point, validation using the semantics of the method can be performed.

The preceding identification on where lock conflicts are detected shows that the type of conflicts includes read-write and write-write conflicts. Section 1.2 describes some examples of these type of conflicts that are released by using method semantics. It is important to note that the essence of releasing read-write conflicts and write-write conflicts are different, as will be described in the following sub-sections.

3.2.1 Handling a read-write conflicts

The essence of releasing a read-write conflict between two transactions on a database item is to allow one transaction to read a database item while the other transaction is writing on it, or vice versa.

This is better explained by an example of a scenario. The example in Section 1.2 shows a scenario where a write on an order can be performed while the order is being read. Let us discuss the scenario in a data-shipping system, as illustrated in Figure 3.1.

```

Client-1          Server          Client-2
-----          -
Start T1
...
read() {
  read s
}

                                Start T2
                                cancel() {
                                  s=cancelled
                                }
                                <-- validate

                                receive validation msg
                                consistency msg on s
                                <--

[commutativity
 detected]
consistency on s is OK

                                validation ok -->

                                Commit T2

                                s=cancelled

Commit T1

```

Figure 3.1: Releasing read-write conflict

First, client 1 is reading an order *s*, and so *s* is fetched from the server and stored into its cache. Client 2 intends to cancel the order *s*, and so it fetches *s* from the server. Therefore each client now currently has *s* in its cache. Next, transaction T2 attempts to cancel the order, by setting *s* to

“cancelled”, but *s* is being read by T1. Upon receiving a validation request from client-2, the server sends a consistency message to client-1 to remove *s* from its cache. As client-1 has a readlock on *s*, a read-write conflict occurs. However, client-1 knows that there is a commutativity relationship, which allows T1 to continue to read *s* while T2 has a writelock on it. Client-1 therefore informs the server that the consistency action is unnecessary, and so the server acknowledges client-2 that the write can proceed. When client-2 commits, *s* is updated to become “cancelled”. From this point, T1 is reading the old value of *s* (i.e. before *s* is “cancelled”) until it commits.

```

Client-1          Server          Client-2
-----          -
Start T1
...
read() {
  requests s -->

  read s
  ...
}

Commit T1

consistency msg.to
  other clients
s is writelocked

receives request for s
[commutativity]
<-- sends s

validation ok -->

s=cancelled

Start T2
cancel() {
  s=cancelled
}
<-- validate

...
Commit T2

```

Figure 3.2: Releasing read-write conflict

The preceding example (Figure 3.1) is a read-write conflict detected at a client. Next, we will discuss an example where a read-write conflict is detected at the server. This occurs when a client wants to read a database item that is not in its cache, and so the client requests the database item from the server, where it is already writelocked. The example is shown in Figure 3.2. Following the

preceding example, firstly client-2 is in the process of canceling the order s , and so s is writelocked at the server. When client-1 requests s from the server, a read-write conflict occurs, but due to method commutativity the conflict is released and the server can send the requested s to client-1.

Thus, the essence of releasing the read-write conflict is that a read on a database item can proceed although a write on the database item is currently being performed.

Any subsequent read of s in client-1 (if any) can proceed only if the method that reads s commutes with the previous `cancel()` method of transaction $T2$. Therefore client-1 must record the previous commutativity by recording s and the `cancel()` method. Any subsequent read of s on client-1 will firstly involve checking, from the client's records, whether the method commutes with the `cancel()` method. If the methods do not commute, client-1 should remove s from its cache and re-requests s from the server.

Next, we describe releasing write-write conflicts.

3.2.2 Handling a write-write conflict

The essence of releasing a write-write conflict on a database item between two transactions, which here are called “commutating transactions”, is that the result of one transaction does not influence the result of the other. The state of an item after a write by a transaction is not needed by the other transaction. Therefore, when a write-write conflict between two transactions can be released, if one commutating transaction aborts, the result of the aborted transaction does not influence the overall result, so that the other commutating transaction can still proceed.

To illustrate, let us take and discuss the example from Section 1.2 regarding a car rental, which was introduced in the previous study [MWBH93]. The status attribute of an Order object can take the values “paid”, “shipped” or “paid and shipped”. The methods `paid()` and `ship()` commute over write on the attribute status (s). Four scenarios (five if we include non-concurrent update) need to be addressed as follows.

Firstly, we show scenario 0 in Figure 3.3, in which the transactions run serially, not concurrently, so that no write-write conflict occurs. When transaction $T2$ starts, client-2 first fetches s which

already has the value “shipped”, so that when T2 updates *s* the value becomes “paid and shipped”.

```

s=null;

client-1          server          client-2
-----          -----          -----
Starts T1
(s=null)
s=shipped
validates -->

                validating T1
                s=shipped
                <-- validation OK

Commit T1 -->

                committing T1
                DB: s=shipped

                                starts T2
                                (s=shipped)
                                s=paid+shipped
                                <-- validates

                validating T2
                s=paid+shipped
                validation OK -->

                                <-- Commits T2

                committing T2
                DB: s=paid+shipped

```

Figure 3.3: Scenario-0: serial, non-concurrent

In the next scenario (Scenario 1) as shown in Figure 3.4, both T1 and T2 overlap in time, and both commit, so that the status *s* ends up with value “shipped and paid”. All clients start with *s*=null in their cache. Client-1 runs the `ship()` method that updates *s*=shipped and then validates the method to the server so that the server then knows that *s*=shipped. Then client-2 sets *s* to paid and validates this (i.e. `paid()` method) to the server. The server detects the commutativity relationship between `paid()` and `shipped()`. However, the server does not update *s* to become “paid and shipped” but instead records *s*=paid for client-2 and *s*=shipped for client-1. The reason is that if eventually T2 or T1 aborts, the value of *s* can revert back to that set only for T2 or only for T1 (The scenario in which one of the transactions aborts is discussed later). When T1 commits the server records *s*=shipped

on client-1 in the database because T2 has not committed. Afterward when T2 commits, the server knows that both T1 and T2 have ended, so that s is finalised to be “shipped and paid” and is stored into the database.

```

s=null;

Client-1          Server          Client-2
-----          -
Starts T1
(s=null)

                                     Starts T2
                                     (s=null)

s=shipped
validates -->

                                validating T1
                                s=shipped
                                <-- validation OK

                                s=paid
                                <-- validates

                                validating T2
                                [commutativity]
                                c1.s=shipped
                                c2.s=paid
                                validation OK -->

Commits T1 -->

                                committing T1
                                DB: c1.s=shipped
                                [commutativity]
                                [T2 has not committed]
                                s not finalised
                                until T2 commits/aborts

                                <-- Commits T2

                                committing T2
                                DB: c2.s=paid
                                [commutativity]
                                [T1+T2 has ended]
                                DB: s=shipped+paid

```

Figure 3.4: Scenario 1: concurrent T1 and T2; both commits

The next scenario, Scenario 2 as shown in Figure 3.5, occurs when T1 commits but T2 aborts. It has similar start as scenario 1 but when T2 aborts, the server cancels s=paid by removing s=paid

for client-2. Then, as T1 and T2 have ended, the server finalises *s* to become “shipped” and stores it into the database.

```

s=null;

Client-1          Server          Client-2
-----          -
Starts T1
(s=null)

s=shipped
validates -->

                validating T1
                s=shipped
                <-- validation OK

                validating T2
                commute
                c1.s=shipped
                c2.s=paid
                validation OK -->

                s=paid
                <-- validates

Commits T1 -->

                committing T1
                DB: c1.s=shipped
                [commutativity]
                [T2 has not committed]
                s not finalised
                until T2 commits/aborts

                aborts T2

                [T2 aborts]
                remove c2.s
                [commutativity]
                [T1 and T2 has ended]
                DB: s=shipped+null=shipped

```

Figure 3.5: Scenario 2: concurrent T1 and T2; T1 commits, T2 aborts

In the next scenario (Scenario 3) as shown in Figure 3.5, T1 aborts but T2 commits. It starts similarly to scenario 1, but when T1 aborts, the server cancels *s=shipped* by removing *s=shipped* for client-1. When T2 commits, the server knows that T1 and T2 have ended, and so *s* is finalised to become “paid” and is stored into the database.

```

s=null;

Client-1          Server          Client-2
-----          -
starts T1
(s=null)

s=shipped
validates -->

                validating T1
                s=shipped
                <-- validation OK

                                s=paid
                                <-- validates

                validating T2
                [commutativity]
                c1.s=shipped
                c2.s=paid
                validation OK -->

aborts T1

                [T1 aborts]
                c1.s=null
                [commutativity]
                [T2 has not committed]
                s not finalised
                until T2 commits/aborts

                                <-- T2 commits

                committing T2
                [commutativity]
                [T1 and T2 has ended]
                DB: s=paid

```

Figure 3.6: Scenario 3: concurrent T1 and T2; T1 aborts, T2 commits

The last scenario, Scenario 4 as shown in Figure 3.7, is when both transactions T1 and T2 abort. Similarly, after T1 or T2 aborts, the server cancels the value of *s* by removing *s* for each client. At the end of the scenario, *s* does not have any value from the transactions.

From the preceding example, it is important to note that a write-write conflict between two

```

s=null;

client-1          server          client-2
-----          -
starts T1
(s=null)

s=shipped
validates -->

                validating T1
                s=shipped
                <-- validation OK

                                s=paid
                                <-- validates

                validating T2
                commutativity
                c1.s=shipped
                c2.s=paid
                validation OK -->

aborts T1

                [T1 aborts]
                c1.s=null
                [commutativity]
                [T2 has not committed]
                s is not finalised
                until T2 commits/aborts

                                aborts T2

                [T2 aborts]
                c2.s=null
                [commutativity]
                [T1 and T2 has ended]
                DB: s=null

```

Figure 3.7: Scenario 4: concurrent T1 and T2; T1 aborts, T2 aborts

transactions can be released when the write operations are independent of each other, that is the result of a write by one transaction does not influence the write by the other transaction.

From the description of the preceding scenarios, we can define the following algorithm at the server:

- When a write to A by T1 conflicts with a write to A by T2 but can be released due to method semantics, the server records the value of A for each transaction: T1.A and T2.A in a data

structure. We name the data structure the Semantic Record (SR). Then, when aborting or committing a transaction, the server runs the algorithm shown in Figure 3.8.

```

If a transaction T1 aborts,
  for each database item A of T1 in SR {
    set T1.A to null
    if a commutativity with T2 is detected {
      if T2 has ended {
        if (T2.A is not null) {
          A = A + ( T1.A=null + T2.A )
          store A into database
        }
        remove T1 and T2 from SR
      }
      else {
        do nothing
      }
    }
  }
}

If a transaction T1 commits,
  for each database item A of T1 in SR {
    if a commutativity with T2 is detected {
      if T2 has ended {
        A = A + ( T1.A + T2.A )
        store A into database
        remove T1 and T2 from SR
      }
      else {
        store T1.A into database
      }
    }
  }
}

```

Figure 3.8: Server's algorithm

The following are the additional overheads required to support releasing a write-write conflict:

1. The server must record the value of each write by each commutating transaction in a data structure.
2. The database storage requires additional space to store the value of each write by each commutating transaction.

3.3 The Protocol Design

This section describes the protocol that is designed and evaluated.

The idea in general in this protocol is that a client validates a transaction at method level, and that locks are acquired on attributes. An attribute is an object's attribute that is atomic. A client executes its transactions locally by running methods on objects that reside in the client's local cache. During the execution of a method, when accessing an attribute, a lock is acquired on the attribute. When the method ends, the client validates the transaction to the server by sending information about the updated attributes, the method and its ancestors if any.

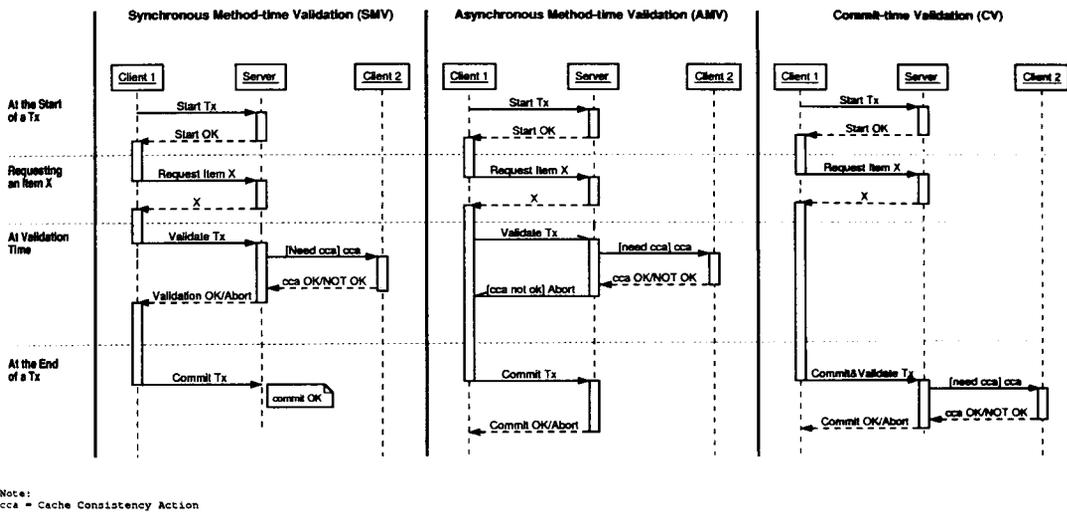


Figure 3.9: SMV, CV and AMV - a brief illustration

Based on when a client sends a validation message to the server, we consider three protocols. An overview of the protocols is shown in Figure 3.9, and described as follows.

- 1. Synchronous Method-time Validation (SMV).** In this protocol, the client runs the method locally and at the end of the method the client sends a validation message to the server and then waits for the result. If the validation is successful the client continues the transaction, otherwise the transaction aborts. At commit time the client can commit because all validations have been successful.

2. **Commit-time Validation (CV)**. In this protocol, a client runs the entire transaction locally without validating until commit time. At commit time the client validates the transaction to the server and then waits for a response. If the validation is successful then the transaction can commit, otherwise it aborts. Being an optimistic protocol, CV is intended to be a comparator to SMV.

3. **Asynchronous Method-time Validation (AMV)**. In this protocol, a client runs methods locally and at the end of the method the clients validates the transaction. Unlike in SMV, after a client sends a validation message to the server the client does not wait for a response from the server, but continues the transaction until commit time. At the end of each method the client sends a validation message to the server, and then continues on its transaction without waiting. At commit time the client sends a commit message to the server and waits for a response from the server. If all the asynchronously-sent validations are successful the transaction can commit, otherwise it is aborted. The purpose of the asynchronous version of the protocol is to reduce blocking time by allowing client not to wait after sending validation message. This protocol is the asynchronous version of the SMV protocol, and will again be used as a comparison. However, the implementation of the basic form of AMV (i.e. without method semantic-based commutativity support) in this thesis has met much more complexity than SMV, and so we believed it will require much more extra overhead if we implement AMV that allows method commutativity to be associated in concurrency control. Therefore, the scope in this thesis is the comparison only to the basic form of AMV.

In the following subsections, we describe the design of each protocol in detail.

3.3.1 Synchronous Method-time Validation (SMV)

As mentioned in the preceding description, SMV is a client cache consistency protocol that can exploit semantic based concurrency control. The following is the description about the protocol. The basic form of SMV, i.e. without method semantic-based commutativity support, is illustrated

in Figure 3.10.

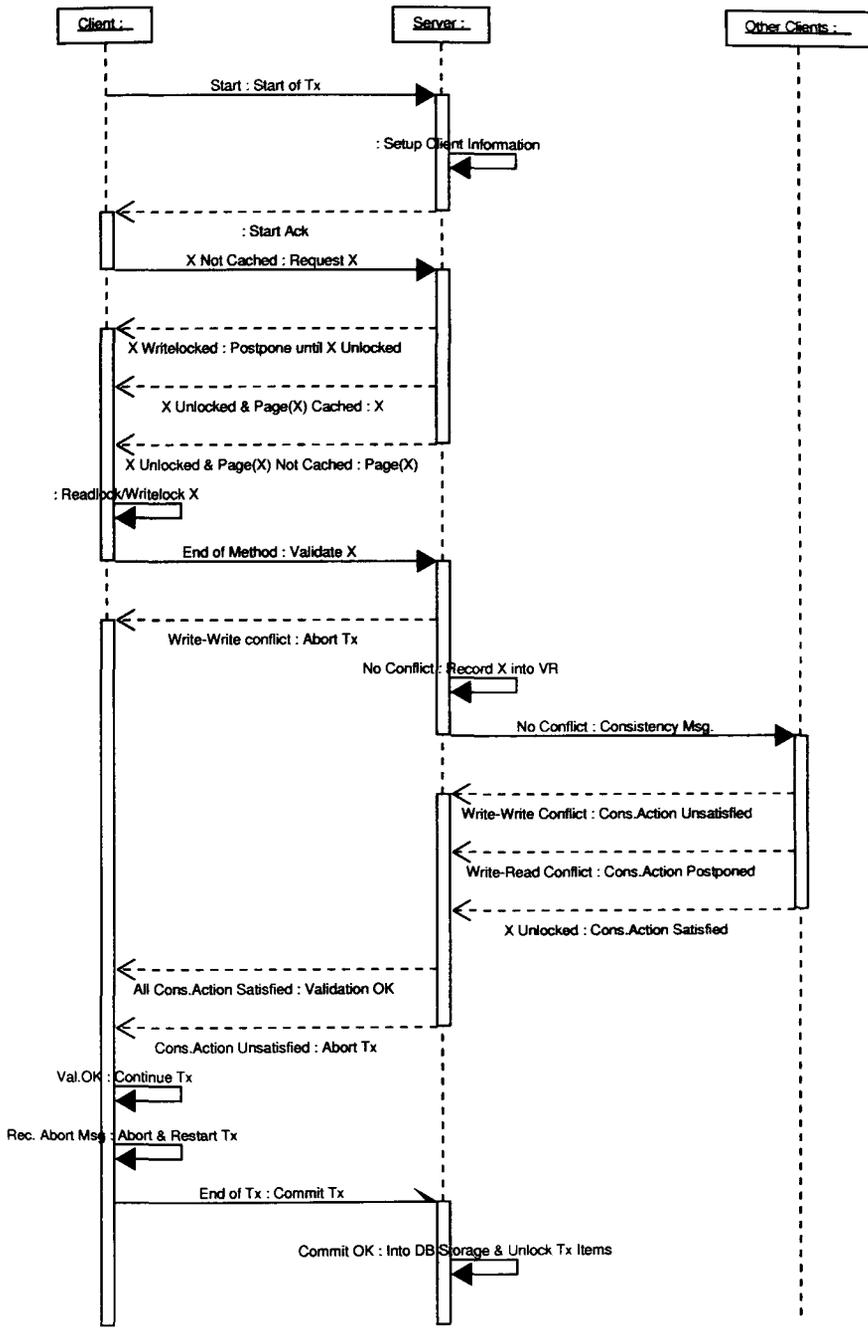


Figure 3.10: The basic form of Synchronous Method-time Validation (SMV) protocol

At the Client

1. At the start of a transaction

At the start of a transaction, a client sends a *start* message and waits for a reply from the server. The reason for waiting is to ensure that the transaction is initialised at the server, this include having its timestamp recorded, which can be used in resolving any future deadlock that may occur.

