Maintaining Consistency in Client-Server Database Systems with Client-Side Caching

Thesis by

Fahren Bukhari

School of Computing Science

In Partial Fulfilment of the Requirements

For the Degree of

Doctor of Philosophy

Newcastle University

Newcastle upon Tyne, UK

2012

Abstract

Caching has been used in client-server database systems to improve the performance of applications. Much of the current work has concentrated on caching techniques at the server side, since the underlying assumption has been that clients are "thin" with application level processing taking place mainly at the server side. There are also a new class of "thick client" applications where clients need to access the database at the server but also perform substantial amount of processing at the client side; here client-side caching is needed to provide good performance for applications.

This thesis presents a transactional cache consistency scheme suitable for systems with client-side caching. The scheme is based on the optimistic approach to concurrency control. The scheme provides serializability for committed transactions. This is in contrast to many modern systems that only provide the snapshot isolation property which is weaker than serializability. A novel feature is that the processing load for validating transactions at commit time is shared between clients and the database server, thereby reducing the load at the server. Read-only transactions can be validated at the client-side, without communicating with the server. Another feature is that the scheme permits disconnected operation, allowing clients with cached objects to work offline.

The performance of the scheme is evaluated using simulation experiments. The experiments demonstrate that for mostly read only transaction load – for which caching is most effective - the scheme outperforms the existing concurrency control scheme with client-side caching considered to be the best, and matches the performance of the widely used scheme that only provides snapshot isolation. The results also show that the scheme in a disconnected environment provides reasonable performance.

Acknowledgements

The results described in this dissertation would not have been possible without the help and support of a number of people. First and foremost, I would like to thank my supervisor, Professor Santosh Shrivastava, for his continuing guidance, encouragement, and support during the past four years.

I would like to thank to the vice rector of Bogor Agricultural University, Professor Yonny Koesmaryono, for his administration and financial support during my study. I acknowledge with thanks to the rector of Bogor Agricultural University, Professor Herry Suhardiyanto, for his encouragement and help. Special thanks go to my colleagues at the Department of Mathematics, Bogor Agricultural University, for their continuous support.

Last but not least, I would like to thank to my family members, Ummu (my wife) and Shazana (my only daughter), for their love, support, and patience. They deserve greater thanks than I can possibly give. I would also like to thank my mother and my mother in law for their continuous support from distance.

This work was financially supported by the scholarship from Directorate General of Higher Education, Ministry of National Education, Indonesia.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

This thesis is concerned with providing good performance along with serializability order for transactions in client-server database systems. Caching techniques have been used extensively in transactional client-server database systems to improve the performance of applications. Much of the current work has concentrated on caching techniques at the server side, since the underlying assumption has been that clients are "thin" with application level processing taking place mainly at the server side (say within the application server). For example, Kossmann et al (2010) describe a typical caching scheme in such systems. There are also new classes of "thick client" applications where clients need to access the database at the server but also perform substantial amount of processing at the client side; here client-side caching is needed to provide good performance for applications.

1.1 Motivation

1.1.1 Server-Side Caching

Client-server database systems are comprised of two logical parts: A server that provides persistent objects and a client that runs applications (see Figure 1). A traditional assumption in the design of client-server database system has been that the client has limited resources and the server has a powerful computer. Accordingly, client functionality has often been restricted to submit read requests and update requests across the network to the server, and to present the received results to the user. In such environment, the application logic computation is installed at the server.

The response time of the server is a critical factor in the performance of the client-server database system. Server resources are shared by many clients. As the number of client increases, the server can become the bottle-neck. Optimizing the performance of the server has been done by many researchers; such as caching objects at the server (Perez-Sorrosal et al., 2011) and allowing transactions to read stale objects (Bernstein et al., 2006).

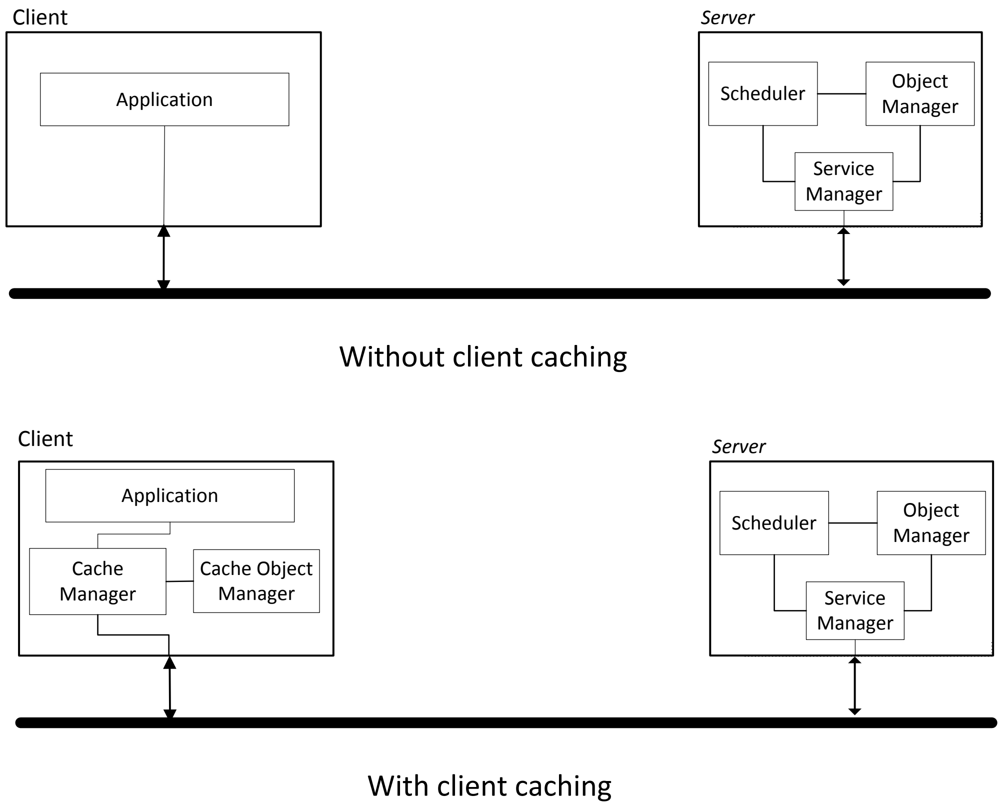


Figure 1: The effect of Client Caching on the Client-Server Structure

1.1.2 Client-Side Caching

With significant advances in computer technology namely, powerful processors and large memories available at low cost, the client functionality has changed. It is possible for business logic to be is installed at client sides and to reduce network latency by caching objects at the client. Now the functionality of such "thick client" is to submit requests to the server for accessing objects only if the objects are not found in client cache and the server functionality is to provide persistent objects. Ideally, the workload of transactional cache consistency scheme should be shared between servers and clients. Client-side caching has been studied in the past (Franklin, et al, 1997). However with the popularity of thick-client applications, it is worthwhile to examine if currently available schemes can be improved.