2. Read or Write access to an attribute

When the client accesses an object's attribute the client takes the appropriate lock, either readlock or writelock, on it. However, an extension to its basic form is that before acquiring a readlock on an attribute that is already cached, the client should check if a readlock has been given to the attribute, based on commutativity. If so then the client should remove the attribute and then request the attribute again from the server.

3. Requesting an attribute

When a client needs to read an attribute that is not currently available in the client's cache, the client sends a request message to the server. Then the client waits until receiving the attribute from the server. When the attribute is received, it is installed into the client's cache, and is then readlocked by the client.

However, when a client needs to read an attribute whose page is not currently available in the client's cache, the client requests the page of the attribute from the server. When the page arrives at the client, the client takes a readlock on the attribute. The purpose of requesting the page rather than the attribute is to try to reduce message overhead. If the client requires subsequent attributes that are also in the same page, then the client will not be required to request the attributes from the server.

In a case where other non-requested attributes in the same page are being writelocked at the server by other transactions, the client cannot fetch the other attributes due to the read-write conflict. Therefore, in this case the client will receive the page, but the other non-requested

writelocked attributes will be labeled as unavailable. Our approach in this case follows that from an existing study [ALM95].

4. At validation time

Validation is performed at method level. At the end of a method, a client sends a validation message to the server and then waits for a response. A validation message from a client is the client's notification to the server that the client has writelocked some attributes and then the client asks the server whether the writelock can continue. The essence of the validation here is similar to that in the existing studies in that a client checks (with the server) whether a database item can be writelocked at the client. The difference is the time when the check is made. In CBL (Callback locking; pessimistic, avoidance-based), the check is made before a database item (i.e. page) is writelocked, and so the validation is a write intention notification. By comparison in our protocol (i.e. SMV) the check is made after a database item (i.e. attribute) is writelocked, which is at the end of method. A validation message contains all the updated attributes within the method, along with information about the method and its ancestors (if any). Information about the method and its ancestors are used by the server for semantic-based concurrency control. If the client receives a response from the server that the validation is successful the client continues its transaction, but if the validation fails the transaction aborts and restarts.

5. At the end of a transaction

At the end of a transaction, all validations must have been successful. The client sends a commit message to the server and then the transaction can commit.

6. Receiving a cache consistency message

A client receives a consistency message from the server when the server asks the client to remove one or more attributes from the client's cache. This is because those attributes reside at the clients' cache but another transaction is requesting writelocks on them. It should be noted that the number of attributes requested for removal by a consistency message can be

more than one, because a consistency message can be on behalf of a validation of method that updates more than one attributes. Then, whether or not the removal of attributes can be performed depends on the current state of the attributes:

- (a) If all the attributes are currently unlocked, the client can remove them from its cache and acknowledge to the server about the successful consistency action.
- (b) If one of the attributes is readlocked, a read-write conflict occurs, and so the client cannot remove it but postpones the consistency action until the attribute is unlocked. Once the attribute is unlocked, the client removes it and informs the server about this successful consistency action. However, as the extension to SMV's basic form, if the read-write conflict can be released by method commutativity, then the client can continue with the read while acknowledging the server that the consistency action is not necessary. If a client detects that a read-write conflict is released due to method semantics, the client should record the commutativity. The record consists of the attribute and the method that performs the write. As was described in Section 3.2.1, the record will be used by the client if there are further reads on the attribute.
- (c) If one of the attributes is writelocked, a write-write conflict occurs and so the client cannot remove it, and the client acknowledges to the server that the consistency action cannot be performed. However, as the extension to SMV's basic form, when a write-write conflict can be released due to method commutativity, all transactions can continue, in that the client can continue with the write while acknowledging to the server that the consistency action is not necessary.

Acknowledgements from client to the server regarding the consistency action are piggybacked on other messages from the client to the server, in order to reduce message overheads.

At the server

1. At the start of a transaction

When the server receives a `start` message from a client, the server initialises the client's transaction. Then the server sends an acknowledgement to the client.

2. Receiving a request for an attribute/page

The server receives a request for an attribute or page from a client when the client wants to be a readlock on it, but it is not available in the client's cache. Upon receiving the request, the server first checks whether the attribute is writelocked. The check is performed by consulting the Validation Record (VR) that contains a list of the attributes writelocked by transactions. Recall that an attribute is writelocked at the server on behalf of a transaction that is holding a write intention on it but has not finished, or when consistency actions on the attribute are being processed by the server on behalf of a transaction. When a request on an attribute arrives, if the attribute is registered in the VR, a read-write conflict occurs, and the server postpones sending it until it is no longer registered in the VR. However, as the extension to SMV's basic form, if the read-write conflict can be released due to method commutativity, the server sends the attribute to the client and also information about the method with which the read method commutes. As described in Section 3.2.1, the record will be used by the client for further reading of the attribute.

As mentioned in the preceding client part, a request made by a client can be for an attribute or for the page containing the attribute. If the request is for the page containing the attribute, other attributes within the page may have been writelocked (i.e. registered in the VR) by other transactions. Therefore if the request is for the page containing an attribute and the attribute is not writelocked, the server will send the page, and if other attributes in the page are registered in the VR then these attributes will be marked as unavailable in the page that is sent to the client.

3. On receiving a validation message

Recall that a validation message from a client is the client's notification to the server that the client has writelocked some attributes and wishes to know if the writelocks can continue to be held. To decide this, firstly the server needs to check if the attributes are being writelocked by another transaction. The check, on whether an attribute is writelocked, is performed by the server by consulting the Validation Record (VR):

- If the attribute is not writelocked at the server (i.e. not registered in the VR), then a write-write conflict has not occurred.
- If the attribute is writelocked, i.e. registered in the Validation Record, a write-write conflict has occurred (i.e. the attribute is registered in the Validation Record). However, as the extension to SMV's basic form, if a method commutativity is detected*, the write-write conflict can be released. As mentioned in Section 3.2.2, when releasing a write-write conflict between two transactions T1 and T2, the server should record the value of each Write by each commutating transaction in the Semantic Record data structure. The database storage should also allocate additional space to store the value of each Write for each commutating transaction.
- If the attribute is writelocked, i.e. registered in the Validation Record, but the write-write conflict cannot be released, the server rejects the validation and sends an acknowledgment to the client that the transaction must abort.

If a write-write conflict does not occur, or can be released due to method commutativity, the server performs a consistency action. First, it checks whether the copy of the attribute is currently cached at other clients. The check is performed by consulting the Cache Record that records the attributes cached at clients.

- If a copy of the attribute is cached by other clients, then the server will perform a con-

*Recall that a validation request from a client contains all the updated attributes and also information about the method and its ancestors (if any). The information then is used for semantic-based concurrency control. When the server detects a write-write conflict on an attribute, the server checks the commutativity between the method of the attribute being validated and the method of the attribute in the Validation Record

sistency action by sending a consistency message to the other clients that are caching a copy of the attribute, and wait until receiving a response from all the other clients.

- If the attribute is not cached by the other clients then the server will register the attribute in the Validation Record, which means writelocking the attribute, and sending a message to the client that the validation was successful.

When the server receives a response regarding successful consistency actions from all the clients, the server's consistency action is successfully performed, and so it sends a message to the validating client that the validation was successful. However, when the server receives a response from a client that the consistency action cannot be performed, the server rejects the validation request and sends an abort message to the validating client.

Figure 3.11 summarises the algorithm when the server receives a validation message from a client.

```

if a write-write conflict does not occur or can be released then {
  if the attribute(s) are cached by other clients then {
    send consistency message to the clients and wait
    if all the consistency actions are successful then {
      the validation is successful
    }
    else {
      the validation fails
    }
  }
  else {
    the validation is successful
  }
}
else {
  the validation fails
}

```

Figure 3.11: Algorithm for handling a validation request

4. The server may abort a transaction, by sending an abort message to a client. When a transaction is aborted, all writelocked attributes are released. In addition, as the extension to SMV's basic form, if an aborted transaction is a commuting transaction (i.e. the transaction has

a history of using method commutativity to release a write-write conflict), as mentioned in Section 3.2.2, the algorithm given in Figure 3.12 is performed by the server.

```

for each database item A of T1 in SR {
    set T1.A to null
    if a commutativity with T2 is detected {
        if T2 has ended {
            if (T2.A is not null) {
                A = A + ( T1.A=null + T2.A )
                store A into database
            }
            remove T1 and T2 from SR
        }
        else {
            do nothing
        }
    }
}

```

Figure 3.12: When a commutating transaction commits

5. Upon receiving a commit request from a client

When the server receives a commit message of a transaction, all previous validations of the transaction must have been successful. The server commits the transaction by storing the corresponding items into stable database storage and releases all previously locked items. In addition, as the extension to SMV's basic form, the algorithm in Figure 3.13 should be performed by the server if the committing transaction is a commutating transaction, as mentioned in Section 3.2.2.

3.3.2 Commit-time Validation (CV)

Commit-time Validation (CV) is the optimistic version of the protocol. In CV, a client validates a transaction to the server only at the end of the transaction. As mentioned in Section 1.3, we wish to compare the performance of Synchronous Method-time Validation (SMV) against that of the optimistic version of the protocols, which is Commit-time Validation (CV). The reason for comparing against an optimistic protocol is to observe the expected performance superiority, as a

```

for each database item A of T1 in SR {
  if a commutativity with T2 is detected {
    if T2 has ended {
      A = A + ( T1.A + T2.A )
      store A into database
      remove T1 and T2 from SR
    }
    else {
      store T1.A into database
    }
  }
}

```

Figure 3.13: When a commutating transaction commits

previous study [Fra96] showed that Optimistic Two-Phase Locking (O2PL) as an optimistic client cache consistency protocols, was superior to the pessimistic Callback Locking (CBL) protocol.

Commit-time Validation protocol is illustrated in Figure 3.14. From the start to the end of a transaction, a client runs its transaction locally. When a client does not have an attribute in its cache, the client requests it from the server. After receiving the attribute, the client continues with the transaction. During the transaction, the client does not validate the transaction at the server. Instead, at commit time, the client sends a commit message and validates the entire transaction to the server, and then waits for a response from the server. If the validation is successful then the client can commit, otherwise the transaction is aborted.

1. At the client

The client's actions when starting a transaction and requesting an item are similar to the basic form of SMV and AMV. But here the validation time and the commit time are at the end of a transaction. A client validates by sending all Write operations in the entire transaction to the server. Then the client waits for a reply from the server.

2. At the server

The server handles the start request, the item request, and the validation request similarly to the basic form of SMV and the AMV protocols. The difference is that the validated attributes are for the entire transaction, in contrast to the SMV and AMV protocols where the attributes

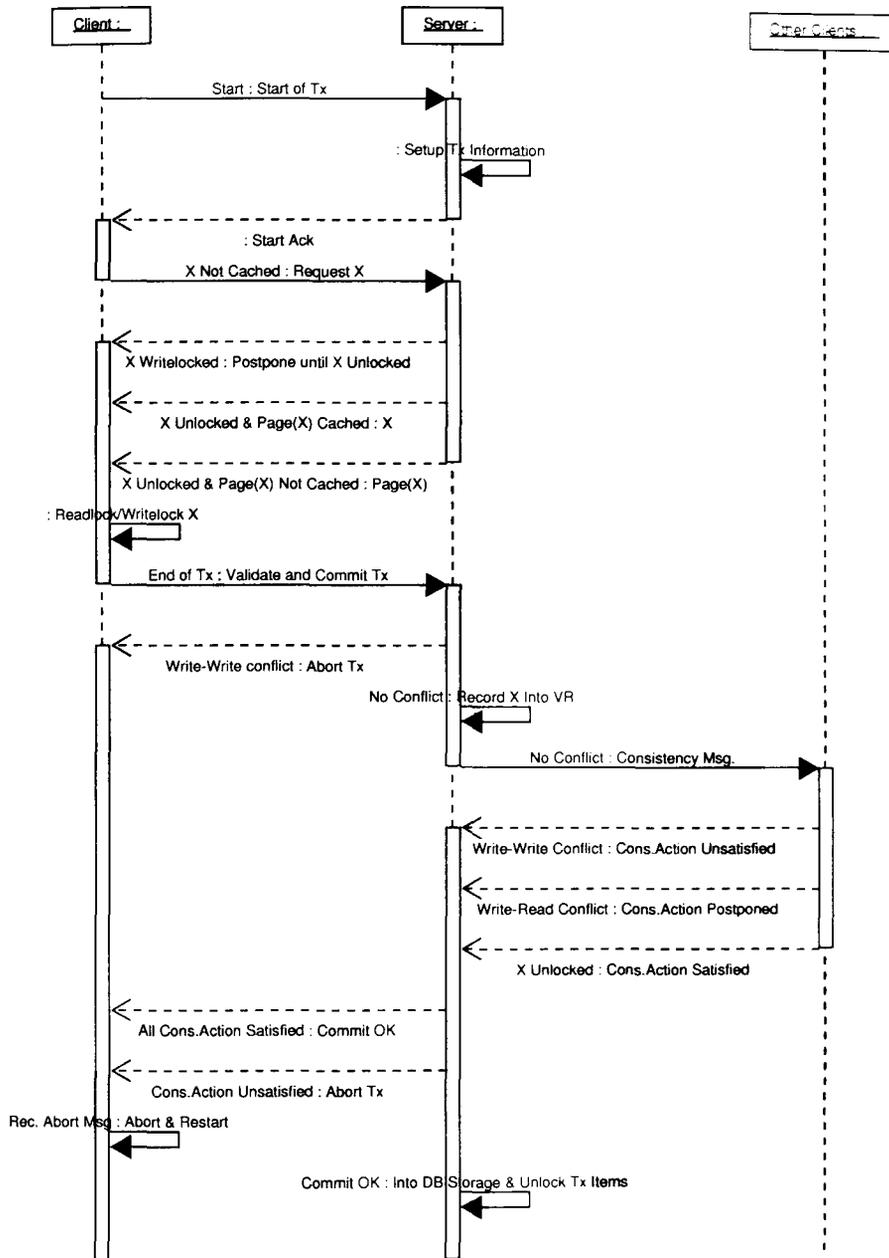


Figure 3.14: The Commit-time Validation (CV) protocol

are from one or several methods.

3.3.3 Asynchronous Method-time Validation (AMV)

Asynchronous Method-time Validation (AMV) is the asynchronous version of the method-time validation protocol. Unlike in SMV, in AMV a client does not wait for a response from the server after sending a validation message to the server. Instead, after sending a validation message to the server, a client continues with its transaction locally until the commit time of the transaction. When the server receives a validation message, the server validates the transaction but does not send the result of the validation back to the client. The server, however, will send explicit abort message if a validation fails or if a transaction needs to abort[†]. The purpose of designing this asynchronous protocol is to reduce the client's blocking time while still maintaining the possibility of detecting conflicts before commit time. It also reduces the number of messages sent from the server because the server does not send validation results to clients. However, while it may appear that the difference between AMV and SMV is small, the design of AMV is much more complicated than that of SMV.

As has been mentioned early in this section (Section 3.3), the scope of the implementation of AMV in this thesis is limited to the basic form of AMV, because it requires much more extra works to implement this asynchronous protocol that allow method commutativity to be associated in concurrency control. Therefore, here the description of AMV protocol covers only the basic AMV, i.e. without method semantic-based commutativity support.

The AMV protocol is illustrated in Figure 3.15. The following are the descriptions of the protocol.

1. At the client

- (a) At the start of a transaction

At the start of a transaction, a client sends a *start* message to the server and waits for the reply. The server will need to initialise the transaction, such as obtaining the time-stamp of the transaction.

[†]An abort of transaction can be due to a deadlock or a failed validation

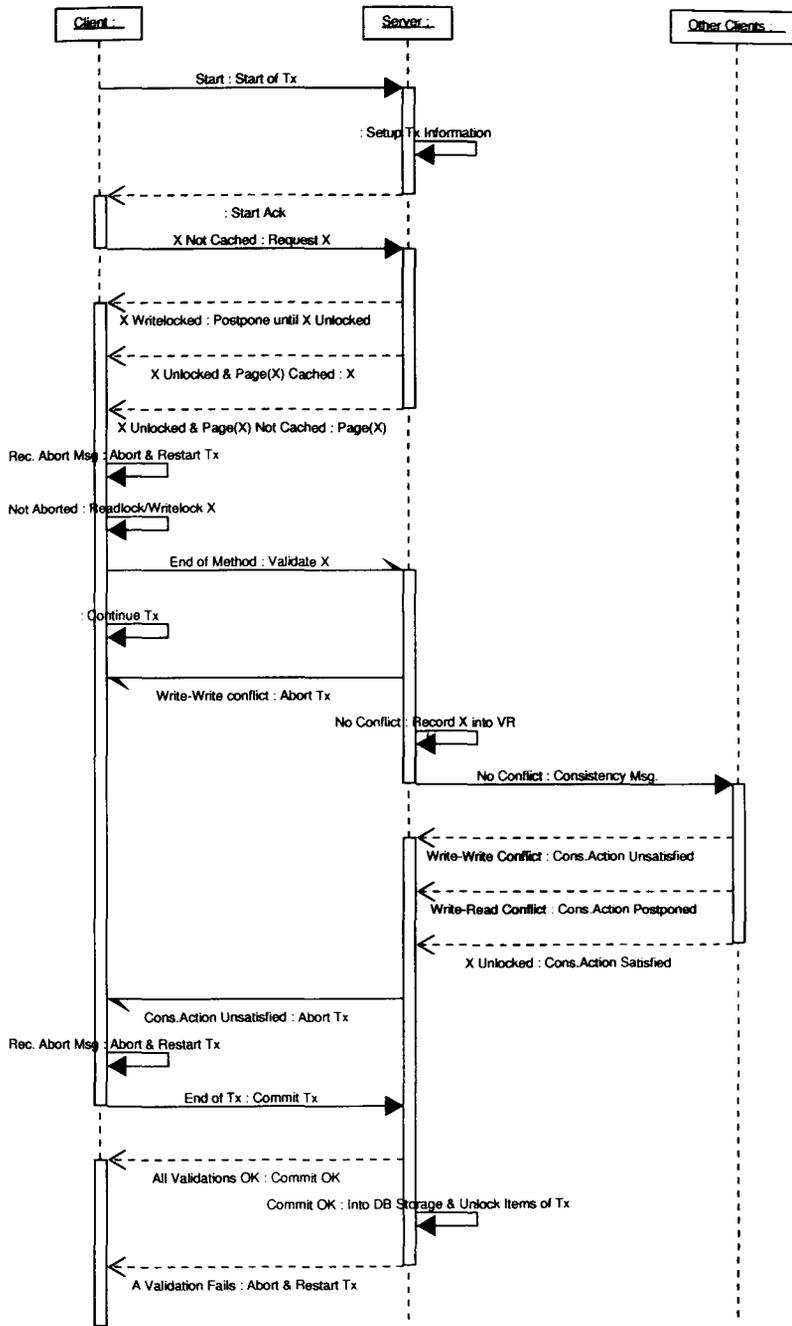


Figure 3.15: The Asynchronous Method-time Validation (AMV) protocol

(b) Requesting an attribute or page from the server

When a client does not find an attribute from its cache, the client sends a request message for the attribute or the page containing the attribute to the server and then waits for a response from the server. After the client receives the requested attribute or page the client installs it in its cache.