This thesis proposes an efficient concurrency control scheme for use in such thick-client applications. The scheme is based on the optimistic approach to concurrency control. Basically, in optimistic concurrency control, a transaction makes local copies of the data objects from the database server and performs computations on them; at commit time, the server performs validation check to ensure that these objects have not been modified by some other transactions; if the validation succeeds, the transaction commits, and the modified object copies are written back into the database else the transaction is aborted. Optimistic schemes are attractive in environments with low data contention (transactions are predominantly read only), precisely the environments where data caching would be most effective. Our scheme has several attractive features discussed below.

- It provides serializability for committed transactions; this is in contrast to many popular database management systems that provide snapshot isolation which is a weaker form of consistency than serializability.
- A static read-only transaction (a transaction which predeclares its objects to be read), is never aborted.
- It is deadlock free.
- Read-only transactions can be validated at the client-side, without communicating with the server.
- For update transactions, validation is done partly at the client and partly at the server. The net effect is that the processing load for validation is subtantially reduced at the server, thereby improving scalability.

Our scheme, named Validation Queue (VQ) concurrency control, is based on the optimistic concurrency control scheme named Read Commit Order Concurrency Control (ROCC) introduced by Shi and Perrizo (2002). Traditional optimistic concurrency control methods abort a transaction when the transaction conflicts with other transactions. The ROCC scheme improves on these methods by only aborting a transaction when the execution of the transaction interleaves with the execution of other transactions. Shi and Perrizo also show that their scheme outperforms two-phase locking in centralized database systems under low data contention load (transactions are predominantly read only).

Our VQ scheme extends ROCC to distributed systems with client-side caching. At the client we use a Cache Validation Queue (CVQ) to record accesses to the objects stored at the client. At the server, we use a Server Validation Queue (SVQ) to record accesses to the objects stored at the server. By traversing CVQ, the client can validate local read-only transactions without communicating with the server. Meanwhile SVQ will be traversed to validate update transactions at the server.

In this thesis, we describe the VQ algorithm and using simulation, compare its performance with two other algorithms that use caching. One is the Multi-Version Concurrency Control (MVCC) algorithm, that provides snapshot isolation, and used in caching systems such as INFINISPAN (http://www.jboss.org/infinispan); the other is the optimistic concurrency control algorithm proposed by Adya et al that has been shown to perform very well (Adya et al, 1995). The simulation work demonstrates that the VQ algorithm outperforms Adya algorithm and closely matches the performance of MVCC.

1.1.3 Disconnected Operation

Disconnected operation is neither a specific technique nor a radical new idea. Rather, it is a general philosophy which holds that it is often better to do something useful for progression than nothing. With the necessary objects cached in the client computer memory and applications installed in the client computer, client logically can work under disconnected mode. In term of network connection quality, mobile clients have different characteristics compared to fixed clients; clients that run transactions from workstations with wired connection to the network. Mobile clients may have an intermittent or low bandwidth connection to the server. To enhance the system performance, clients may disconnect to the server and work offline. There are other reasons for clients to disconnect their connection network. For examples, clients may disconnect to the server for saving the battery life, for reducing network charges, or for maintaining radio silence in military operations (Jin, 1999). Thus the ability to operate in disconnected mode can be useful even when connectivity is available.

Maintaining cache consistency so as to provide transaction serializability in a disconnected environment has not been studied so far. In this thesis, we extend our scheme to work in disconnected mode.

1.2 Thesis Contributions

This thesis makes a number of contributions in the area of transactional cache consistency:

- It presents a new transactional cache consistency scheme which improves the system throughput by distributing or sharing the transactional workload between servers and clients. The scheme uses an optimistic concurrency control method which consists of two validation algorithms; the validation at client side and the validation at server side. The validation at client side is to check the client accesses against the updates of other clients sent by the server to the client. Meanwhile the validation at server side checks the client accesses against the accesses of other clients at the server. Both validation algorithms are an extension of the technique introduced by Shi and Perrizo (2002). For cache consistency protocol we design our update propagation based on *Notify locks* presented by Wilkinson and Neimat (1990).

- It evaluates and compares the performance of the proposed scheme via simulation work. Our simulation results show that the proposed scheme has better performance than the scheme presented by Adya et al. (1995) that is currently considered to offer the best performance. At the same time the performance of our scheme closely matches that of Multiversion Concurrency control (MVCC) widely used in industry but which offers snapshot isolation that is weaker than serializability.

- The scheme has been extended to work in disconnected mode. Disconnected operation includes the commits of transactions while running in disconnected mode, the update propagation to the disconnected clients, and the reconciliation process when a client reconnects again. The performance of the proposed scheme in the disconnected environment via simulation work has also been performed and shown to be acceptable.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents the necessary background information on commonly used concepts within the setting of this thesis and summarize the related work.

In chapter 3, we present our proposed scheme. We describe the system architecture, the validation algorithm of our proposed scheme, the design of the proposed scheme, and the correctness of the scheme.

The experimental framework for comparing the performance of the proposed scheme with other scheme is described in chapter 4.

A number extensions to our proposed scheme are described in chapter 5. These extensions include disconnected operation, multi-server systems, parallel transactions, MushUp server-side application and edge-computation configuration.

Finally, we conclude our thesis with summary and future work in chapter 6.

Chapter 2. Background and Related Work

The purpose of this chapter is to provide necessary background information on commonly used concepts within the setting of this thesis and to summarize briefly the related work. It provides definitions for required terms and fundamental concepts used later. We start this chapter by describing the definition of transactions and providing information about concurrency control and some techniques commonly used in concurrency control schemes. The basic issues in client-server architectures and client caching are discussed. Then we present some transactional cache consistency algorithms which are studied in our simulation work. Finally, we describe some current published papers about transactional cache consistency schemes and conclude with discussion on the performance of the schemes.

2.1 Transactions and Concurrency Controls

A *database* is a collection of objects. In this discussion a single object will be denoted as *x, y, z,* etc. Objects are assumed independent; one object does not have a relation with others. An object is either a physical resource (*e.g.*, a memory) or an abstract resource (*e.g.,* a record, a picture, a data structure). Each object has a unique identifier and some attributes (or fields). The attributes of objects are associated with their values which must be at all times related in a way that satisfies the integrity constraints of the database. However, many times when a user accesses the databases these constraints may be temporarily violated in order to transform the database to a new consistent state. Therefore, the accesses to the database are grouped together into units of consistency, called *transactions*. This means that transactions preserve consistency or transactions transform one consistent state into a new consistent state of the database.

Practically, a transaction is a collection of *readset* and *writeset*. Readset is a collection of objects to be read while writeset is a collection of objects to be written. It preserves ACID properties (Özsu and Valduriez, 1999; Gray and Reuter, 1993):

Atomicity: when a transaction contains write operations, all or none of them must be performed. In other words, if a transaction has to abort then all changes it has made in the database have to be undone.

Consistency: a transaction should be a correct state transformation, and it maps a database from one consistent state to another.

Isolation: each transaction should execute as if it is running alone in the system. Even though transactions execute concurrently, it appears to each transaction, *T*, that others are executed either before *T* or after *T*, but not both.

Durability: when a transaction is committed, the changes made by it in the database will never be lost even when the system crashes.

Although each transaction preserves consistency, some transactions need to run concurrently, in order to increase the throughput and availability of the database. However, two or more transactions executed concurrently can cause programs to behave incorrectly, thereby leading to an inconsistent database (Bernstein, Hadzilacos and Goodman, 1983). Therefore there is a need for a mechanism which monitors and controls the concurrent execution of transactions so that the consistency of database is enforced and incorrect execution of concurrent transactions is avoided. The mechanism is called *concurrency control* (Eswaran *et al.*, 1976).

Bernstein et al. (1987) define concurrency control as an activity to coordinate the actions of transactions that execute concurrently, access shared objects, and therefore potentially interfere with each other. The problems of concurrency control appear when two or more transactions are executed concurrently. One operation of a transaction may execute in between two operations of another transaction. This interleaved execution may cause an inconsistent database. An execution in which no two transactions are interleaved called *serial*. An execution is serial if, for every two transactions, all operations of one transaction execute before any of other operations of the other.

Let us consider the following example, there are two transactions, $T_1=\{R_1(x),R_1(y)\}$ and $T_2=\{W_2(x),W_2(y)\}$, and $H_1=\{R_1(x),W_2(x),W_2(y),R_1(y)\}$ is a *history* that indicates the order in which the operations of the transactions were executed. The execution of operation $R_1(x)$ precedes the execution of operation $W_2(x)$; through the object x, the execution of transaction T_1 precedes the execution of transaction T_2. The execution of operation $R_1(y)$ succeeds the execution of operation $W_2(y)$; the execution of transaction T_1 succeeds the execution of transaction T_2. Thus, the execution of transaction T_1 and T_2 is not serial.

To have a serial execution, one way would be that a system executes transactions one at a time. However, this is too inefficient. The system may make poor use of its resources.

There are allowable concurrent executions to include executions that have same effect as serial one. Such executions are called *serializable*. An execution is serializable if it produces the same results and the same effect on the database as some serial execution of the same transactions. Since the execution of the serial one is correct, and the serializable execution has the same effect as a serial execution, then the serializable execution is correct too.

Let us see the previous example. The execution of concurrent transactions in the previous example is not a serializable execution. A history $H_2=\{R_1(x),W_2(x),R_1(y),W2(y)\}$ is serializable, because it has ordered transactions to a serial execution in which transaction T_1 precedes transaction T_2.

To produce a serializable execution, the system employs a concurrency control scheme to synchronize accesses to shared objects. There are four main approaches used in concurrency control. These are listed below:

- **Locking**: If two transactions conflict, conflicting operation of one transaction must wait until the operations of another transaction are completed. This approach requires each operation must have appropriate locks before its execution.
- **Timestamp**: The execution of transactions is ordered based on their attached timestamp. Each transaction is assigned a unique timestamp. Conflicting operations of two transactions are processed in timestamp order.
- **Optimistic**: Optimistic concurrency control allows a transaction to execute unhindered to its commit point, at which time it is validated to determine whether or not to commit the transaction. It is expected that conflicting transactions hardly happen.
- **Multi Versions:** Concurrency control schemes in this category assume that operation write, $W(x)$, does not overwrite object x, instead it creates a new

copy (or version) of x. Other transactions are allowed to read the previous version of x. Thus, Reads on x are not delayed by a concurrent writer of x.

In the following paragraphs, we summarize each approach.

Concurrency Control by Locking. Concurrency control by locking requires every single object having locks associated with it; read lock and write lock (Bernstein *et al.*, 1987). To access an object, transactions should get the appropriate lock on the object. Transactions hold all locks on objects until it finishes its execution. When it finishes its execution, a transaction releases all its locks. Read lock is a shared mode lock. More than one transaction can hold read lock on object x at the same time. But write lock is an exclusive mode lock. Only one transaction can hold write lock on object x. A transaction cannot get read lock on object x if another transaction holds write lock on object x and vice versa. If a transaction cannot get one or more locks, the requesting transaction must wait until the requested locks are available.

Locking is well known mechanism and it is easy to understand. Many variations of concurrency control algorithms based on locking mechanism have been introduced in the literature. However, concurrency controls based on locking mechanism commonly have some disadvantages, one of which is that locking introduces deadlocks. Since transactions are forced to wait for other transactions to release the requested locks when they cannot get locks on the objects, this might cause deadlocks. These concurrency control algorithms should have mechanism to resolve deadlocks.

Timestamp-based Concurrency Control. In locking, the ordering of transactions in a serialization order is determined dynamically while transactions are executing based on interleaving of their requests. In timestamp-based concurrency control, the execution order of conflicting transactions is based on their timestamp. When the operations of two transactions conflict, it orders the execution of operations based on the timestamp attached to the transactions. Therefore, each transaction is assigned a unique timestamp. To achieve a unique timestamp for transactions arriving from different sites of distributed systems, all clocks at all sites must be synchronized.