(c) Accessing an attribute

When a client accesses an item, an appropriate lock (either readlock or writelock) is acquired by the transaction.

(d) At validation time

At validation time, which is at the end of a method, a client sends a validation message to the server and then the transaction continues without waiting for a response from the server. Hence, a client sends validation messages asynchronously. As in the SMV protocol, the validation message contains all Write operations, its method and its ancestors if any.

(e) At the end of transaction

At the end of transaction the client sends a commit message and waits for a response from the server. If the previous asynchronous validations were successful then the client will receive a response from the server that the transaction can commit. However, the client can also receive an abort message from the server, ordering the transaction to abort, which can be due to failed validation or because the transaction is chosen to be aborted due to a deadlock.

(f) When receiving a consistency message from the server

Recall that a client receives a consistency message because the server asks the client to remove one or more attributes from the client's cache, as those attributes reside at the clients' cache and are intended to be writelocked by another transaction. If at least one of the attributes are not currently locked at the client, it is removed from the cache and the client acknowledges to the server that the consistency action is complete. If at least

one of the attributes are being readlocked at the client, their removal is postponed until those attributes are unlocked. If at least one of the attributes are being writelocked at the client then they cannot be removed and the client acknowledges to the server that the cache consistency action cannot be satisfied. As in SMV, to save message overheads, the acknowledgement from client whether or not a consistency action can be satisfied is piggybacked on another message to the server.

2. At the server

(a) At the start of a transaction

When the server receives a *start* message from a client, the server initialises information for the client. Firstly, the client initialises the timestamp of the transaction to be used for resolving deadlock. Secondly, the server must record the fact that the transaction has not yet been aborted. The reason of this is that in AMV, the server receives validation messages asynchronously, but when a validation fails the server explicitly sends an abort message to the client. As a consequence, the server may receive a validation message from a client to which it has previously sent an abort message. In this case the validation message should be discarded. Consequently, the server must keep a record of whether or not the transaction has been aborted.

(b) When receiving a request for attribute or page from a client

When receiving a request for an attribute or page, the attribute or page can be sent if the attribute is currently not locked. If the attribute is writelocked, it can only be sent to the client after it is unlocked. Like in SMV, the server will send either the attribute itself or the page holding the attribute, depending on what the client requests. This will depend on whether or not the client has already cached the page holding the attribute.

(c) When receiving a validation request from a client

The server's action when handling a validation request from a client is, firstly, to check if the attributes validated are writelocked. If they are not writelocked, the server sends

a cache consistency message to the other remote clients holding the attribute, and the validation is successful if all the cache consistency actions are satisfied (all the remote clients can remove the attributes from their local cache). However, when the validation is successful, the server does not acknowledge the validating client. But if a validation fails, the server explicitly sends an abort message to the validating client.

(d) Receiving a commit request from client

When receiving a commit message from client, the server must have processed all previous validations on behalf of the transaction. If all the validations have been successful, the server commits the transaction by sending a commit response to the client, and stores the committed attributes into the database. Otherwise the server sends an abort message to the client.

The implementation of the changes that must be made on the SMV protocol to create this asynchronous protocol are more complex than it may appear to be.

Firstly, from the preceding description, it may be assumed that the server can merely ignore the messages sent by an aborted client. However, important information may be on a message (piggybacked) from the client. This can be information on the result of a cache consistency action, or information that a client is no longer caching a page (due to the cache replacement policy). Figure 3.16 shows an example in which the server must accept the acknowledgement from an aborted transaction. Transaction T-1 has just been aborted when the server receives a message from the corresponding client containing piggybacked information about failed consistency action. Since the information can be important for another transaction that is waiting for the consistency action result, the server must accept the message, although the message comes from an aborted transaction. Moreover, information that a client is no longer caching a page, is necessary to synchronise the contents of the client's cache and the information held by the server. If the server has recorded that an attribute is currently cached by a client but actually the client has no longer cached the attribute, then it can cause redundant cache consistency messages to be sent by the server to the client. Consequently information (piggybacked on to messages) from aborted transactions must still be processed by the

server.

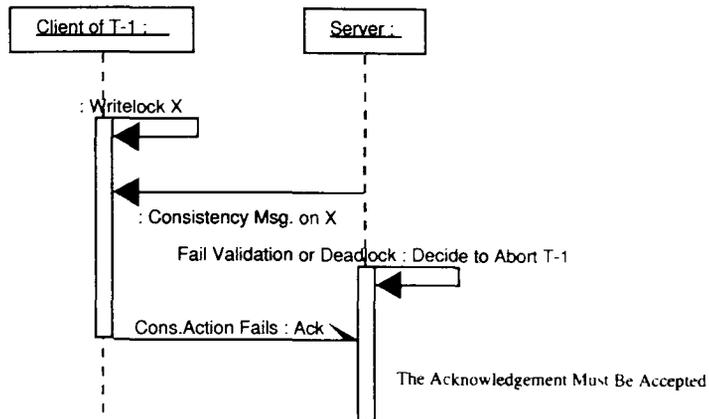


Figure 3.16: A case in the AMV protocol

Furthermore, the AMV protocol now needs additional complexity to handle validation messages. The amount of network delay time experienced by a message can vary, and so a validation message that is sent before another validation message may actually arrive later than the other validation message. In other words, validation messages may be received by the server out of the order in which they were sent. This issue does not occur in the SMV (synchronous) protocol since a transaction cannot send another validation message before receiving the result of its previous validation. To handle this issue, in the AMV protocol each message is tagged with a sequence number, and the server processes validation messages in sequence number order. When the server receives an out of order message, the server postpones processing the message, by putting the message into a temporary queue, until after any predecessors have arrived.

3.4 Possible implementation

This section describes five key aspects of the implementation of the proposed protocols. These are a client accessing an attribute and a client validating a method call.

3.4.1 A client requesting a database item

Here we will describe the implementation of a client fetching an object's attribute from the server. Recall that when a client tries to readlock an attribute but the attribute is not in the client's cache, the client requests the attribute or the page of the attribute from the server. Here, we would show a possible implementation of it in C++.

In C++ we can create a template class, which is a class that can be re-used (generic) for instance of any type. An attribute of object can be represented by a template class whose instance can be called by an object using operator as follows:

```
object -> attribute
```

All necessary operations when accessing an attribute can be encapsulated within the DBField class as shown in Figure 3.17.

3.4.2 The method validation processor

At the end of a method, the client validates its transaction to the server. A method is validated at the server if it contains a Write operation. Before sending the validation message, information about the method that includes all attributes updated within the method, the method itself, and the ancestors of the method if any, as well as the object identifier, are processed by the method processor as shown in Figure 3.18.

The information about a method includes the attributes accessed within the method and the identifier of the method and its ancestors. Hence the method processor gets information about all locks on attributes within a method as well as the method hierarchy.

A possible implementation of identifying the method hierarchy in the method processor is by performing a top-down parsing of the method. The method processor maintains a collection of waiting methods recording the methods which are waiting until their descendants (sub-methods) have finished (Recall the close nested transaction approach described in Chapter 2, in which a transaction cannot finish until its sub-transactions have finished). This is used to record method

```

/**
 * DBField template class defining
 * an object's attribute.
 * An instance of this class
 * can be called by an object:
 *     obj -> attr
 * where attr is an instance of
 * DBField<any type>
 */

template class<T>
DBField<T>::operator T() const {

    //in pseudo code
    //supposed attr = this of type T

    if (attr is not in cache) {
        if (page of attr is not in cache) {
            get page from the server
        }
        else {
            get attr from the server
            (oid, offset, attr-id)
        }
        install attr/page into the cache
    }
    acquire readlock on the attr

    return (attr read from the cache)
}

```

Note

oid : object identifier
(of instance 'obj' in the example)

offset : location of the attribute within
the object's physical space

attr-id : attribute identifier,
defined in object's schema

Figure 3.17: A possible implementation of accessing an attribute in C++ -based pseudo-code

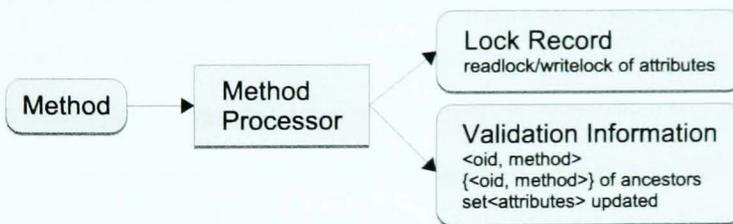


Figure 3.18: Method processor

hierarchy in a transaction.

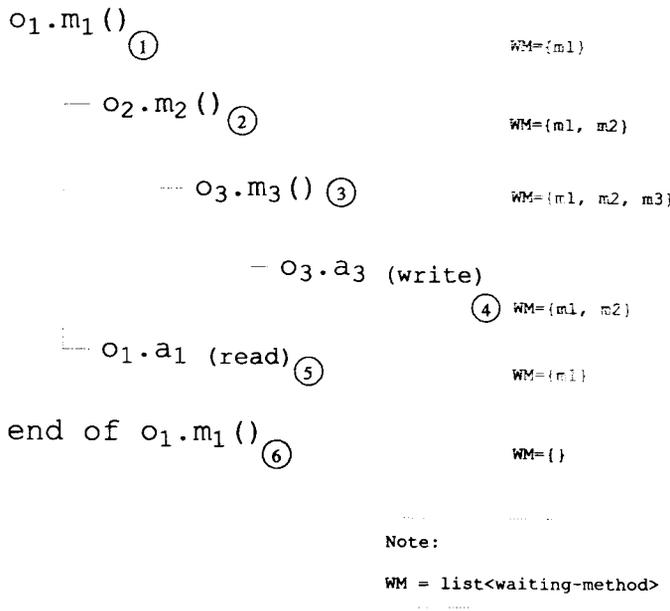


Figure 3.19: An example of method processing

This is best illustrated by an example as shown in Figure 3.19. Suppose that object o_1 has an attribute a_1 , and the method execution hierarchy of the client's transaction is as shown in the figure. The parsing goes from the top of $o_1.m_1()$. How the parser works is language specific. The `waiting-methods` firstly contains $o_1.m_1()$ because the method has not finished (point 1). Then, $o_2.m_2()$ and $o_3.m_3()$ are executed and now the `waiting-methods` contains $o_1.m_1()$, $o_2.m_2()$, $o_3.m_3()$ (point 3). The $o_3.m_3()$ contains a Write on the attribute a_3 , therefore $o_3.m_3()$ needs to be validated to the server. The information to be validated includes the attribute a_3 and the methods recorded in the `waiting-methods`. When the validation is successful (i.e. has received a positive response from the server), the client continues with the transaction. After $o_3.m_3()$ has finished, it is removed from the `waiting-methods` (point 4), and so after $o_2.m_2()$ has finished the `waiting-methods` now contains only $o_1.m_1()$ (point 5). The method $o_1.m_1()$ contains only a Read on the attribute a_1 , so that the method is not validated to the server but the lock is recorded at the client, and at the end

(point 6) the method is removed from the `waiting-methods`.

3.5 Summary

This chapter describes the design of the protocols for this study. The protocol called Synchronous Method-time Validation (SMV) incorporates semantic-based concurrency control in client cache consistency protocol by validation of transactions at the end of each method, so that method semantics can be exploited during concurrency control in order to enhance concurrency. To investigate its characteristics, we also design the optimistic protocol called Commit-time Validation (CV) to which the SMV will be compared. We also design the asynchronous version of the protocol called Asynchronous method-time validation (AMV). However, because of additional complexity in the implementation of AMV, the scope of this thesis includes AMV only with its basic form, without allowing method commutativity in concurrency control. Furthermore, this chapter describes a possible implementation on some key aspects in the protocol. Their performance will be compared using simulation. The next chapter will describe the simulation model for measuring the performance.

Chapter 4

The Simulation Design

In order to investigate the characteristics of the performance of the protocols described in Chapter 3, we measure the performance using simulation. Simulation has the advantage that is to allow us to vary system parameters without changing the actual software or hardware. Moreover, by using simulation we are able to focus more on the algorithms and data structures of the protocols, than on intricate implementation details such as the message passing between client and server.

This chapter describes the simulator and the model used in the simulation. The model includes the system model, the database model and the workload model. The model will be the basis for the results described in Chapter 5.

4.1 The simulation package

We used Simjava-1.2, a Java-based event-driven simulator from the University of Edinburgh [MH96] [HM98], for our simulation. Simjava gained our interest because it allows simulations to be run with and without animation. The animation, in a Java applet, can show visually the simulation entities and the message passing between the entities. It is a more attractive way of observing behaviour than by analysing merely a textual trace file. We found the animation useful for debugging, such as for identifying the state of the simulation entities, and for checking whether the message pass-

ing was implemented correctly. A screenshot of our animation can be seen in Figure 4.1. The left part shows the state of each transaction and the number of times transaction starts and commits, which are useful to notice visually the state of transactions when debugging. The center part shows the clients (penguins) connected to the server, message with a meaningful symbol passing between client and the server, and disk indicating whether it is reading, writing or idle. The right part shows options and input fields for the animation setting. Then, the non-animated simulation is used for measurements after the debugging.

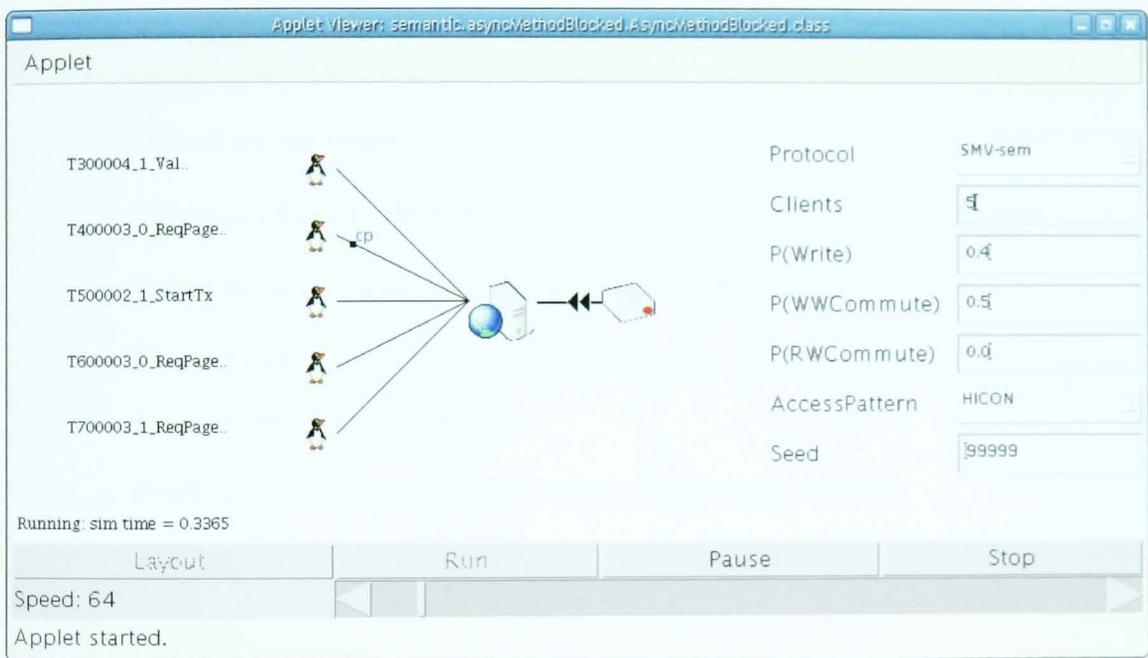


Figure 4.1: A screenshot of the animation of the simulator

Our simulation package consists of about seventy java classes consisting about ten thousand lines of codes. Figure 4.2 shows the class diagram showing the core classes in the simulation package. Each client, and the server, is defined as an entity in the simulation, so we derive both classes from Sim_entity in Simjava. The figure shows the main data structures associated with client and with server.

- **Client.** The *Client* class maintains Lock Records of the attributes accessed by a client. The Client class is associated with the *Workload* class that generates a workload for each client. A workload gets database items from the database. The *Database* class manages the database items. The *Client* class is also associated with the *Cache* class that represents a client's cache, as well as the *Semantic Record* that records the releases of lock conflicts by method semantics-based commutativity.
- **Server.** The *Server* class is also associated with the Cache, the Semantic Record and the Database classes. The other data structures associated with the Server are:
 - Validation Record, which records the successfully validated items of transactions that have not yet committed.
 - Consistency Record, which records consistency actions performed by the server
 - Cached Set, that contains information about attributes cached at each client
 - Modified Buffer, which stores the bytes of database items that are validated by transactions that have not yet committed
 - Deadlock Detector, which checks whether a deadlock has occurred.

4.2 The system model

The system is a client-server model in that many clients are connected to one server through a network, as shown in Figure 4.3. The system parameters are listed in Table 4.1.