Optimistic Concurrency Control. Unlike the locking or the timestamp ordering, optimistic concurrency control allows a transaction to execute unhindered to its commit point, at which time it is validated to determine whether or not to commit the transaction. The execution of a transaction consists of three phases; read, validation, and write. On its read phase, a transaction may read any object. Before a transaction begins its write phase, the system validates the transaction. If it passes the validation process, it can begin its write phase; otherwise it is aborted (Kung and Robinson, 1981).

Two validation techniques; introduced by Hardet, T., have been described Herlihy (1990); *backward* and *forward* validation. In backward validation, the system checks the validating transaction against recently committed transactions. If the validating transaction reads any object that has been invalidated by a recently committed transaction, the validating transaction is aborted. In forward validation, the system checks the validating transaction against active transactions. If it modifies any object read by a currently active transaction, it is aborted. This validation favours read-only transactions which are never aborted. Meanwhile update transactions are required to validate their writes.

Multi-version Concurrency Control: Concurrency control schemes in this category assume that operation write, $W(x)$, does not overwrite object x, instead it creates a new copy (or version) of x. Other transactions are allowed to read the previous version of x. Thus, Reads on x are not delayed by a concurrent writer of x.

However, an obvious cost of maintaining multiple versions is storage space. The system may store more than one version and each version has concurrency control information to be stored with it. To control this storage requirement, the system should periodically purge versions. Since a certain versions may be needed by active transactions, the system should synchronize with the active transactions in the process of purging versions.

However, maintaining two versions may not add much to the cost of concurrency control, because the versions may be needed anyway by the recovery algorithm. Moreover, in internet applications, since clients and persistent objects are situated in different sites, objects are copied to the clients. Thus, two version objects are already in the systems. Some concurrency control schemes may consider a persistent object x at the server and a copy of object x at clients as two versions, but others may consider as one version.

2.2 Client-Server Architecture

Client-server architecture divides distributed systems in two logical parts; clients and servers. The client-server architecture has been around for a long time and has made a significant impact on the way people do computing in distributed environment (Özsu and Valduriez, 1999). The basic idea of client-server architecture is to identify and distinguish the responsibilities and jobs that need to be done and divide these responsibilities and jobs into two sides; client side and server side.

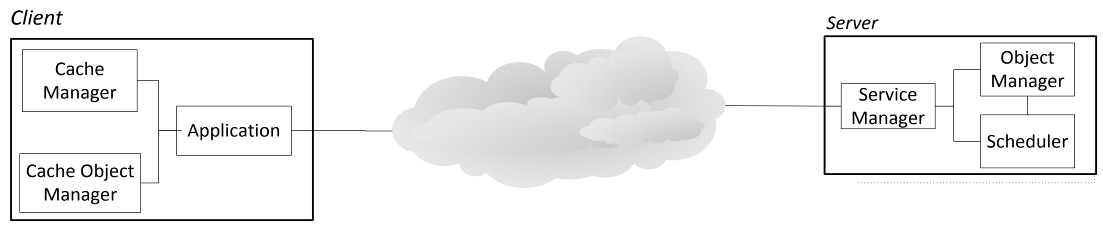


Figure 2: Client-Server Architecture

Lewandowski (1998) discusses issues of alternative designs of client-server architectures; fat servers vs. fat clients. The client-server architecture with fat servers assumes that clients have limited resources. In this type of architecture, clients send service requests to the server and the server provides the services. To response the client requests, the server may access objects and do some computation according to a certain logical business implemented at the server. The result of the computation is sent to the client, and the client presents the results to the user. Due to the proliferation of the low-cost hardware and the need to decrease the system response-time, the client-server architecture with fat clients has gained popularity. In this client-server architecture, the clients run the computation and cache some necessary objects. Therefore, the business logic of the computation is installed at clients and the server provides persistent objects and tracing objects at clients. Furthermore, Delis and Roussopoulos (1992) conclude in their research that client-server architecture with fat clients scales up a lot better for higher number of clients.

2.3 Caching

Caching is a technique that has been used in various areas of computer and database systems for quite some time. To reduce disk latency, database systems use caching technique to cache data in a buffer. It is a simple concept of storing of necessary objects in an easily accessible storage so that time and resources are saved because objects do not have to be retrieved from the original source.

Caching objects at the server side has been studied most recently in (Perez-Sorrosal et al., 2011). With the current technology, it is possible to cache all database objects in the server's computer memory. By caching objects at the server, the system resolves disk-latency and bottle-neck problems.

Caching objects at client side has also been studied by many researchers. It has some advantages (Voruganti et al. , 2004; Franklin et al., 1997). First, it exploits the resources present at the clients. Second, in the presence of locality (i.e. the affinity of the applications at certain workstations for certain subsets of database objects), caching necessary objects at client side certainly reduces the volume of objects that clients must request from servers. Third, caching objects at client side means moving objects closer to clients. Therefore, it resolves network latency and eventually reduces the system response time. Last, it resolves bottleneck problems in client-server database systems because caching will reduce the work of servers.

Caching is like replication. It introduces objects redundancy. Copies of an object are stored at multiple places. The system should ensure that the presence of multiple copies of an object does not harm any transaction. It should maintain the consistency of objects. If a transaction updates an object at the server, the updates should be available to others as soon as possible.

2.3.1 Cache Replacement Strategy

Cache size at client side is usually limited, and if the space is exhausted, cache replacement strategies decide which object should be removed. Podlipnig and Böszörmenyi (2003) describe some characteristics of web objects that can influence the replacement process. Those are *recency* (time of the last reference to the object), *frequency* (number of requests to the object), *size* (size of the object), *cost* (cost to fetch the object from the server), *modification* time (time of the last modification), and (heuristic) *expiration* time.

Podlipnig and Böszörmenyi (2003) mention some well-known strategies; LRU and LFU. LRU (least recently used) is a strategy in replacement strategy by removing the least recently referenced object. LFU (least frequency used) is a strategy in replacement strategy by removing the least frequently referenced object. LRU has been applied successfully in many different areas.