The system consists of a set of components. The components include:

1. **CPU (central processing unit).** CPU is the processor, which processes machine instructions at the client and server. We set the processing speed parameter to be in the millions of instructions per second (MIPS).

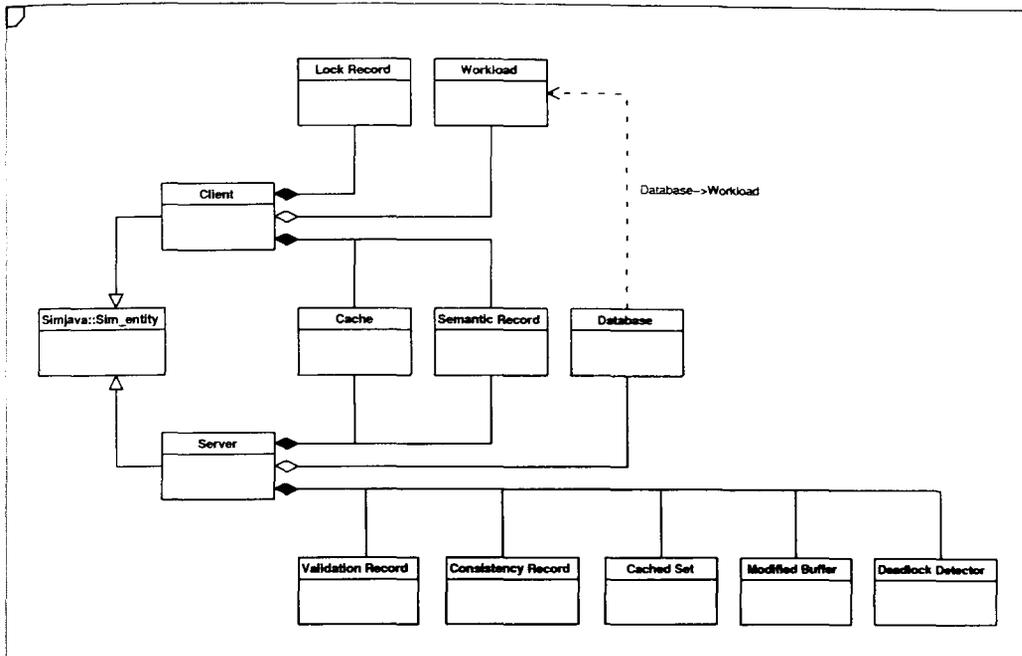


Figure 4.2: Class diagram of the simulation package

2. **Disk.** Disk is stable storage to store persistently database items. The disk is only at the server. The time to read a database item from disk or to write a database item to disk is calculated as the average time*.
3. **Cache.** Cache is a memory area allocated to store database objects that are used by the application. On the client side, the cache allows database objects to be stored closer to the client, whereas at the server cache allows frequently-used database objects to be accessed from memory rather than from disk. The cache has capacity that is measured as a fixed number of pages. When the cache is full, the Least Recently Used (LRU) pages are removed from it. We set a fixed number of instructions as the cost of reading a database item from

*The actual disk cost covers the seek time, settle time and the latency

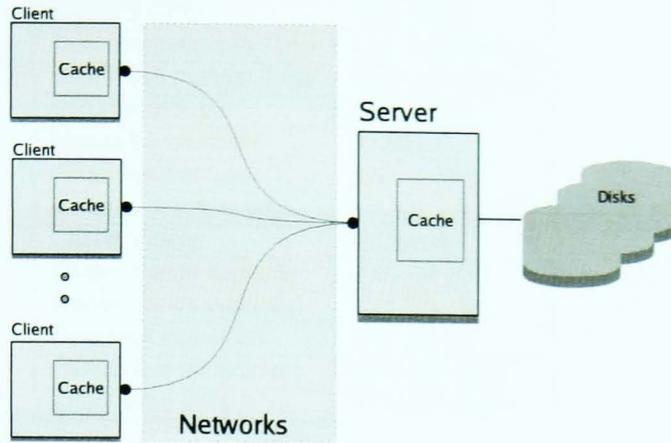


Figure 4.3: Client-server system

cache and writing an item into the cache.

4. **Network.** Network is a medium for transferring messages between client and server. Two types of cost are associated to the network: *fixed* cost and *variable* cost. The fixed cost is the cost at client and at the server, covering both the CPU and the network controller, and its value is assumed to be a fixed number of instructions. The variable cost is the cost per byte of message transferred, and the value is calculated based on the network bandwidth defined in Millions of Bits per Second (Mbps).

The values of the system parameters are listed in Table 4.1. The disk read access time is set one millisecond less than the disk write access time [Koz00]. The values of network parameters, i.e. the bandwidth and the fixed and variable network costs, are adopted from the previous study [OVU98].

Parameter	Value
Client's CPU	500 MIPS
Server's CPU	1000 MIPS
Disk Read Access Time	13.3 milliseconds
Disk Write Access Time	14.3 milliseconds
CPU for disk I/O	5000 cycles
Network bandwidth	10 Mbps
Fixed network cost	6000 cycles
Variable network cost	7.17 cycles/byte
Cache lookup	300 cycles
Clients cache capacity	25% DB size
Server cache capacity	50% DB size
Deadlock detection	300 cycles
Client read think time	50 cycles/byte
Client write think time	100 cycles/byte

Table 4.1: The System Parameters

4.3 The database and workload model

The database is modeled as a collection of page identifiers. Each page contains a number of objects, and each object contains a number of attributes.

When a client runs a transaction, the client runs methods on objects, and each method accesses a number of the object's attributes. This is shown in Figure 4.4. We defined two types of methods: Read-Write method and Read-Only method. In a Read-Write method, the transaction readlocks and writelocks attributes within the method, whereas in a Read-Only method the transaction only readlocks the attributes.

```

1  for (i = 0 to Transaction size)
2  {
3    generate OID
4    generate M=Method(OID)
5    for (each attribute A in M)
6    {
7      access (readlock or writelock) A
8    }
9  }

```

Figure 4.4: Transaction Run

The identifier of the objects accessed, and the type of methods run by an object (either Read-Write method or Read-only method) are generated by the workload. To describe the workload in our model, firstly let us recall the workload model in the existing studies, as has also been described in Chapter 2.4.3.

In the existing studies, data locality and data sharing were modeled in the workload. The following is how they were modeled:

- Pages in the database were divided into regions [ALM95] [OVU98][†].
 - Private region, containing pages that are accessed most of the time by a particular client.
 - Shared region, containing pages that are shared by all the clients.
 - Other region: a region outside the Private and Shared regions.

In another study [Fra96] the Private region was called the *hot* region, and the Shared and Other regions were simply called the *cold* regions. Thus, each client was allocated a Hot region and a Cold region. Moreover, the Hot region belonging to a client overlapped with Cold regions belonging to other clients.

How often each region is accessed during a transaction was determined by an access probability for each region. In addition, whether or not a transaction performs Writes on pages in a particular region was determined by a Write probability for pages in that region. Thus, a workload set the access probability value and a write probability value for each region.

The workload that was claimed to be in representative of general database applications was HotCold [Fra96], which is known as Sh/HotCold in other studies [ALM95] [OVU98]. The values of the access probability and the write probability on each region for the workload are shown in Table 4.2.

Thus, the existing studies modeled data locality and data sharing by setting probabilities for access to pages in the Private, Shared and Other regions in each client's database, and by setting a write probability value.

[†]As illustrated in Figure 2.18 in Chapter 2

Study	General workload	P(Access Hot Region)	P(Access Cold Region)	P(Write)
[Fra96]	Hotcold	80%	20%	20%
[ALM95]	Sh/Hotcold	80%	10% on <i>Shared</i> , 10% on <i>Other</i>	5%
[OVU98]	Sh/Hotcold	80%	10% on <i>Shared</i> , 10% on <i>Other</i>	varied

Table 4.2: Probability values in HotCold and Sh/HotCold workloads

In our workload model, we need to extend the existing model so that we can set a workload that affects the characteristics of the protocols when commutativity exploiting method semantics are used to release lock conflicts. We require a way to determine whether or not lock conflict can be released using a methods semantic commutativity.

In the real application, the commutativity relationships between methods are defined in the object schema:

- An object has m methods. Then, we have an $m \times m$ relationships between methods, which is represented by an $m \times m$ matrix
- Some relationships between the methods are semantic commutativity relationships, and we explicitly define the semantic commutativity (SC) relationships in the matrix
- When a lock conflict on an attribute occurs, we check from the matrix whether a commutativity relationship exists between the methods. If a semantic commutativity relationship exists then the conflict can be released.

If we consider a workload model based on the actual object schema, however, we need a specific matrix of relationships between methods to be defined for that schema. As the schema varies in every database application, we are not able to assume a particular schema for the workload.

Therefore, the chosen approach is to decide whether methods have a semantic commutativity relationship randomly based on probability. By setting the probability for whether methods can commute, we can investigate the performance sensitivity when the probability varies. Moreover, we do not need to assume a particular object schema that must be determined in advanced.

The probability of whether or not semantic commutativity exists can vary from 0% to 100%. As an illustration, an object having two methods will have four method relationships (two to the power of two). If one of the relationships is semantic commutativity relationship, with a uniform probability of access to all the methods, then the probability of releasing the conflict due to semantic commutativity will be 25%. An object having three methods and two semantic commutativity relationships has a 22% probability. Again, an object having one method that semantically commutes to itself will have a 100% such probability.

For each workload, we need the following parameters to describe the database and the workload:

- The number of pages in the database and the number of objects within a page. These two parameters will determine the number of objects in the database. As in the preceding description, a transaction accesses a number of objects from the database (shown in the preceding Figure 4.4), and the objects in the database, classified into regions, are shared by the clients to some extent. As a consequence, the smaller the number of objects in the database, the smaller the number of objects that will be shared by the clients, which means higher data contention if the number of clients accessing the database remains constant.
- The number of attributes of an object and the number of attributes accessed within a method. These parameters are needed in our protocol because in our simulation a method accesses a number of attributes during a transaction (as described in the preceding Figure 4.4). The attributes accessed by a method are selected from the available attributes in the object. Therefore, fewer of attributes of an object gives a higher chance of an attribute being accessed, and this means higher data contention if at least the same number of attributes are accessed within the method.

Another parameter in our simulation is a probability value that dictates whether a restarted transaction accesses the same objects and attributes as those before the transaction aborted. When a transaction aborts and restarts, the workload can be that the transaction re-accesses the previously accessed attributes, or that the transaction accesses completely different attributes. The former

corresponds to setting 100% to the probability, while the latter corresponds to setting 0% to the probability. However, setting the probability to 100% could lead to livelock, in which a transaction is always aborted without having a chance to commit. In this simulation, we therefore set the probability to 50%, in that a transaction has a 50% probability of accessing the same attributes as those accessed before the transaction was aborted.

The parameter values of the database and workload model used in our simulation are listed in Table 4.3.

Parameter	Value
Page size	4 Kbytes
Number of pages in database	200 pages
Number of objects in a page	10
Methods per Tx (Tx Length or Size)	20-50
P(run RW methods)	80%
P(commutativity)	0-100%
P(run new transaction)	50%
Total attributes in an object	5
Total attributes run in a method	2 per method

Table 4.3: Database and workload parameters

4.4 Correctness

This section describes how we check the correctness of the simulation.

In the simulation, a number of clients are concurrently running transactions and accessing shared objects at the server. By using the models described, it was not possible to check the correctness by checking the actual value of an attribute because, unlike in a real application, the result in the simulation does not contain value. Instead, we addressed the correctness of the results by assertions put at points where we could predict that a particular state should apply.

Assertions state that a particular condition must apply. If the condition does not apply, an exception will occur. The following is an example of assertions used in the simulations. The notation $\langle PRE \rangle$ denotes a pre-condition, and $\langle POST \rangle$ denotes a post-condition. Modified Buffer at the server stores successfully-validated attributes belonging to transactions that have not

yet committed. Whether or not the modified buffer is empty is important because the server will store the content of the modified buffer onto disk when the transaction commits, and the disk cost will significantly affect the performance. Therefore, in the assertion in Figure 4.5 we want to ensure that upon receiving a validation request, a read-only transaction does not validate any attributes, while a non read-only transaction validates an attribute [‡]. Before calculating the total bytes B of attributes to be put into the modified buffer, we assert a pre-condition that when the server does not detect a write-write conflict when handling a validation request, the transaction must be a read-only transaction, or a non read-only transaction without lock conflict. Then we assert a post-condition that B is not zero in a non read-only transaction but zero in a read-only transaction. Thus, by using this assertion we ensure that recording the disk cost is correctly implemented.

```

1  if (write-write conflict does not occur)
2  {
3      <PRE>  Either Read-Only Transaction
4             or Non Read-Only Transaction
5             without lock conflict
6
7      calculate the total bytes (B) validated
8
9      <POST> B > 0 in non Read-Only Transaction
10            B = 0 in Read-Only Transaction
11
12     THE ASSERTION:
13     if ( (Read-Only Transaction) AND (B > 0) ) {
14         throw Exception
15     }
16     if ( (Non Read-Only Transaction) AND (B == 0) ) {
17         throw Exception
18     }
19
20     put B into Modified Buffer
21 }

```

Figure 4.5: An example of an assertion

Then, during the simulation run, it was often the case that an assertion failed. The failed asser-

[‡]A read-only transaction validates at the end of a transaction in the Commit-time Validation (CV) protocol, whereas a non Read-Only transaction validates at the end of a method in Method-time Validation protocols (SMV and AMV). This is explained in Chapter 3.

tion might lead us to identify a fault in the implementation, or identify new issues in the protocol design. A revised design or implementation might lead to more assertions being added. We found that unexpected states were even more common in the asynchronous protocol (Asynchronous Method Validation). After going through many fail-and-revise cycles, when there were no failed assertion, we got more confidence on the correctness of the simulation implementation.

In addition, we had to ensure that the simulation results (graphs) were correct. It was often the case that a set of results were obtained but turned out to be flawed. This was because one performance metric measured did not tally with the other performance metrics measured. It was often the case that this led us to correct the implementation or revise the implementation design. To ensure that we obtained sensible results, we measured more performance metrics that were needed to support the analysis, such as measuring a performance metric that was a component of another performance metric. Assertions were again used.

4.5 The limitations of the model

In our model, a transaction contains accesses on a number of methods. At each method a validation message is sent to the server. The drawback is that it is unable to detect the waiting time under a certain situation. This is better explained by the following example. Supposed that T_1 and T_2 have a sequence of interleaving operations as shown in Figure 4.6. The overall sequence of operations is as follows:

$$T_1(o_2.m_1), T_1(o_3.m_2), T_1(o_1.m_4), T_2(o_1.m_1), \dots$$

Notice from the figure (Figure 4.6) that the sequence $T_1(o_2.m_1), T_1(o_3.m_2), T_1(o_1.m_4)$ is invoked within method $T_1(o_1.m_3)$. Supposed that in o_1 (object 1) a semantic commutativity relationship occurs between methods $o_1.m_3$ and $o_1.m_1$, and so when T_2 is validating $o_1.m_1$ the server detects a Write-Write conflict with $o_1.m_4$ but does not detect the conflict with $o_1.m_3$ because $o_1.m_1$ and $o_1.m_4$ does not commute while $o_1.m_1$ and $o_1.m_3$ commutes. In this situation, T_2 can proceed but it needs to wait until $o_1.m_3$ by T_1 has ended. The execution of $o_1.m_3$ involves other subsequent methods

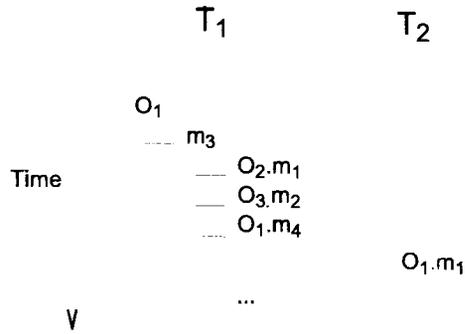


Figure 4.6: Example to illustrate the model limitation

after the call of method $o_1.m_4$ within the method $o_1.m_3$. Thus, the amount of time T_2 has to wait includes the time to execute subsequent methods after method $o_1.m_4$ within the method $o_1.m_3$. As our workload model does not assume a particular object schema, the waiting time by transaction T_2 is not identified.

4.6 Summary

Our Java simulation package used models that include the system model, the database model and the workload model. The correctness of the simulation was checked using assertions. We also described the limitation of our models under a certain situation. The next chapter contains the performance measurement of the protocols based on the models.

Chapter 5

Results and Analysis

In this chapter we investigate the performance characteristics of our protocols using simulation. The following are the main points:

- We implement two protocols that were described in the previous work[Fra96]: Optimistic Two-Phase Locking (O2PL) and Callback Locking (CBL), and compare their performance. The measurement in our implementation results in similar characteristics to those in the previous work[Fra96]. By implementing the two protocols described in the previous work and comparing the relative performance with that given in the earlier work, we demonstrate that our simulator is reasonable.
- Secondly, we compare the performance of our CV protocol with that of O2PL, which tends to be the best performing of the earlier protocols[Fra96], and find that they are of comparable performance.
- We investigate the performance of SMV and AMV with respect to that of CV in two steps:
 - First, we measure the performance of the protocols in their basic form and show that SMV and AMV can outperform CV under common workload.
 - Finally, we investigate what performance improvement might be expected if we were to implement a scheme for exploiting semantic relationships between methods. We do

this by assuming some probability that a lock conflict may be released through some semantic relationship between the methods. These experiments show that with a high probability of commutativity to release write-write conflicts the improvement on the performance can be significant.

The results obtained will be based on the models described in the preceding chapter (Chapter 4). It should be noted that the values of the simulation results are not to be regarded as absolute, but as relative to the values of the other protocols.

In all of the protocols, the measurements are made under HotCold workload, which gives moderate data contention and has been claimed to be the most common database workload [Fra96][OVU98][ALM95].

The variable as the x-axis in the measurements is the one that varies the level of data contention. Generally, we employ the number of clients as the x-axis, since this indicates the scalability of a protocol under simultaneous access by an increasing number of clients.

In addition, our measurement will investigate the characteristics when the number of operations per transaction varies. The method-time validation protocol is intended to detect conflicts earlier than an optimistic protocol such as CV, and thereby abort transactions that cannot complete early rather than at commit time. Therefore, we investigate their performance under varying number of operations per transaction in order to show that this earlier detection of conflicts can lead to a performance benefit through avoiding wastage of resources.

Furthermore, with respect to investigating what performance improvement might be expected using method semantics, we measure the performance under high data contention i.e. Hicon workload. This is because to get noticeable improvement of performance, the number of attribute-level lock conflicts should be high, which is when the data contention is high.