2.3.2 Degrees of Consistency

With the aim of providing of improved concurrency and better performance for some workloads by sacrificing consistency, the degree of consistency is introduced in (Gray et al., 1976). They define four degrees of consistency. Some authors may use the degree of isolation instead of the degree of consistency. In the following definition, *dirty* data refers to data values that have been updated by a transaction prior to its commitment. Then, based on the concept of dirty data, the four degree levels are defined as follows:

"Degree 3: Transaction *T* sees *degree 3 consistency* if:

i. *T* does not overwrite dirty data of other transactions.
ii. *T* does not commit any writes until it completes all its writes (i.e., until the end of transaction (EOT)).
iii. *T* does not read dirty data from other transaction.
iv. Other transactions do not dirty any data read by *T* before *T* completes."

Degree 2: Transaction *T* sees degree 2 consistency if properties i, ii, and iii hold.

Degree 1: Transaction *T* sees degree 1 consistency if properties i and ii hold.

Degree 0: Transaction *T* sees degree 0 consistency if property i holds.

For some internet applications, it is common to allow users reading data that is a little out of date; such as item prices or number of bids in an auction site, or the number of items in stock in online public store. Furthermore, most of users do not mind to read stale objects as long as their bids or their transactions are executed correctly.

We use the term of *data currency* to represent the state of objects that are accessed by transactions. The state of objects here is *up to date* or *current* and *stale* or *out of date*. An object is up to date if there is only one version of it in the system. If an object is updated by any transaction, the object is in two versions; new version and old version. We may use the word up to date for new version and stale or out of date for old version.

The use of relaxed currency, representing the state of objects that are retrieved from the server, in database systems is frequently acceptable and commonly used to enhance performance. There are a couple of published papers related to this matter that are worth discussing. These are Bernstein et al. (2006) and Guo et al. (2004). Bernstein et al. describe an extended serializability model which is called *Relaxed Currency Serializability*. This model allows transactions including update transactions to read stale objects. But their model guarantees the correctness in replicated database systems. Meanwhile Guo et al. focus on expressing the relaxed currency on SQL language. So applications have some understanding of which queries can use data that are not entirely current and which copies are "good enough".

Snapshot Isolation (SI) is a multi-version concurrency control approach that provides lock-free reads. Whenever a transaction reads an object, it does not necessarily see the latest value of the object; instead it sees the last committed version of the object. In practice, most implementations of SI use locking during updates to prevent a transaction from modifying an object if a concurrent transaction has already modified it. The first transaction to acquire the lock for an object is permitted to update the object; concurrent transactions attempting to update the same object will block waiting to acquire the lock. SI is introduced in the literature by Berenson et al (1995), and it has been implemented by many commercial systems, such as INFINISPAN (http://www.jboss.org/infinispan). It provides significant performance improvements over serializability implemented with two-phase locking (Cahill, 2009).

In conclusion that through these papers, the assumption that transactions can read data that is a little out of date, has been accepted. For some applications, the currency of data cannot be compromised. However, for many applications performance is much more important than the currency of data.

2.3.3 Edge Computing

Edge-server computing is conceptually similar to client-side caching, as the aim is to bring data closer to the client. It is widely used to improve the system performance by caching objects to edge of the network.

Leff and Rayfield in [7] explore how edge-server technology can be extended to applications requiring the use of transactional data. However, updates to shared objects cannot be made at the edge-servers. Updates take place at the server and distributed to the edge-servers in a non-transactional fashion.

2.4 Overview of Concurrency Control and Cache Consistency

2.4.1 Two-Version Concurrency Control Schemes

In this subsection we summarize briefly some researches that have been done in applying two versions of objects in concurrency control.

Maintaining two version database objects may not add much to the cost of concurrency control, because two version objects may be needed by the recovery algorithm. Bernstein et al. (1987) describe that many recovery algorithms presented in their book maintain some before image information, at least of those objects that have been updated by active transactions. The recovery algorithm needs the before image information in case any of the active transactions abort. The before image of an object is exactly the old version of an object. Thus, it is a small step for the system to make two-versions of an object explicitly available to other applications.

Two-version concurrency control schemes are presented in Bukhari (1990) and Bayer (1980). They show that two-version concurrency control increases the concurrency level and read operations never block write operations to get a write lock on the same object. Compare to one-version and multi-version, two-version concurrency control has the best performance. The one version algorithm has the best performance if the workload is 100% read-only transaction. In the various percentages of read-only transactions, the two version algorithm performs better than the others in the replicated database systems (Bukhari, 1990; Bukhari and Osborn, 1997).

Kuo et al. (2003) present a two-version concurrency control for real-time client-server database systems. They define a consistent version and working version for each object. Read operations always read from the consistent version of object and write operations always write into the working version of object. The algorithm use locking techniques to synchronize accesses to the objects. In their simulation work, they show that the use of two-version technique reduces the blocking time of the higher priority transactions and improves the response time of client-side read-only transactions. It also supports an efficient and predictable recovery mechanism.

2.4.2 The validation techniques of Optimistic Concurrency Control Schemes

The original proposal of optimistic concurrency control (OCC) is introduced by Kung and Robinson, 1981. Since then many OCCs have been introduced (Adya et al. 1995; Shi and Perrizo, 2002). Each of OCC introduces a different technique in its validation phase. The goal of the validation is to order the execution of transactions. Let transactions T_1, T_2, …, T_n be executed concurrently. Denote an instance of the shared objects by d, and let D be the set all possible d, so each transaction T_i may be considered as a function T_i: $D \rightarrow D$. If the initial of the shared objects is $d_{initial}$ and the final of the shared objects is d_{final}, then the execution of concurrent transactions is correct if any some permutation π of $(1,2, \ldots, 3)$ such that

$$d_{final} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \ldots \circ T_{\pi(2)} \circ T_{\pi(1)} \, (d_{initial})$$

where "$\circ$" is the operator for functional composition. If each transaction is consistent; or each transaction transfer the databases from one consistent to another consistent, then functional composition $T_{\pi(n)} \circ T_{\pi(n-1)} \circ \ldots \circ T_{\pi(2)} \circ T_{\pi(1)}$ will transfer $d_{initial}$ to d_{final}.

Kung and Robinson require each transaction to get a number which is called a transaction increment number during its read phase, somewhere before its validation. The validation of serial equivalence assumes the order of transactions based on the transaction increment number attached to each transaction. If the transaction increment number of transaction T_i less than the transaction increment number of transaction T_j, then the execution of T_i must precede the execution of T_j or transaction T_i must be validated before the validation of transaction T_j. Even if transaction T_j completes its read phase much earlier than T_i, before being validated, transaction T_j must wait for the completion of the read phase of T_i.

2.4.3 Disconnected Operation

Disconnected operation refers to the ability of a client to continue working on local cached objects in spite of disconnections. Kistler and Satyanarayanan (1992) show that disconnected operation is feasible, efficient and usable in the Coda file system.

Advances in computer technology have made powerful computers and large memories available at low cost. So now clients are common to run applications from high performance machines with substantial memory and processing power. To utilize the client computer resources and to solve the bottle-neck, many published papers introduce interesting solutions (Franklin, et al., 1997). Clients keep copies of the objects at their local memory. When they need the objects in the future, they can access the objects locally so that expensive communication can be avoided. Caching objects at client computer memory can enhance the overall performance of a client-server database system, especially when there is significant *locality of access* in the system workload, conflicts are rare to happen, and updates are in low percentages (Franklin, 1996).

Client caching alters the structure of client-server model. Without client caching, a client application submits its requests directly to the server. With client caching, a client application submits its requests to *cache manager*. It is served locally. The system can scale better now in the number of clients because congestion at the server can be reduced by client caching.

Advances in computer technology and wireless telecommunication have made the use of mobile computers popular in client-server database systems. This technology provides clients with the ability to access database from anywhere, and this ability is very important in future client-server database systems. The demand for ubiquitous data access is evident in the increasing prevalence of mobile computing and wireless communication.

However, mobile computers have wide variations in connectivity ranging from high-bandwidth, low latency communications through wired network to total lack of connectivity. At work, they may have access to cheap, reliable, and high-speed connectivity. In other locations, they have access to network via wireless connection which is intermittence, low bandwidth, high latency, or high expense. It is very prone to frequent disconnections.

Traditionally in client-server database systems when there is no connection to the server, clients stop working, they cannot do anything because persistent objects and the information process are stored at the server. In modern client-server database systems they hypothetically can work under disconnected mode because some needed objects and the code of information process are cached on their computer memory.

Gruber et al. (1994) discuss some issues related to disconnect operation. They discuss how to ensure that all objects needed by the client are cached prior to disconnection. They mention use of the hoarding processes (usual LRU cache policy plus user supplied 'hoarding profiles') to ensure that the right files are in client cache before getting disconnected; this mechanism is necessary for enabling users to work under disconnected mode. Then they discuss about what to do if there is a cache miss and what to do about transaction commits while running disconnected. They suggest that these problems be handled in user specific manner and the users ought to have control over how to proceed. They also discuss how to reconcile after reconnection.

2.4.4 Transactional Cache Consistency

Franklin et al. (1997) state that there are two types of caching: *intratransaction caching* and *intertransaction caching*. Intratransaction caching refers to caching within transaction boundaries. Cached objects are removed from the client cache when the transaction is committed. Intertransaction caching refers to systems that allow clients to cache objects even cross transaction boundaries. After a transaction committing, objects at the client cache are not protected by regular concurrency control. Therefore, caching requires an additional *cache consistency protocol* to regular concurrency control for maintaining the consistency of cached objects at clients.

The design of client caching mechanism for client-server database systems must respect the correctness of that environment. Client-server database systems must be able to provide the same level of transaction support in traditional database systems. Since caching is a dynamic form of replication, the criteria of correctness in replicated database systems are applicable in caching. The extension of serializability in replicated database systems is *one-copy serializability* (Bernstein et al. 1987). The execution order of transactions in replicated database systems is in one-copy serializability if it is equivalent to some *serial* orders of those transactions in non-replicated database systems.

Taxonomy of transactional cache consistency algorithms for client-server database systems can be found in Franklin et al. (1990). They categorize transactional cache consistency algorithms based on the choice of invalid access prevention. There are two categories; *Detection Based Protocols* and *Avoidance Based Protocols.* Avoidance Based Protocols ensure that all cached objects are valid. Meanwhile Detection Based Protocols allow stale objects to remain in client caches and ensure that transactions are allowed to commit only if they have not accessed stale objects. Our proposed scheme actually cannot be classified in the taxonomy presented by Fraklin et al. because our proposed scheme allows read-only transactions to commit even they read stale objects. However, the closed classification for our scheme is detection based protocol as the invalid access prevention; the validity check initiation of our scheme is deferred until commit; we propagate updates after the transaction commit.

2.4.5 Invalidation versus Propagation

Just like replication, caching introduces global redundancy by creating multiple copies of single objects. Redundant copies have to be kept consistent; coherence of copies has to be ensured in such a way that different copies give the same values. To make all copies consistent, the server employs a cache consistency protocol. There are three options for cache consistency protocol existing in the literatures: *invalidation*, *propagation*, and choosing dynamically between the two.

Cache invalidation is a process to remove stale copies from the clients as a result of the persistent objects updated at the server. The invalidated objects at clients will be inaccessible for subsequent transaction. Any subsequent transaction that wishes to access the object must obtain a new copy from the server. Moreover, the cache invalidation is based on invalidation messages the server broadcasts upon modifications of cached objects, which is prone to poor scalability due to client state managed at server side. To achieve a consistency, the commit of update transaction has to be delayed until all client caches have been invalidated.

On the other hand, propagation replaces the stale copy with the fresh one. The updates are propagated to clients which cached the updated objects right after the update transaction committed at the server. The clients can keep caching the objects and any subsequent transaction can access the objects locally. A dynamic algorithm can choose between invalidation and propagation in order to optimize performance for varying workloads (Franklin, 1996).

Wilkinson and Neimat (1990) present a cache consistency protocol using propagation. They use the term *notification* instead of propagation. The server sends notifications of updates to clients after the updates are committed. When a client receives a notification of updates from the server, it checks the list of updated objects against the readset and writeset of its active transactions. Any transaction with an updated object in its readset or writeset must abort. The server sends a notification message including a sequence number to a client. This sequence number is used for handshaking purposes only with the server. When a client submits a commit request to the server, it is required that the client has seen the most recent notification message by checking its sequence number. If the sequence number is too low, the server rejects the commit request, and asks the client to verify if the transaction should be committed and then resend the request.

2.5 Transactional Cache Consistency Schemes

In this section we describe three transactional cache consistency schemes using respectively locking, timestamp, and multi-version concurrency control.