5.1 The metrics

The main performance metric that will be measured is Throughput, which is the number of transactions that can commit per second. In addition, to understand the result, we will measure other performance metrics and checks whether one metric explains another. The following is the description of the metrics that may be included in our measurements:

- **Throughput.** This is the number of transactions that commit per unit of time. Here, one unit of time is equivalent to one second. It is measured as the number of transactions that commit throughout the entire simulation, divided by the total simulation time. Throughput is generally regarded as the main indicator of the superiority among concurrency protocols.
- **Average response time.** Response time is the overall time measured from the start of the transaction until the commit of the transaction. This also includes the time that the transaction aborts and restarts. It is measured by accumulating the time from the start until the commit time, of every transaction, and dividing it by the number of committed transactions. This metric is important to users (i.e. a user-centered metric), as it tells the time a user's transaction needs to be able to commit. The response time consists of all the overheads of clients, servers, disks and networks. In our study, we measure the major components of a response time, which are **average validation time** and **average fetching time**.
 - **Average validation time.** This is the average time needed by a client to perform all validations in each transaction. This metric is measured to understand the component of the average response time. A single validation time is measured as the time since sending a validation until getting the result of the validation.
 - **Average fetching time.** This metric is the time needed by a client to fetch attribute or page from the server. A fetching time is measured since sending a request for a page or attribute until receiving it.
- **Abort rate.** This is the number of aborts that a transaction experiences before committing. It

is measured by counting the number of aborts experienced by all the transactions and divide it by the number of committed transactions. Furthermore, we investigate the components of the abort rate by measuring the aborts due to deadlocks and the aborts due to fail validations. These metrics are also user-centered metrics as some applications considers abort rate as important, for example highly interactive applications cannot tolerate high abort rate [OVU98]. In addition, abort rate is a key metric that can explain the response time because high abort rate usually causes high response time.

- **Releases of lock conflicts per commit.** This metric tells the number of lock conflicts that are released due to method commutativity. This metric shows the frequency of the releases of conflicts due to method commutativity. It is measured by counting the number of releases of lock conflicts using method commutativity and divide it by the number of committed transactions. For further details, we measure the releases of read-write conflicts and the releases of write-write conflicts.
- **Disk read.** This is the number of disk reads performed in a transaction. As previously mentioned, a disk read is performed at the server when the server is sending a database item to a client because the item is not cached by the client. It is measured by totaling the number of disk reads and then divides it by the total number of committed transactions. This metric is important as disk is a dominant overhead.

5.2 Our O2PL implementation

In this section we ensure that our simulator is reasonable, implementation of Optimistic Two-phase Locking, which is the optimistic avoidance-based page-locking protocol, can represent the one in the previous work [Fra96]. The purpose of this is to allow the O2PL to be compared with our Commit-time Validation (CV), described in the next section.

To ensure whether our implementation of O2PL represents the one in the previous study, we compare O2PL with Callback Locking (CBL) that is the pessimistic avoidance-based page-locking

protocol in the previous study [Fra96]. By comparing them and achieve the same performance characteristics in the previous study, we ensure that our simulator is reasonable.

In the previous study [Fra96] the O2PL and CBL have some variants. The O2PL that we implement here is O2PL-i, which stands for Optimistic Two-Phase Locking by invalidation, in which the consistency action is invalidating/removing stale pages from client's cache. The CBL that we implement is CBL-R, which stands for Callback Locking - Read, which has the same way of consistency action as that in O2PL-i.

The following describe our measurement of the existing O2PL and CBL.

The measurement uses the parameters shown in Table 5.1. The parameters are similar with those in the previous study [Fra96].

Some of the settings, however, differ from the ones used in the previous work[Fra96] as listed in Table 5.2. First, the HotCold adopted is the one with a Shared region used in another work [OVU98], as described in Section 2.4.3. We believe it is a more reasonable HotCold setting and so it is used for all the measurements in our simulation. Secondly, for the purpose of simplicity in the implementation, in our deadlock detection algorithm, the building of a wait-for-graph is performed at the server whenever a deadlock can potentially occur. It differs from the previous work[Fra96], in which a wait-for-graph is built locally at each client and collected by the server periodically. We believe that this does not give significant effect on the result.

Parameter	Value
No. of Clients	3-25
Client's Cache Size	25% DB
Server's Cache Size	50% DB
Pages in Database	1300
P(Write)	20%
Transaction size	20 pages
Disk read and write	20 milisecond
Fixed message cost	20000 instr
Variable message cost	2.44 cycles/byte
Network speed	8 Mbps

Table 5.1: Parameter Values for O2PL vs CBL

The result is shown in Figure 5.1. The result shows that the performance characteristics are

[Fra96]	Our simulation
HOTCOLD: 80 percent on Private, 20 percent on Other	HOTCOLD: 80 percent on Private, 10 percent on Other, 10 percent on Shared
Deadlock detection is performed by the server using wait-for-graphs collected from all clients periodically	Deadlock detection is performed by the server using a wait-for-graph built by the server whenever a deadlock may potentially occur.

Table 5.2: The differences in the simulation settings

similar with those in the previous work[Fra96]. First, their throughput increases, reaching a peak at 10 clients, and then declines. This characteristic is due to insufficient server’s cache capacity to accommodate all the pages accessed by more than 10 clients. After 10 clients, least-recently-used pages starts to be removed from the cache and so further accesses on these pages require disk accesses, and this degrades the performance. Secondly, the result is similar with that of the previous work in that O2PL performs better than CBL but the performance of CBL tends to be similar to O2PL at 25 clients.

By achieving the similarity of the characteristics between the protocols in the previous work, we demonstrates that our simulator is reasonable.

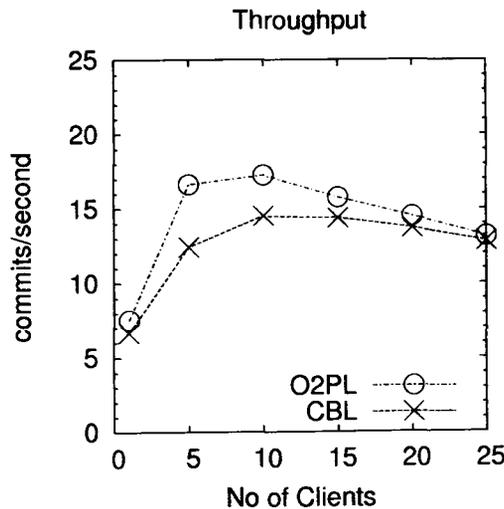


Figure 5.1: O2PL vs CBL

Next we compare the performance of our Commit-time Validation (CV) and the O2PL.

5.3 CV vs O2PL

In this section we compare our Commit-time Validation (CV) with the Optimistic Two-phase Locking (O2PL).

Commit-time Validation (CV) and Optimistic Two Phase Locking (O2PL) [Fra96] protocols are both optimistic, in that a client's transaction runs locally at the client from the start until the commit time. At commit time, the client validates the entire transaction to the server, and the server checks whether the transaction can commit.

Firstly, recall that the difference between CV and O2PL is that CV and O2PL have different granularities of lock. The granularity of a lock in CV is an object's attribute, whereas the granularity of a lock in O2PL is a page.

How a client runs transactions in our simulation is affected by the difference in the granularity of lock. In O2PL, a transaction is a loop over a number of pages accessed, as in Figure 5.2. For example, if the transaction size is 20 operations per transaction, then less than 20 pages are accessed in the transaction. A Write probability dictates whether the transaction takes readlock or writelock on each page.

```
1 for (i = 0 to Transaction Size)
2 {
3   generate a Page P
4   readlock/writelock P
5 }
```

Figure 5.2: Transaction in O2PL

In CV, a client's transaction in this simulation is a loop over a number of object's methods and each method accesses a number of object's attributes. The loop of a transaction is shown in Figure 5.3. For example, if the transaction size is 20 operations per transaction, then 20 methods are accessed; each method is of different object identifier (OID). A write probability dictates whether

a method accessed is Read-Write method or Read-only method. In a Read-Write method, both readlock and writelock are acquired on the attributes accessed by the method (half of the attributes are readlocked and the other half are writelocked).

```
1 for (i = 0 to Transaction size)
2 {
3   generate an OID o
4   generate o.Method M
5   for (each o.attribute A in M)
6   {
7     readlock/writelock A
8   }
9 }
```

Figure 5.3: Transaction in CV

5.3.1 The measurement

The parameters for the measurement are listed in Table 5.3.

The results are shown in Figure 5.4.

The throughput in Figure 5.4(a) shows that O2PL outperforms CV with small numbers of clients but CV outperforms O2PL at large numbers of clients. As the number of clients dictates the level of data contention, the result means that CV loses against O2PL under low data contention workload but wins against O2PL under high data contention workload. This is reasonable because under low data contention workload, the lock conflicts are rare, so that O2PL, which uses page granularity of locking, gets benefit by saving locking overhead. With higher data contention workload, the high number of lock conflicts is better resolved by CV that uses attribute granularity of locking.

Moreover, CV uses attribute-level locking, so that it does not experience *false-sharing* of a page, a condition in which an object cannot be accessed by a transaction because the page of the object is being locked for another transaction's access on another object in that page. Under high data contention, there can be contention on attributes in the same page, so unlike O2PL (that uses page-level locking), CV that uses attribute-level locking cannot experience false-sharing. By being able

Parameter	Value
No. of clients	1-25
Transaction size	20
Write probability	40%
Database size	1300 pages
Method size	2 attributes
n objects in Page	10
Client Cache size	25% DB
Server Cache size	50% DB
Client's CPU	500 MIPS
Server's CPU	1000 MIPS
Disk Read Access Time	13.3 milliseconds
Disk Write Access Time	14.3 milliseconds
CPU for disk I/O	5000 cycles
Network bandwidth	10 Mbps
Fixed network cost	6000 cycles
Variable network cost	7.17 cycles/byte
Cache lookup	300 cycles
Deadlock detection	300 cycles
Client read think time	50 cycles/byte
Client write think time	100 cycles/byte

Table 5.3: Parameters in CV vs O2PL

to prevent false-sharing, CV has less waiting time than O2PL, reducing the potentials for deadlocks under high contention workload.

False-sharing can impact on performance in O2PL when the server receives a request for a database item (page or attribute) from a client, or at commit time when the server receives a validation message from a client:

- When a client requests for a database item (a page or attribute) from the server but it is being writelocked at the server, a read-write conflict occurs. Unlike in CV, in O2PL false-sharing can occur and the client needs to wait until the writelock on the page is released.
- At commit time, a client validates the entire transaction and waits for the result. Following a validation process at the server, if consistency actions are needed, the server sends consistency messages to other remote clients and waits for the result of the consistency actions, so that the faster the remote clients can respond the lower the waiting time. In CV, under high contention workload, with attribute-level granularity of locking, false-sharing can be avoided and so

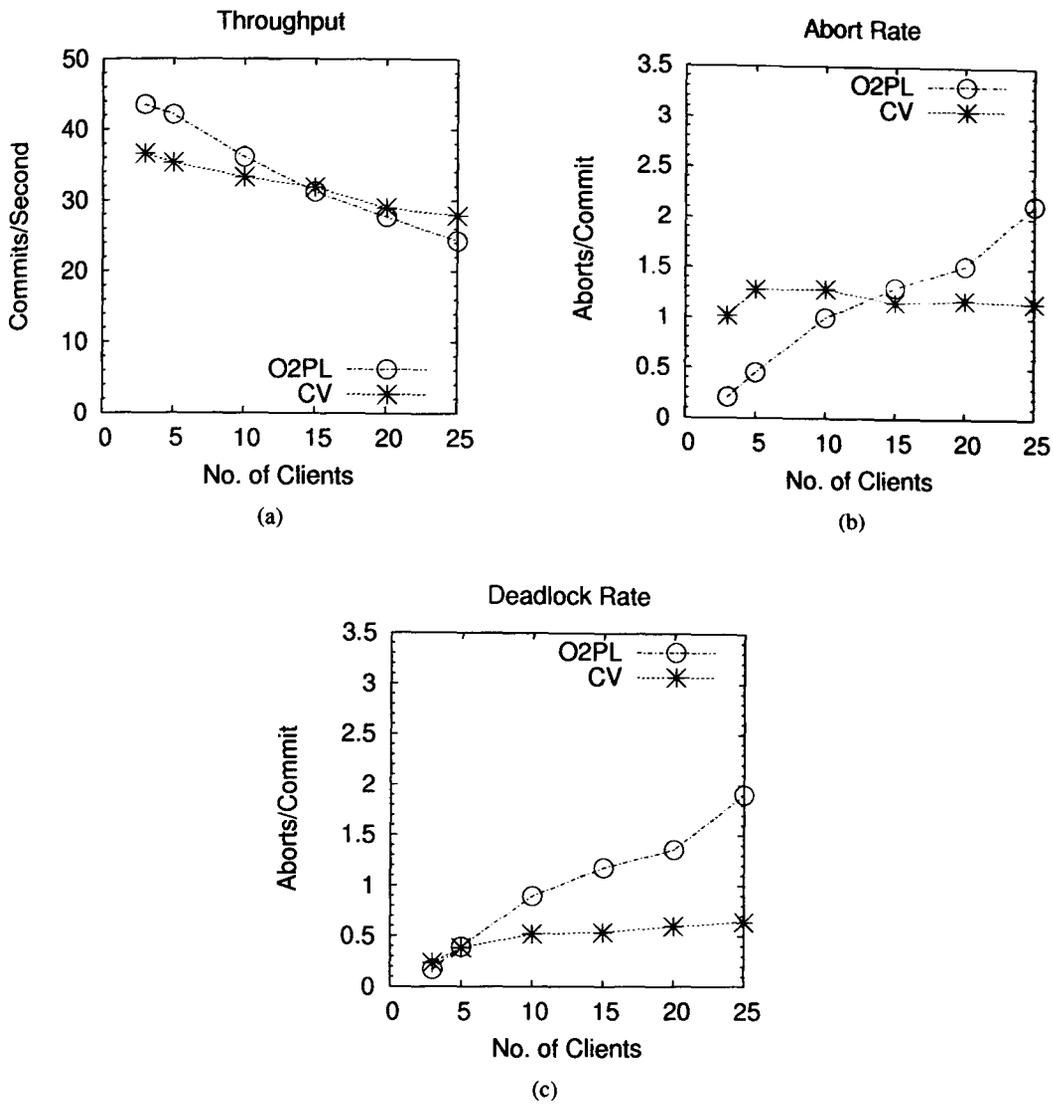


Figure 5.4: CV vs O2PL

a consistency action can be better handled by a client, reducing the waiting time and the potential of deadlocks.

The lower potential of deadlock in CV than in O2PL is supported by the abort rate result in Figure 5.4(b). The abort rate in CV is relatively steady compared to O2PL with increasing number of clients. The abort rate components in Figure 5.4(c) shows that, unlike in CV, the aborts in O2PL

are dominated almost entirely by deadlocks.

5.3.2 Summary

In this section we measured the performance between Commit-time Validation (CV) and Optimistic Two Phase Locking (O2PL). The result shows that CV is better than O2PL under large number of clients i.e. high data contention workload, but is worse than O2PL under small number of clients i.e. low data contention workload. The difference between the performance, however, is not significant under a range of data contention even though other characteristics differ. In this sense, CV and O2PL can be optimistic protocols of comparable performance. Next we compare our method-time validation protocols (i.e. SMV and AMV) with CV.

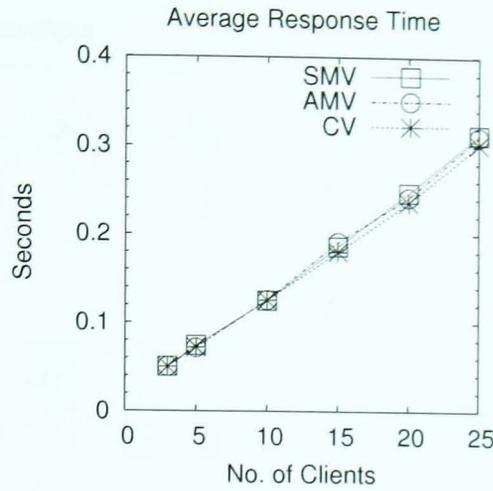
5.4 SMV, AMV and CV

5.4.1 With short-length transactions

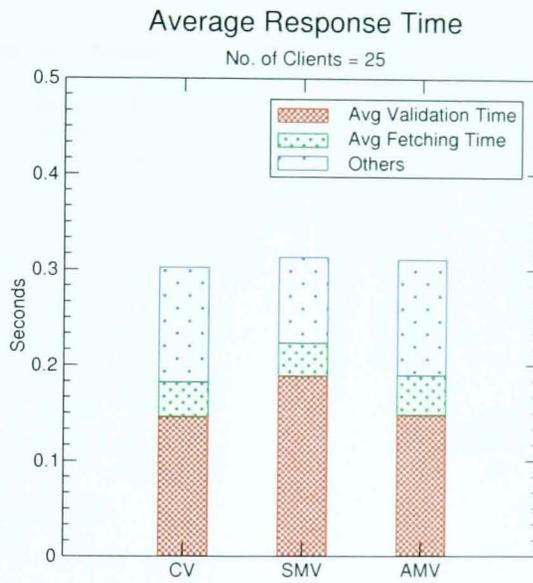
Figures 5.5 and 5.6 show the performance of the method-level validation protocols under the Hot-Cold workload. Under short transactions, in general the differences between the three protocols are not significant, while the response time of CV (Commit-time Validation) is slightly better (i.e. lower) than SMV and AMV, as shown in figure 5.5.

The validation time of SMV is the highest among the others as shown in figure 5.5(b). This is reasonable since SMV experiences more blocking time for validations than in the other protocols. Moreover, the fetching times are about the same for all the protocols. Considering the similar number of disk reads in all protocols and the fact that there is little difference in the abort rate among the protocols, it is reasonable to expect that they all have a quite similar fetching time. For short transactions, the number of disk reads is considerably smaller because short transactions do not require a significant number of items to be fetched from disk.

Referring to Figure 5.6(b), it is interesting to observe that under short transactions, the abort rate in SMV is lower than the abort rate in AMV, while under longer transactions it is the other way



(a)



(b)

Figure 5.5: The average response time; HotCold; Short-length Tx

around (compare figure 5.6(b) and 5.8(b)). Therefore, we were interested in further investigation on the variations in the performance characteristics under different transaction lengths, as will be described in the next subsection.

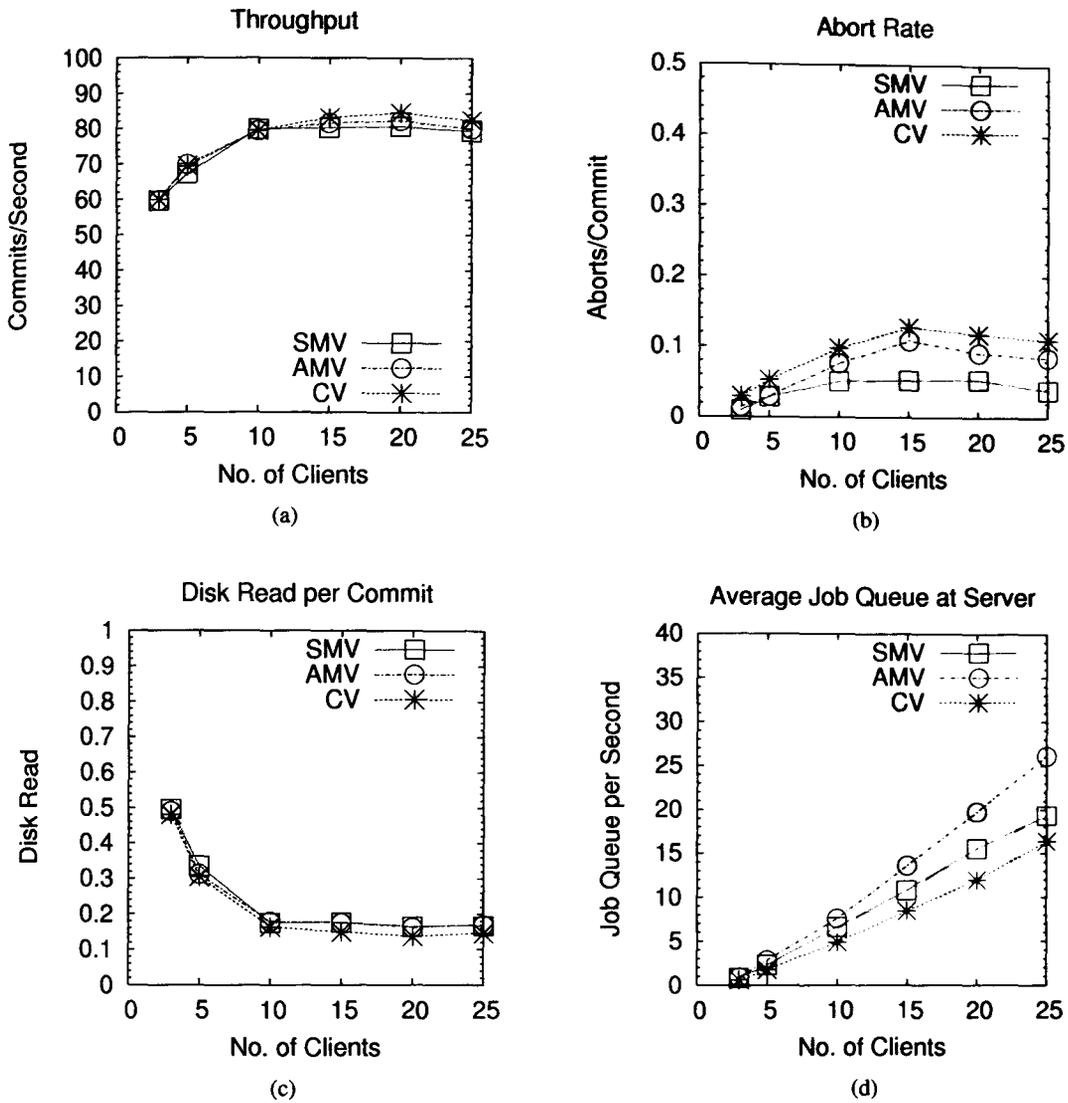


Figure 5.6: HotCold; Short-length Tx;

5.4.2 With medium-length transactions

The results under HotCold and medium transactions can be seen in Figures 5.7 and 5.8. In general, the average response time of the SMV (Synchronous Method-time Validation) and AMV (Asynchronous Method-time Validation) protocols are lower, i.e. better, than in the CV (Commit-time Validation) protocol as shown in the figure 5.7(a). Correspondingly, Figure 5.8(a) shows that the

throughput of SMV and AMV is higher than in the CV protocol.

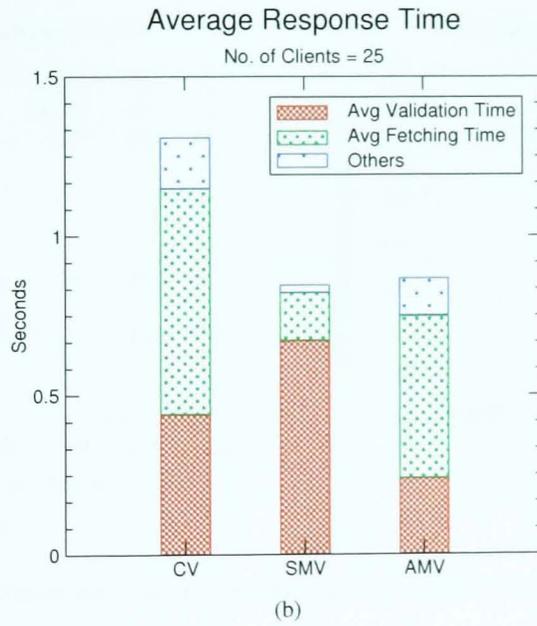
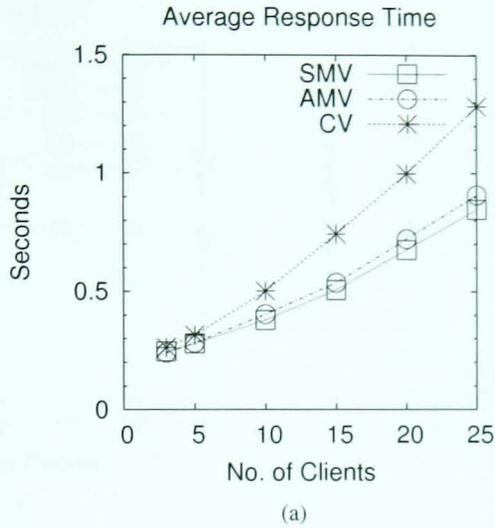


Figure 5.7: The average response time; HotCold; Medium-length; No method semantics

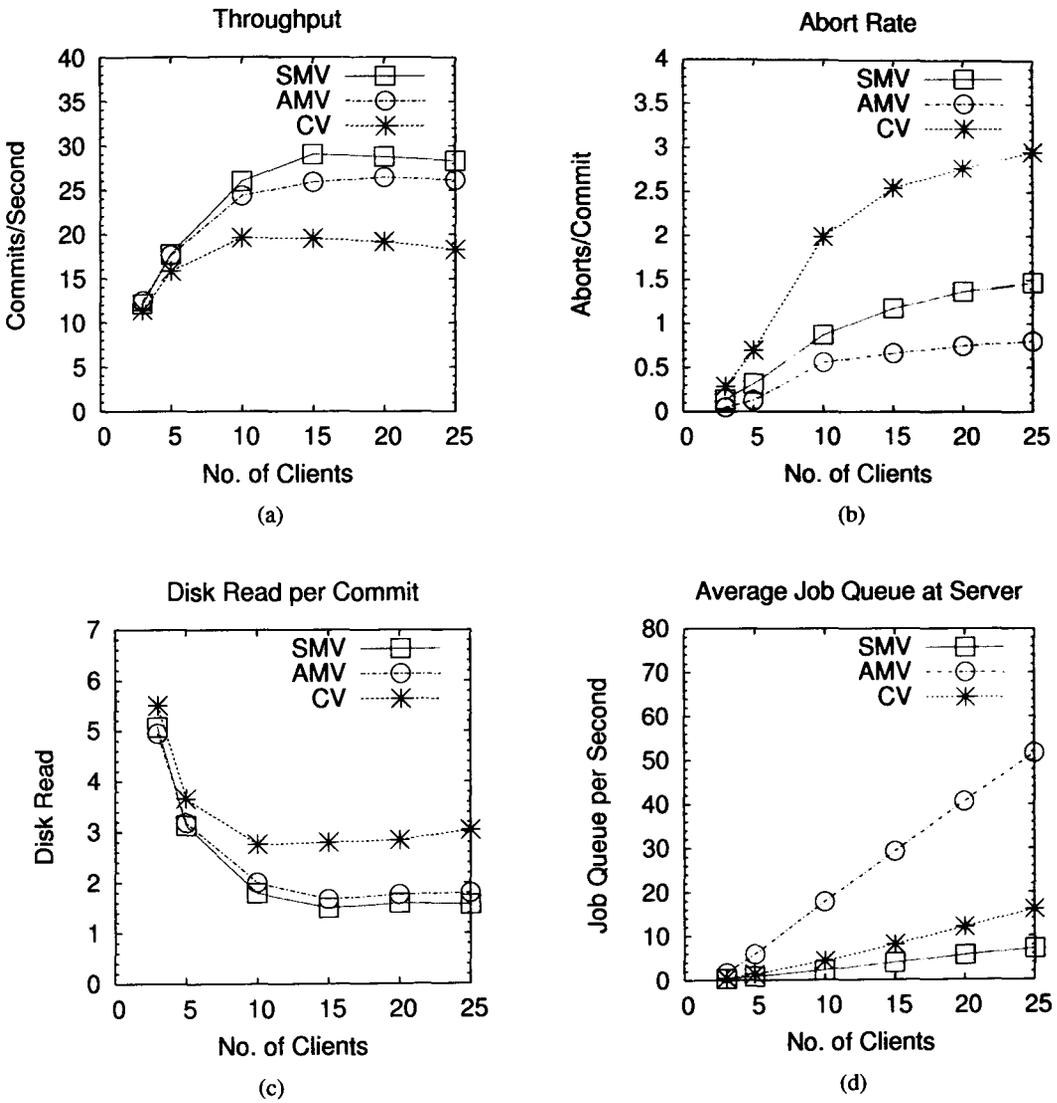


Figure 5.8: HotCold; Medium-length; No method semantics

The average response time

To help explain this result, the major contributors to response time, namely average validation time and average fetching time, are shown in the particular case where the number of clients is 25, in Figure 5.7(b). It can be seen from the figure that the average validation time in CV is lower than that in SMV. This is reasonable because CV only validates once at the end of a transaction, whereas SMV validates at each method accessed. However, it can be seen from the figure that the average fetching time of CV is significantly higher than that in SMV. Referring to the disk reads per commit shown in Figure 5.8(c), it is clear that the high fetching time in CV is caused by the high number of disk reads. This is due to the fact that CV experiences a higher abort rate* than the other protocols as can be seen in Figure 5.8(b). With a higher number of aborts, more database items will be accessed in total, so that more disk accesses will be required, increasing the total fetching time.

With regards to AMV (Asynchronous Method-time Validation), it can be seen from Figure 5.7(b) that its validation time is the lowest compared to the other protocols. Compared to SMV, this is obviously reasonable as AMV is asynchronous (without blocking) when sending validation messages. However, it is surprising that the validation time of AMV is still lower than CV which validates only at commit time. This is caused by the significantly lower abort rate in AMV than in CV. It can be seen from Figure 5.8(b) that the abort rate of CV is significantly higher than in AMV. With the higher abort rate in CV, the number of restarted transactions is higher, resulting in its total validation time exceeding that in AMV.

In terms of the fetching time, however, AMV is higher than SMV, and is almost as high as CV. The cause of the high fetching time is not that AMV requires more disk reads, as the disk read count in AMV is almost similar to that in SMV, as shown in Figure 5.8(c). The fact that AMV causes a significant job queue at the server, as can be seen from Figure 5.8(d) suggests that the high fetching time in AMV is caused by the longer queue for attribute or page requests to be processed by the server. Compared to the other protocols, in AMV the number of messages sent by the server is larger, due to it sending explicit cache consistency messages and explicit abort messages to the

*Abort rate is the number of aborts per committed transaction

clients. The other protocols (SMV and CV), on the other hand, allow the server to piggyback their messages to the clients onto the other messages. As a result, in AMV more queuing time is needed at the server than in SMV and CV, so increasing the fetching time.

The overall result is that the response time of AMV is lower than CV, and slightly higher than SMV. This is also reflected in AMV's throughput (figure 5.8(a)) that is higher than CV, and slightly lower than SMV.

The preceding results suggest that under HotCold workloads, protocols with lower abort rates have better performance. This, however, applies only if the ways messages are sent in the protocols being compared are similar. For example, the lower abort rate of AMV does not help in increasing its performance because the greater number of messages sent by the server has a crucial effect on the performance by creating a longer job queue at the server. A possible workaround for this messaging problem is to separate the process for sending messages and for handling requests, while also enabling them to deal with shared data.

The abort rate

The results have suggested that abort rate is the key contributor to performance. Therefore, it is important to further understand the aspects that influence the abort rate. Abort rate is a metric that indicates the average number of aborts incurred from the start until the commit of a transaction. A transaction is aborted because of one of the following reasons:

1. Failed validations. The validation of a transaction is performed at the server in which lock conflicts are detected. During a validation process, when the validated attribute is being writelocked at the server on behalf of another transaction, a write-write conflict occurs. If the write-write conflicts cannot be resolved by method commutativity, then the transaction is aborted.
2. Deadlocks. A deadlock occurs when a set of transactions are in the waiting state but cannot proceed because they actually wait for each other to complete.

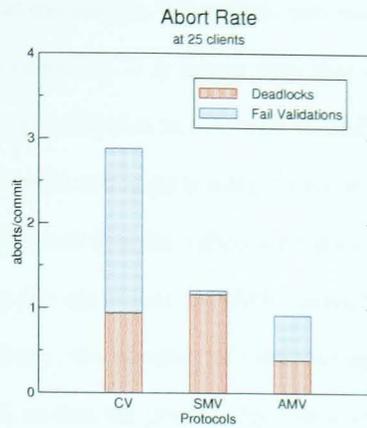


Figure 5.9: Abort rate components. HotCold; Medium-length; No method semantics

To identify the components that constitute the abort rate, the average abort rate due to failed validations and that due to deadlocks were separately measured. The results are shown in Figure 5.4.2 for 25 clients. Comparing SMV and CV, it can be seen that the largest percentage of the aborts in SMV is due to deadlocks, whereas that in CV (the optimistic scheme) is due to failed validations. Therefore, deadlocks occur more frequently in SMV than in CV, and failed validations occur more frequently in CV than in SMV.

Regarding the deadlocks, the reason that more deadlocks are detected in SMV than in CV is because transactions in SMV validate more frequently than transactions in CV. SMV requires a response for every validation at the end of each method, whereas CV only validates once at the end of a transaction.

In terms of failed validations, the reason that they occur more frequently in CV (an optimistic scheme) is that the number of attributes validated at a time in CV is more than that in the SMV protocol. The attributes being validated in CV include all attributes accessed from the start until the end of a transaction, whereas the attributes being validated in SMV are those accessed within a method. Consequently the probability that a conflict occurs in a validation (i.e. failed validations) is higher in CV than that in SMV. Moreover, even if there is no conflict, the consistency actions in CV may involve a larger number of remote clients. While the consistency actions are requested, the

database items are writelocked at the server. Therefore, the number of database items writelocked while waiting for consistency actions in CV is larger than that in SMV, and this results in a higher probability of having conflicts in a validation in CV than in SMV.

With respect to AMV (the asynchronous protocol), it can be seen that the deadlock rate in AMV is lower than that in SMV and CV, and that the failed validation rate in AMV is significantly lower than that in CV. The reasons that the abort rate in AMV (asynchronous) protocol is smaller than in SMV and CV are as follows. Firstly, the number of database items validated on each validation in AMV is the same as that in SMV, so that the probability that a validation fails in AMV is as small as that in SMV. Secondly, AMV does not experience as many blockings as in SMV, because on each validation a transaction in AMV does not wait for a response, so there are fewer deadlocks in AMV than in SMV. Having lower deadlock rate and lower fail validation rate allows AMV to have lower abort rate than SMV and CV.

5.4.3 The effect of variable transaction lengths

The results for the short and medium length transactions described above have highlighted some differences in the characteristics of the proposed protocols for different transaction lengths. Therefore, we conducted further investigations with varying transaction lengths. The results can be seen in Figures 5.10 and 5.11, for 15 clients.

From Figure 5.10(a), it can be observed that the response time in CV (Commit-time Validation) protocol increases significantly when transactions are longer, whereas the increase for SMV and AMV is relatively linear. It can be seen from Figure 5.10(c) that the sharp increase in the response time in CV is caused by the sharp increase in the average fetching time when transactions are longer. This is caused by the high number of disk reads in CV as shown in Figure 5.11(c). The high number of disk reads in CV is due to the abort rate of CV that is relatively much higher than the other protocols, as shown in Figure 5.11(b). With the high abort rate in CV, more transactions are restarted and may require more disk accesses.

The response time results are reflected in the throughput, shown in Figure 5.11(a), in that the throughput of SMV and AMV are markedly better than that of CV for longer transactions.

With respect to the comparison between SMV (Synchronous Method-time Validation) and AMV (Asynchronous Method-time Validation), Figure 5.10(a), 5.10(b) and 5.10(c) show that in SMV the largest percentage of the increase is in the validation time rather than the fetching time. In contrast, in AMV the large percentage of the increase is in the fetching time rather than the validation time. The high increase in the validation time in SMV is more due to the blocking time for sending validations in longer transactions, whereas the high increase in the fetching time in AMV is a result of higher numbers of jobs queuing at the server when transactions are longer, as shown in Figure 5.11(d). The longer job queue could arise from the server processor performing validation actions asynchronously for clients.