2.5.1 Callback Locking

We describe briefly a concurrency control which is an extension of a pessimistic, locking-based protocol known as *Callback locking* (Howard et al., 1988). The algorithm is designed by Zaharioudakis, et al. (1997). It is an adaptive granularity callback locking scheme which uses Callback Read technique studied in (Franklin, 1996) to copy a page at a client side. Callback Read techniques guarantee that copies of pages at client side are always valid, so client transactions can read objects safely without communicating with the server. When a client wants to read a page which is not cached yet, it sends a request for the page to the server. Upon receiving this request, the server checks whether any other client holds write lock on the page. If there is no other client holding write lock on the page, the server sends immediately a copy of the page to the client, otherwise it delays to do so. In general the server manages write locks and tracks pages cached by each client, while read locks are recorded at the clients only.

To update a page, a client must get a write lock on the page from the server. When a write lock request arrives for a page that is not locked at the server, the server issues *callback* to all clients (except the requester) that cache a copy of the page. At the client such callback request is treated as a request exclusive lock for specified page. If the page is being read by active transaction, the client responds that the page is currently in use; this respond is used by the server for deadlock detection. Otherwise, the clients remove from the client cache and an acknowledgement is sent to the server. Whenever all callbacks have been acknowledged, the server registers the write lock on the page for the requesting client and sends a positive response to the requesting client. Any read or write request for the page from other clients will be blocked at the server until the write lock is released by the holding transaction. At the end of the transaction, the client sends the updated page to the server and releases its write lock.

Zaharioudakis, et al. (1997) also design the algorithm for object server which is the same algorithm for page server. The author argues that the object server is a better approach. It is avoiding the potential communication, memory usage, and false sharing problems of the coarse-grained project server approach. However, in the low-contention environment, the use of object server can greatly increase the number of messages required to manage cache consistency. Therefore, the author also designs an algorithm with allowing the granularity to adapt to the current level contention.

2.5.2 Adya Algorithm

Adya et al (1995) proposed a new optimistic concurrency control algorithm for use in distributed database systems. Objects are cached and manipulated at client while persistent storage and transactional support are provided by servers. There may be more than one server. The algorithm uses a loosely synchronized clock to achieve global serialization. It provides serializability and *external serializability* for committed transactions. External serializability means that the serialization order is such that if transaction T_1 is committed before transaction T_2 began, then transaction T_1 is ordered before transaction T_2. The author demonstrates that their proposed algorithm outperforms adaptive callback locking algorithm for low to moderate contention workloads, and scales better with number of clients. This section describes briefly the algorithm for providing a good background to readers. In this paper, we refer this algorithm as Adya.

Adya algorithm allows client to cache necessary objects. Objects are fetched from servers when needed. The server tracks the objects in the client cache; for each client, it maintains a table called *cached set* that records this information. The cached sets are used for maintaining cache consistency.

Whenever a transaction is about to end its execution, it submits a commit request to a server that is the owner of some objects accessed by the transaction. If the server owns all objects accessed by the transaction, it commits the transaction unilaterally. Otherwise, it acts as a coordinator in a standard two-phase commit protocol with the other servers. Note that read-only transactions also require committing their actions at servers. When server receives a commit request of a transaction, it tries to validate the commiting transaction. If validation succeeds, the server commits the transaction and sends a positive acknowledgement to the client. Otherwise it aborts the transaction and sends a negative acknowledgement to the client.

The purpose of validation process of a transaction is to prevent the commit of any transaction that would violate the consistency requirements; *serializability* and *external serializability*. The validation process uses a backward validation to preserve consistency, a validating transaction is checked against all validated transactions, earlier and later validated transactions. If the validating transaction conflicts with the validated transactions that have earlier timestamps, the system makes sure that the validating transaction accesses the correct versions of objects. If the validating transaction conflicts with the validated transactions that have later timestamps, then the validating transaction fails the validation process. Otherwise, the committing transaction succeeds the validation process. After committing an update transaction, the server sends an *invalidate message*, which is containing a list of object identifiers updated by the update transaction, to other clients that are caching any object updated by the transaction. When a client receives an invalidate message, the client drops all objects contained in the invalidate message. If the current transaction already reads any object in the list of updated objects, the client aborts the transaction immediately. When subsequent transactions wish to access any object in the list, the system could get the copy of the object from the server.

This algorithm records validation information of transactions in a *validation queue*, or VQ (note that this VQ is different than the VQ which we propose for our own algorithm later in the thesis). The validation information recorded in VQ contains the timestamp of the transaction, the transaction's readset, the transaction's writeset, and the identity of the client. To maintain VQ, it uses a *threshold* timestamp. The validation record is removed for all transactions with timestamp below the threshold. Consequently, a transaction timestamped below the threshold fails validation.

The validation process performed at the server for transaction T is described as follows:

Threshold Check

```
If $T.ts$ < Threshold then
    Send abort message to the client;
```

Checks Against Earlier Transactions:

```
For each uncommitted transaction $S$ in VQ
Such that $S.ts < T.ts$
    If ($S.Writeset \cap T.Readset \neq \{\}$) then
        Send abort message to the client;

Current Version Check
For each object x at $T.Readset$
    If (x is the invalid version) then
        Send abort message to the client;
```

Checks Against Later Transactions

```
For each transaction S in VQ
Such that $T.ts < S.ts$
    If ($T.Readset \cap S.Writeset \neq \{\}$)
    Or ($T.Writeset \cap S.Readset \neq \{\}$) Then
        Send abort message to the client;
```

Adya et al (1995) demonstrate that their algorithm outperforms an adaptive callback locking algorithm which outperforms other non-adaptive callback locking algorithms (Carey et al (1994)) and considered best so far. Therefore, it is a good reason for us to compare our algorithm with Adya algorithm.

2.5.3 Multiversion Concurrency Control (MVCC) Algorithm

In a multiversion database, each write on object *x* creates a new object or a new copy (or version) of *x*. Since writes do not overwrite the object, one or more transactions can keep read the old version of *x* while a transaction writes object *x*. This increases the level of concurrency of the system. Some systems manage one old version of objects; other systems manage more than one version of objects. A concurrency control exploiting the existence of versions of objects is called Multiversion Concurrency Control (MVCC).