Finally, with regards to the abort rate in AMV (Asynchronous Method-time Validation), it can be observed from Figure 5.11(b) that when transactions are longer the increase in the abort rate in AMV is smaller than that of the other protocols. Firstly, apart from at commit time, validation

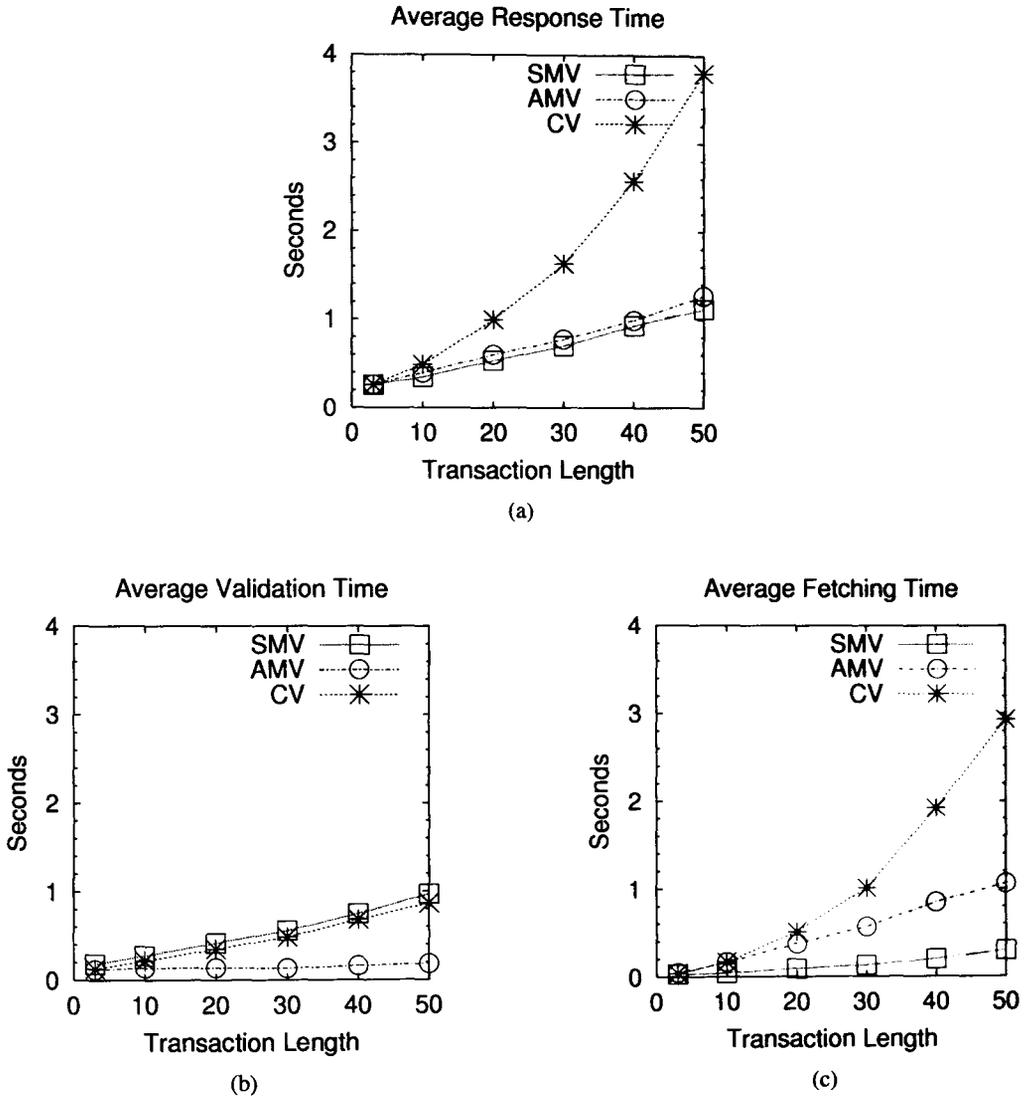


Figure 5.10: The average response time; HotCold; Variable lengths; No method semantics

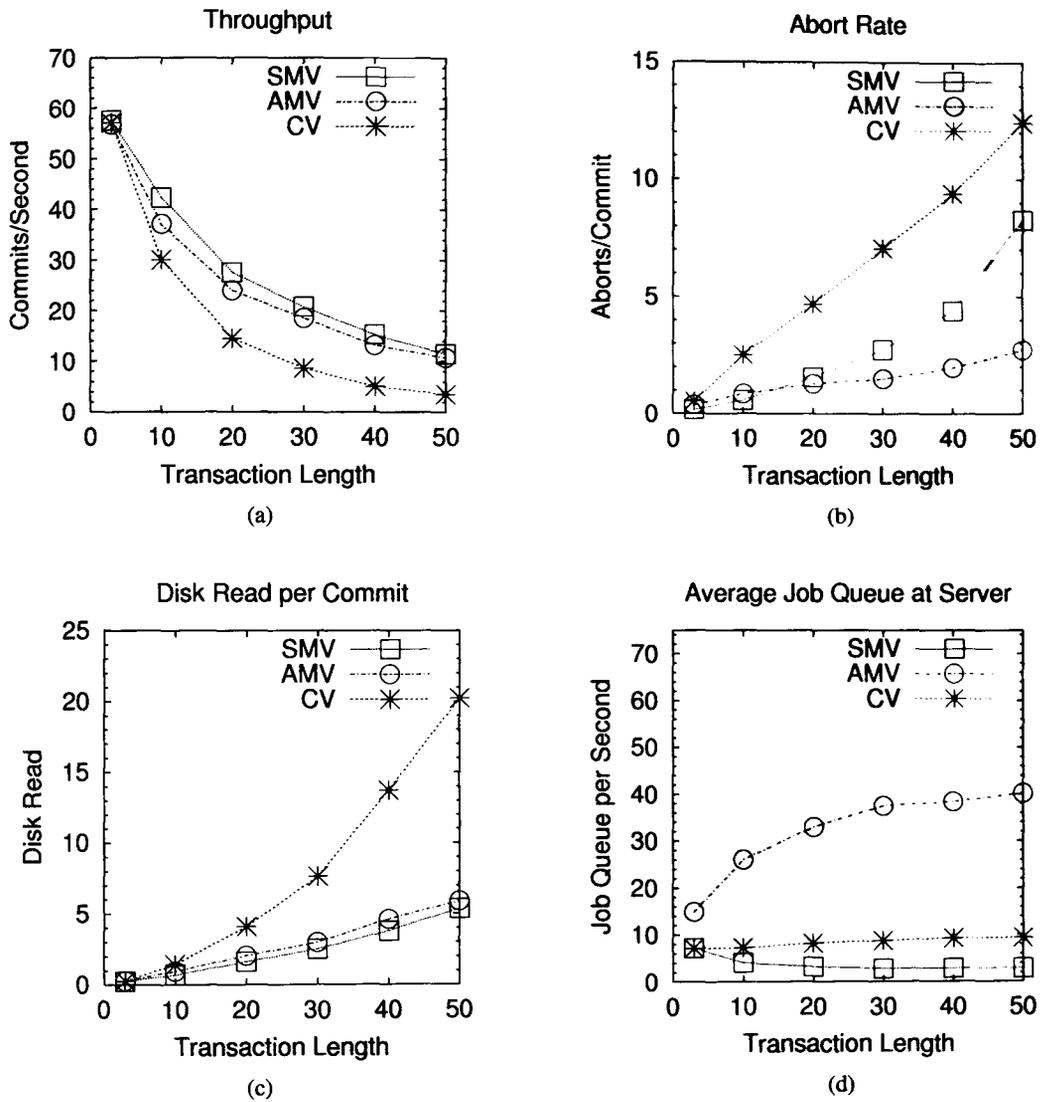


Figure 5.11: HotCold; Variable lengths

requests in AMV are asynchronous and so cannot lead to deadlocks. Secondly, as in SMV the more frequent validation requests lead to fewer failed validations than CV. Therefore, it can be concluded that under longer transactions AMV is better able to maintain a stable abort rate.

5.4.4 Summary

In this section, we compared the performance of SMV, AMV and CV. The result shows that SMV and AMV can outperform CV under moderate data contention workload. In addition, the result shows that the asynchronous protocol i.e. AMV has a better stable abort rate than SMV and CV with increasing number of operations in transaction.

5.5 SMV with a probability of commutativity

This section investigates what performance improvement might be obtained in SMV when method semantics are used in the concurrency control. The measurements compare the performance of SMV when method commutativity is ignored, with its performance when a chosen probability of method commutativity is assumed.

As described in Chapter 3.3.1, the release of lock conflicts by method semantics can be on read-write conflicts and write-write conflicts. The measurements here will vary the commutativity probabilities of each of the conflict (i.e. read-write conflict and write-write conflict).

The workload will firstly be that representing the common database workload which is under the moderate data contention: HotCold, explained in the preceding chapter. Then, we investigate the performance when conflicts are frequent, which is under HiCon workload that generates high data contention. In each workload the observation is on SMV with probability of commutativity to release either read-write conflict or write-write conflict.

5.5.1 Under HotCold

This measurement sets the system parameter values as described in Chapter 4, and sets the workload and database parameters as shown in Table 5.4. The workload is HotCold, with high value of probability to release write-write conflict (80%), and with 100% write probability. In practice, such values may not be commonly found. The purpose here is to get a noticeable observation, on how the characteristics of the performance improvement might be. The other parameter values are taken from the previous measurement.

The result, in Figure 5.12, shows that the probability of commutativity, either to release read-write conflict (Figure 5.12(b)) or to release write-write conflict (Figure 5.12(a)), does not improve the performance. This result is reasonable because HotCold is a moderate data contention workload. Figure 5.12(c) shows that the actual number of releases of write-write conflicts per commit of transaction is very low, scaling to 0.1 that means less than 1 releases per 10 committed transactions.

Releasing Write-Write conflicts

Releasing Read-Write conflicts

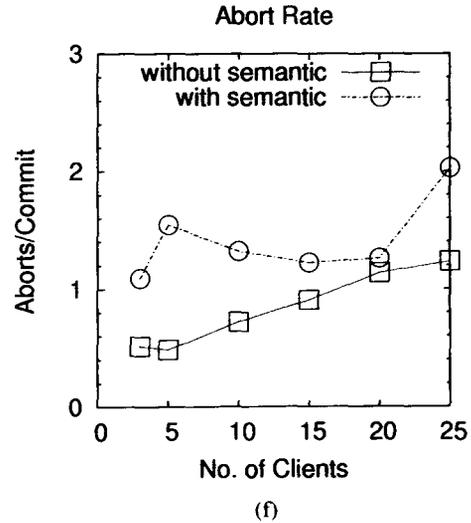
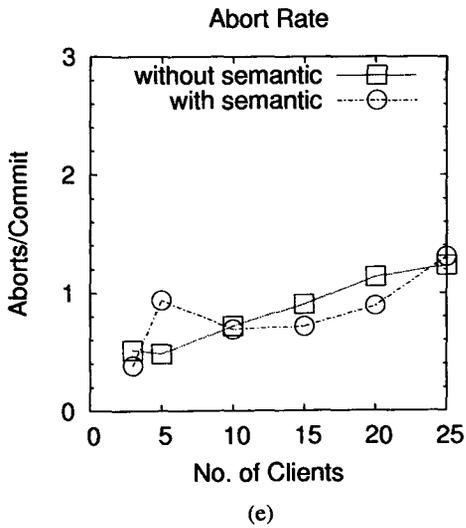
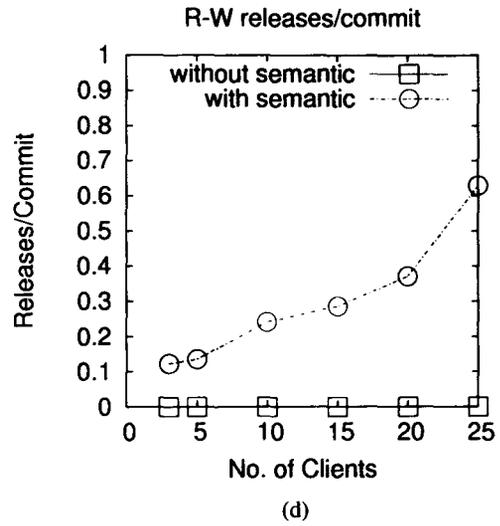
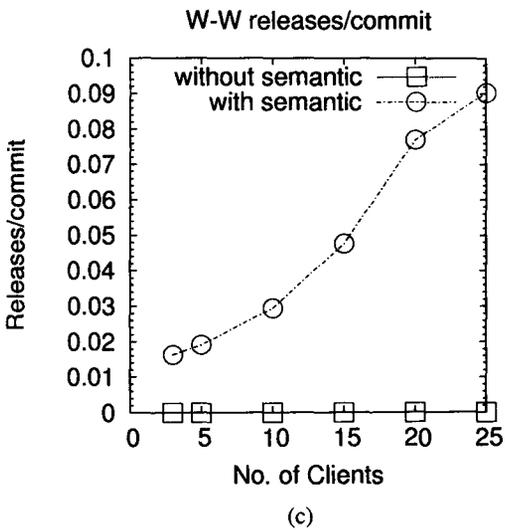
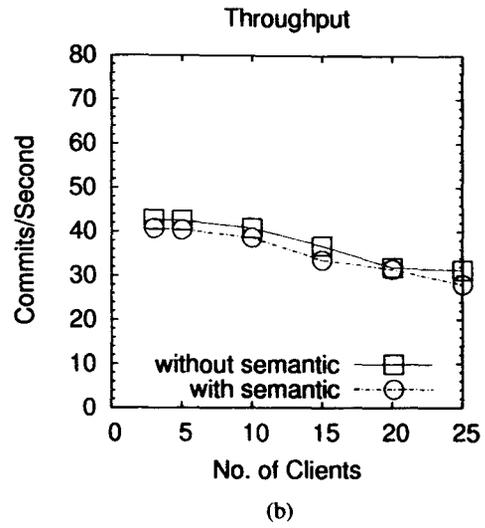
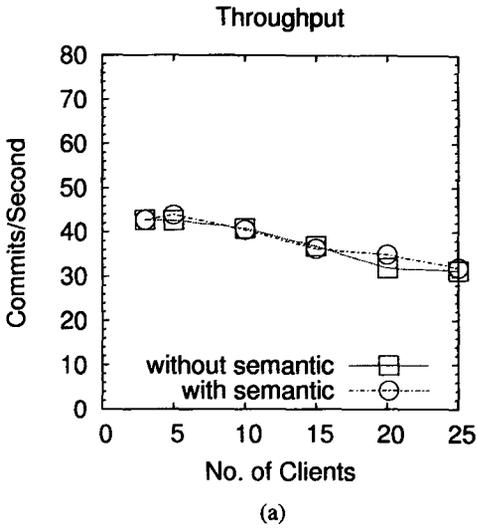


Figure 5.12: Under moderate data contention workload

Parameter	Value
Workload	HOTCOLD
P(read-write commute)	0 and 80%
P(write-write commute)	80 and 0%
P(Write)	100%
Method per Tx (Tx Size)	20
Total Pages in DB	260
Attribute run per method	2
Attribute in an object	5

Table 5.4: Workload and System Parameters

With respect to releasing read-write conflicts, however, the result shows that the number of releases of read-write conflicts per commit of transaction is fairly high, scaling to 1.0 i.e. 1 releases per commit of transaction. However, although it is quite high, the throughput of SMV with semantics is slightly below SMV without semantics. The abort rate explains this.

By investigating the abort rate, it can be seen that the abort rate of SMV with semantics (Figure 5.12(f)) is noticeably higher than SMV without semantics. This is because the aborts are caused entirely by fail validations. A fail validation occurs when a write-write conflict is detected at the server when the server is processing a validation request but the conflict cannot be released by method commutativity. This is illustrated in the scenario explained in Figure 5.13. When a client requests an attribute from the server while the attribute is being writelocked at the server (i.e. a read-write conflict occurs), the read-write conflict can be released due to method commutativity, and so the attribute can be fetched and read by the client although it is being writelocked at the server. If further access on the attribute at the client locally is a write on the attribute, the attribute is then validated by the client to the server. However, when the server receives the validation of the attribute, the attribute is still writelocked by the server, which means a write-write conflict occurs, and consequently the validation fails and the transaction is aborted. From this, we conclude that the use of method commutativity to release read-write conflicts at the server can cause high abort rate due to fail validations, causing a loss rather than a gain of performance.

In the next measurement we investigate how the performance might improve under HiCon, the high data contention workload.

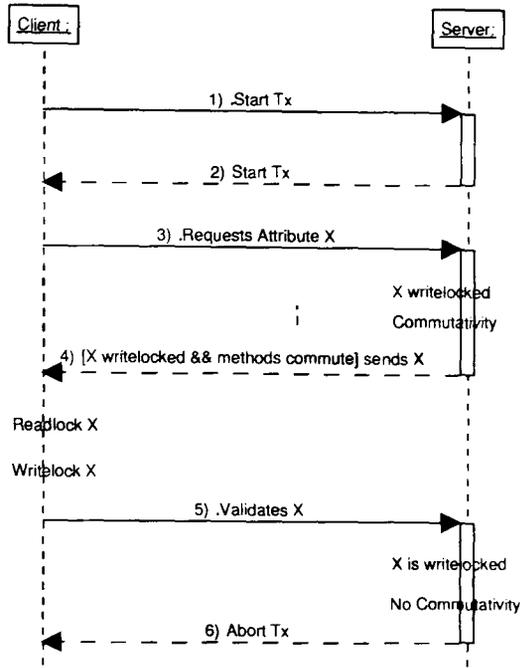


Figure 5.13: A case in SMV

5.5.2 Under high data contention

In this measurement we set the workload to be HiCon, which is a high data contention workload. It uses the same parameters as the previous measurement (i.e. under HotCold), except that the number of operations in each transaction is 5, which is sufficient for HiCon to produce observable results. The parameters are shown in Table 5.5.

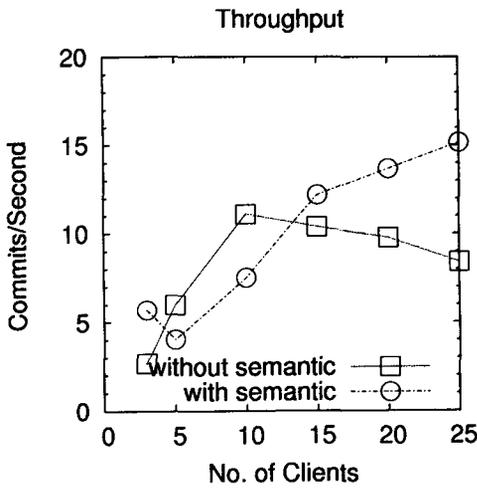
Parameter	Value
Object access pattern	HiCon
No of operations in transaction	5

Table 5.5: Workload and System Parameters

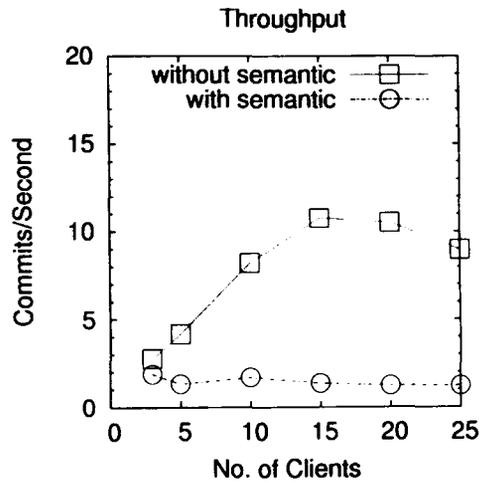
The result, in Figure 5.14(a), shows that releasing write-write conflicts significantly improves the performance where the number of clients is greater than 15. The gain on the performance is due to the increasing number of releases of write-write conflicts per commit of transaction as shown in

Releasing Write-Write conflicts

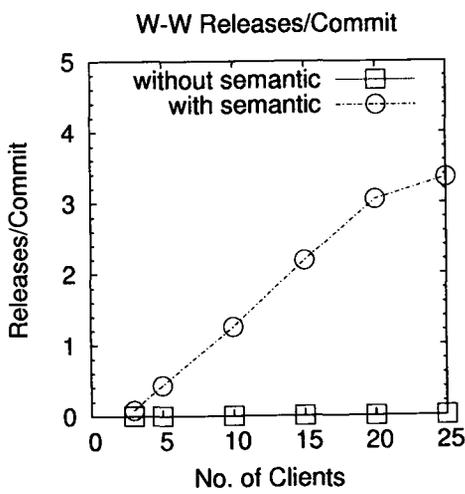
Releasing Read-Write conflicts



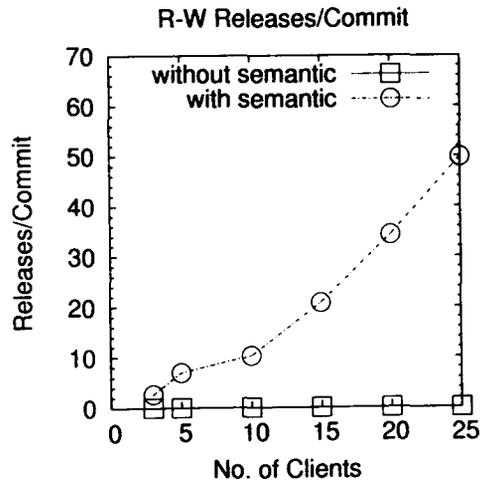
(a)



(b)



(c)



(d)

Figure 5.14: Under high data contention workload

Figure 5.14(c).

By contrast, the result shows that releasing read-write conflicts under high data contention workload causes the performance to drop as shown in Figure 5.14(b). This may be due to the scenario of Figure 5.13 occurring very frequently.

5.5.3 Summary

From the result, we conclude that the use of method commutativity to release lock conflicts under moderate data contention workload does not affect the performance. When its use is to release read-write conflict at the server, the performance may be worse. However, we may expect a performance improvement in SMV by using method commutativity when releasing write-write conflicts under high data contention workload.

5.5.4 Chapter Summary

In this chapter we measure the performance of our method-time validation protocols SMV, AMV and CV. CV is our optimistic protocol that has been verified to be comparable to the existing optimistic protocol O2PL that tends to perform better among other avoidance-based client cache consistency protocols. The measurements suggest that the performance of SMV and AMV outperform CV under moderate data contention workload. The measurements also suggest that the overall performance of SMV and AMV is similar, but the detailed characteristics of the protocols do differ, for instance the result shows that the asynchronous protocol i.e. AMV has a better stable abort rate than SMV and CV with increasing number of operations in transaction.

Moreover, we measure how the improvement of performance might be obtained by SMV that allows method semantics relationship to release lock conflicts. We discover that under moderate data contention workload it does not improve the performance. However, we may expect a performance improvement when releasing write-write conflicts under high data contention workload.

Chapter 6

Conclusions and Further Work

6.1 Conclusions

This study has investigated client cache consistency protocols in which method semantics can be exploited in concurrency control in a client-server, data shipping, object-oriented database system. The motivation of this study comes from the fact that previous studies regarding client cache consistency protocols did not use method semantics for the concurrency control.

Although studies of semantic-based concurrency control have been conducted over the last decade, there has until now been no investigation of their implementation in a client-server environment. Method semantics can exploit the object-oriented database model that allows method calls to enhance performance.

The approach taken in this thesis to utilising method semantics in client cache consistency protocols has been to design, implement and investigate a new protocol where validation is performed by clients to the server at the end of a method call. The protocol studied is named Synchronous Method-time Validation (SMV). The term “synchronous” comes from the behaviour of the protocol, in that a validation message is sent synchronously from clients to the server; the client waits for the response of the validation from the server before continuing (or aborting). This study also investigated a new asynchronous version of the protocol named Asynchronous Method-time Vali-

dation (AMV), which aims to improve performance through less blockings at each validation from client to server.

The study was conducted using simulation, as this allows system parameters to be altered without physical modification of a real system.

In order to investigate their characteristics, the new protocols, i.e. SMV and AMV, were compared with the optimistic version of the Commit-time Validation (CV) protocol that also is attribute-level locking.

In an existing study[Fra96], an optimistic protocol - O2PL - tends to perform better than a pessimistic protocol with the same page-level granularity of locking: Callback Locking (CBL). By comparison, in our study we compared our method-time validation protocols (SMV and AMV) against the optimistic CV, all of which use the attribute-level granularity of locking.

In order to check the characteristics of CV, we compared it with O2PL, and find out that they have comparable performance characteristics. However, as expected, CV can outperform O2PL and vice versa, depending on the level of data contention. CV with attribute-level granularity of locking outperforms O2PL when the data contention is high, whereas O2PL with page-level granularity of locking outperforms CV when the data contention is low.

The investigation also shows that our O2PL implementation behaves as that in the previous work [Fra96]. This was determined by also implementing the pessimistic CBL, and comparing its performance relative to the optimistic O2PL. The results show the same relative performance characteristics as we found in the previous work that compared these two protocols that both use page-level granularity locking.

Having demonstrated that our simulation is reasonable, including checking the characteristics of CV, we compared our new method-time validation protocols (SMV and AMV) with CV. We carried out two major comparisons. First, the SMV, AMV and CV are compared in their basic form, without any probability of commutativity. Secondly, we investigate what performance improvement might be achieved if we implement a scheme for exploiting semantic relationships between methods. We do this by assuming that there is some finite probability that a lock conflict may be released in SMV

through exploiting a semantic relationship between the methods.

With regards to the comparison between SMV, AMV without exploiting method semantics and CV, this performance study has produced the following findings:

1. The results show that the SMV (synchronous) protocol can outperform the CV (optimistic) protocol under common database workloads. The main cause of this is that SMV experiences a lower abort rate than CV. The lower abort rate results in better server responsiveness, and consequently a better response time for clients. The low abort rate in SMV is caused by the fact that failed validations in SMV are significantly fewer than that in CV although there are more deadlocks in SMV than in CV due to more blockings. Overall in SMV, the reduction in the abort rate due to failed validations far exceeds the increase in the abort rate due to deadlocks. Having fewer database items validated in SMV than in CV allows SMV to have a lower chance of failed validations. Moreover, having fewer database items validated in SMV also means fewer consistency actions have to be performed by the server, and this results in a shorter writelocking time at the server, reducing the probability of failed validations.
2. We also investigated the asynchronous version of the protocol named Asynchronous Method-time Validation (AMV). The results show that the AMV has an even lower abort rate than the SMV (synchronous) and the CV (optimistic) protocols. Fewer deadlocks in AMV than in SMV, and fewer failed validations in AMV than in CV allowed AMV to have a lower abort rate than SMV and CV. The fewer deadlocks in AMV are caused by the fact that AMV does not have as many blockings as in SMV, whereas the fewer failed validations in AMV are due to the same number of attributes in a validation message as in SMV.
3. The measurements for varying transaction lengths show that SMV performs better for longer transactions than does the CV (optimistic) protocol. The longer the transactions, the greater the difference in the abort rate between SMV and CV. This is because as the transaction length increases, the chances of having failed validations in SMV becomes even smaller than that in CV. Because the reduction in the abort rate due to failed validations exceeds the increase

due to deadlocks, the abort rate in SMV is smaller than that in CV, for longer transactions. Because the abort rate influences the number of disk reads in a transaction, and these are significant for performance, SMV performs better for longer transactions. Furthermore, the results show that for longer transactions the abort rate of AMV is more stable than for SMV. This is because the longer the transactions the lower are the chances of deadlocks in AMV than in SMV.

Finally we investigated the performance improvement that may be expected if we exploit the semantic relationships between methods in our SMV protocol. The results show that under moderate data contention workloads the use of method commutativity to release lock conflicts does not improve performance. In fact, when it is used to release read-write conflicts at the server, the performance may be worse. However, there was a performance improvement when using method commutativity to release write-write conflicts under high data contention workload.

Therefore, overall we believe that this work has shown that the new protocols can have advantages over existing protocols for particular, realistic workloads. The basic synchronous and asynchronous method-level protocols can outperform existing optimistic protocols, while there are circumstances in which exploiting method semantics delivers better performance. However, interestingly, exploiting method semantics can reduce performance in some cases.

6.2 Further work

In this section, we identify further work that builds on this study.

Firstly, the results in this study are based on simulation, with the parameter and modeling limitation described in Chapter 4. A more accurate analysis of the protocols could be obtained by implementing the protocols in a real database system and then running and measuring real workloads.

The implementation of our asynchronous protocol AMV with its basic form, i.e. without method semantics support, is more complex than that in SMV, and it will need extra overhead if we implement AMV that can exploit method semantics. However, our results shows that AMV is potential in reducing abort rate. Our results also show that there are circumstances in which exploiting method semantics can deliver better performance. Therefore, despite having to meet additional complexity, further exploration of AMV will be a useful and interesting work.

This study used sequence diagrams to design the protocols and simulation to test their behaviour. This approach is appropriate in the sense that simulation allows parameters to be altered without requiring changes to a physical system. However, when tackling some concurrency issues, it would be desirable to have a more formal way to verify the correctness of the protocols. In this study, to test the correctness of the protocols many assertions are added into the simulation implementation code. An assertion allows incorrect states to be detected during the simulation runs. However, it turns out to be difficult to anticipate all transaction states. Often some unidentified states were revealed by finding a failed assertion and investigating the causes. After that, additional assertions were added, and the simulation re-executed, which may then reveal further unidentified states, and so on. This cycle is much more exhaustive when investigating the asynchronous protocol as it has many more unanticipated transaction states. An investigation into the use of a formalism such as petri-nets to perform protocol verification may therefore lead to useful future advances.

Finally, we note that the exploitation of semantic-based concurrency control in a data-shipping scheme is part of a larger area of investigation known as “cooperative transactions” [Kai93][AKT⁺96].

In cooperative transactions each client has a private local database and runs transactions locally. Cooperation between transactions can lead to high concurrency. This is achieved by defining semantics at some level of granularity, such as for a group of objects or at the application level. It would be worthwhile investigating the relationship between the protocols and performance results in this study and those for cooperative transactions.

Bibliography

- [AB04] MySQL AB. Mysql reference manual. <http://dev.mysql.com/doc/>, 2004.
- [Ady99] Atul Adya. *Weak Consistency: A generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.
- [AKT⁺96] K. Aberer, J. Klingemann, T.Tesch, J.Wasch, and E.J.Neuhold. Transaction models supporting cooperative work, the transcoop experiences. *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'96)*, 1996.
- [ALM95] Atul Adya, Barbara Liskov, and U. Maneshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD*, 1995.
- [BBG⁺95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD '95, San Jose, CA USA*, 1995.
- [BCR⁺03] Zohra Bellahsene, Akmal B. Chaudhri, Erhard Rahm, Michael Rys, and Rainer Unland. *Database and XML Technologies, First International XML Database Symposiums, XSym 2003, Berlin, Germany*. Springer-Verlag, 2003.
- [BM96] Arthur J. Bernstein and Phillip M.Lewis. Transaction decomposition using transaction semantics. *Distributed and Parallel Databases, Vol.4, No.1*, 1996.

- [Cat97] R.G.G Cattell. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
- [CB89] N.Goodman C. Beeri, P.A.Bernstein. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230-269, 1989.
- [CB02] Thomas Connolly and Carolyn Begg. *Database Systems, A Practical Approach to Design, Implementation, and Management*. Pearson Education Limited, Essex, 2002.
- [CFLS91] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene Shekita. Data caching tradeoffs in client-server dbms architectures. *Computer Science Technical Report 994, University of Wisconsin-Madison, Computer Sciences Department*, 1991.
- [Cor03] Versant Corporation. Press and media, 2002-2003 press releases. <http://www.versant.com/index.php?module=ContentExpress&func=display&bid=23&bttitle=News%20and%20Events&mid=7&ceid=4>, 2003.
- [DR96] Pedro Diniz and Martin Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *TRCS96-07, University of California, Santa Barbara, Department of Computer Science*, 1996.
- [DW04] Johannes Dwiartanto and Paul Watson. Exploiting method semantics in client cache consistency protocols for object-oriented databases. *Proceeding of the International Conference on Information and Knowledge Engineering, Las Vegas, Nevada, U.S.*, 2004.
- [FCL95] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *University of Maryland College Park Technical Report CS-TR-3511 and UMIACS TR 95-84*, 1995.
- [Fra96] Michael J. Franklin. *Client Data Caching, A Foundation for High Performance Object Database Systems*. Kluwer Academic Publishers, 1996.

- [Gil01] Phillip J. Gill. Placing its bets on java. <http://www.javareport.com>, March 2001.
- [Gmb03] Poet Software GmbH. News and events press releases. http://www.fastobjects.com/eu/F0_EU_NewsEvents_PressReleases_Body.html, 2003.
- [GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, New Jersey, 2000.
- [GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.
- [Gra81] Jim Gray. The transaction concept: Virtues and limitations. *VLDB 1981*, 1981.
- [HM98] Fred Howell and Ross McNab. simjava: a discrete event simulation package for java with applications in computer systems modelling. *First International Conference on Web-based Modelling and Simulation, San Diego CA, Society for Computer Simulation, Jan 1998*, 1998.
- [JG98] Woonchun Jun and Le Gruenwald. Semantic-based concurrency control in object-oriented databases. *Journal of Object-oriented Programming*, 1998.
- [Kai93] Gail E. Kaiser. Cooperative transactions for multi-user environments. *CUCS-006-93*, 1993.
- [Kim95] Won Kim. *Modern Database Systems*. ACM Press, 1995.
- [KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. 1990.
- [KM94] Alfons Kemper and Guido Moerkotte. *Object-oriented Database Management, Applications in Engineering and Computer Science*. Prentice-Hall International, Inc., 1994.

- [Koz00] Charles M. Kozierok. Reference guide - hard disk drives. <http://www.storagereview.com>, 1997-2000.
- [KTW96] Justus Klingemann, Thomas Tesch, and Jurgen Wasch. Semantic-based transaction management for cooperative applications. *Proceedings of the International Workshop on Advanced Transaction Models and Architectures*, 1996.
- [LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communication of the ACM*, 34(10), 1991.
- [MH96] R. McNab and F.W. Howell. Using java for discrete event simulation. *Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)*, Univ. of Edinburgh, 219-228, 1996.
- [Mos85] J. Eliot B. Moss. *Nested Transactions, an Approach to Reliable Distributed Computing*. The MIT Press, 1985.
- [MWBH93] R. Peter Muth, Gerhard Weikum, Peter Brossler, and Christof Hasse. Semantic concurrency control in object-oriented database systems. *9th IEEE International Conference on Data Engineering*, 1993.
- [Nie94] Jacob Nielsen. Response times: The three important limits. <http://www.useit.com/papers/responsetime.html>, 1994.
- [Nie97a] Jacob Nielsen. Loyalty on the web. <http://www.useit.com/alertbox/9708a.html>, 1997.
- [Nie97b] Jacob Nielsen. Report from a 1994 web usability study. http://www.useit.com/papers/1994.web_usability_report.html, 1997.
- [OVU98] M. Tamer Ozsu, Kaladhar Voruganti, and Ronald C. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching dbms. *Proceedings of 24th VLDB Conference, New York, USA*, 1998.

- [Ozs94] M. Tamer Ozsu. Transaction models and transaction management in object-oriented database management systems. *Advances in Object-oriented Database Systems, chap.7, Springer-Verlag, 1994.*
- [PLP97] Ching-Shan Peng, Kwei-Jay Lin, and Tony P.Ng. A performance study of the semantic-based concurrency control protocol in air traffic control systems. *Proc 2nd International Workshop for Real-time Database Systems, 1997.*
- [RAA94] F. Resende, D. Agrawal, and A. El. Abbadi. Semantic locking in object-oriented database systems. *University of California, Santa Barbara, 1994.*
- [RW91] L. Rowe and Y. Wang. Cache consistency and concurrency control in a client/server dbms architecture. *ACM SIGMOD, 1991.*
- [SdSWN00] Jim Smith, Sandra d.F.M. Sampaio, Paul Watson, and N.W.Paton. An architecture for a parallel object database. *In Proceedings of the Workshop on High Performance Object Databases (HIPOD 2000), Cardiff University, Cardiff, UK, 2000.*
- [SKS02] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw Hill, New York, 2002.
- [SWdSN00] Jim A. Smith, Paul Watson, Sandra d.F.M. Sampaio, and N.W.Paton. Polar: An architecture for a parallel odmng compliant object database. *In Proceedings of the 9th International Conference on Information Knowledge Management (CIKM 2000), McLean, Virginia, USA, 2000.*
- [VOU04] Kaladhar Voruganti, M. Tamer Ozsu, and Ronald C. Unrau. An adaptive data-shipping architecture for client caching data management systems. *Distrib. Parallel Databases, 15(2):137–177, 2004.*
- [Wat99] Paul Watson. The design of an odmng compatible parallel object database server. *In*

Proceedings of the 3rd International Conference on Vector and Parallel Processing (VECPAR '98), Porto, Portugal, 1999.

- [WN90] W. Wilkinson and M. Neimat. Maintaining consistency of client cached data. *Proceedings of the 16th International Conference on Very Large Data Bases*. 1990.
- [WSMB95] Z. Wu, R.J. Stroud, K. Moody, and J. Bacon. The design and implementation of a distributed transaction system based on atomic data types. *The British Computer Society, The Institution of Electrical Engineers and IOP Publishing Ltd.*, 1995.
- [X3.92] ANSI X3.135-1992. *American National Standard for Information Systems - Database Language - SQL*. 1992.