Carey and Muhanna (1986) studied the performance of MVCC algorithms. In their simulation work, they show that MVCC algorithms offer significant performance improvement despite additional disk accesses involved in accessing old versions of objects.

Many variations of MVCC are published in the literature. Bernstein et al. (1987) describes some MVCC algorithms, such as MVCC two-phase locking. In two-phase locking, write lock on an object *x* prevents other transactions from obtaining read lock on object *x*. The system can avoid this by using two versions of *x*. When a transaction writes an object *x*, it creates a new version of *x* and sets a write lock on *x* that prevents other transactions writes object *x*. But other transactions can read the old version of *x*.

To apply this scheme, the system should store one or two versions of each object. Once the update transaction that writes object *x* commits, the version of *x* becomes one version. The previous version of *x* becomes inaccessible. Two version database systems are commonly used for system recovery purposes. When a transaction T_i writes object *x*, object *x* will be in two versions; T_i's before image of *x* and the new value of *x*. When T_i commits successfully, T_i's before image of *x* will be deleted. Therefore, two version database systems have been used for system recovery purposes.

Two version two-phase locking described in Bernstein et al. (1987) uses three locks; read locks, write locks, and certify locks. Read locks are compatible with read locks and write locks, but read locks are not compatible with certify locks. Write locks are compatible with read locks, but they are not compatible with write locks and certify locks. Meanwhile certify locks are inclusive locks, and they are not compatible with other locks. The scheduler of two version two-phase locking sets read and write locks as usual time, when it process reads and writes. When an update transaction is about to commit, the scheduler converts all of transaction's write locks to certify locks.

When the scheduler receives a write request on object *x* from a transaction, it attempts to set write lock on *x*. Since write locks conflict with certify locks and with each other, the scheduler set write lock on object *x* for the transaction as long as no other transaction owns a certify lock on *x* or a write lock on *x*. Otherwise, it delays the process of write on *x*.

When the scheduler receives a read request on object *x* from a transaction, it attempts to set read lock on object *x* for the transaction. Since read locks are not compatible with certify locks, it can grant read lock on object *x* for the transaction as long as no transaction owns a certify lock on *x*.

When a transaction is about to commit, the scheduler attempts to convert all transaction's write locks into certify locks. The scheduler sets certify locks on object *x* if no other transaction holds a read lock on *x*. If any, the scheduler delays the lock conversion until all read locks on *x* are released.

A transaction may deadlock while it converts all its write locks to certify locks. Therefore this algorithm uses any deadlock detection or prevention technique. In this process, the transaction may be aborted.

Cahill (2009) mentions that *Snapshot Isolation* is a multi-version concurrency control approach that provides lock-free reads. Unlike most other MVCC algorithms, update transactions can also avoid locking for their reads. When a transaction T_i starts executing at Snapshot Isolation and reads object *x*, it does not necessarily read the latest value written to *x*; instead it sees the latest committed version of *x*. To update object *x*, transaction T_i should acquire write lock (an exclusive lock). If T_i fails to get write lock on *x*, it waits until the lock is available. This may cause T_i involve in deadlock. The system should employ a mechanism to solve deadlock problems. In practice, one may abort the update transaction which fails to get any lock.

Bober and Carey (1991) use MVCC in different degrees of consistency, such as degree 1 or degree 2 consistency, for long running queries. The reason to use different degree of consistency for long running queries is to increase the performance. Long running queries introduce a high level of data contention. MVCC definitely reduces data contention. For similar reason, many commercial database systems use these kinds of MVCCs which can only ensure a weaker form of consistency than serializability; such as INFINISPAN (http://www.jboss.org/infinispan). In this MVCC, read operations can always get accesses. There is no read lock for a read operation. It always reads a committed object. Meanwhile, a write lock is required for write operation. A transaction should get write locks from the server before it executes its write operations.

2.6 Conclusion

Adya algorithm presented in Adya et al. (1995) validates a transaction against validated transactions. The validating transaction is aborted if it reads any object written by younger validated transactions or it writes any object read by older validated transactions. Otherwise the validating transaction is passed the validation process. Thus, the order of transactions is based on the timestamp attached to each transaction.

The Adya algorithm and the original optimistic concurrency control algorithm use external numbers to order the execution of transactions. Read-commit Order Concurrency Control (ROCC) introduced by Shi and Perrizo (2002) and discussed in the next chapter is more flexible. It does not need an external number, such as timestamp or transaction increment number, to order the execution of transactions. It can order the execution of transactions based on the transaction's accesses because it records the accesses of each transaction. Based on the record, the validation process of ROCC tries to insert a transaction into some serial order. If it cannot insert a transaction to any place into some serial order, this means the execution of the transaction interleaves with others; therefore the transaction is aborted.

Thus, the validation process of ROCC gives transactions a better chance to success because it has many options to order the transactions. Meanwhile, validation in each of the other techniques orders transactions based on one serial order. For examples if the timestamp (or the transaction increment number) of T_i is less than the timestamp (or the transaction increment number) of T_j, then T_i must precede T_j in the execution order produced by the validation process of Adya and Kung's algorithm. Meanwhile, ROCC could produce any order ($T_i \rightarrow T_j$ or $T_j \rightarrow T_i$). Let $T_1=\{R_1(x), W_1(x)\}$, $T_2=\{R_2(x),W_2(y)\}$ and $H_1=\{R_1(x),R_2(x),W_1(x),W_2(y)\}$. ROCC's validation would produce $T_2 \rightarrow T_1$. But the others fail to produce $T_1 \rightarrow T_2$ because T_2 conflicts with T_1 and is aborted.

Adya's and Kung's validation technique is not suited for systems which have long running transactions and short running transactions. Adya's scheme makes long running transactions suffer. The Adya's long running transactions will tend to abort because they will check against many short running transactions. On the other hand, as ROCC's validation produces some serial order, ROCC's long running transactions still have a better chance to succeed. Meanwhile, in Kung's validation short running transactions are forced to wait the long one. Consider the case of two transactions, T_i and T_j starting roughly at the same time, assigned transaction number n and $n+1$, respectively. T_i is a long running transaction and T_j is a short running transaction. Before being validated, T_j must wait for the completion of the read phase of T_i.

In the following chapter, we present a transactional cache consistency scheme. The scheme is based on the optimistic approach. It is an extension of ROCC which is considered more flexible in ordering transactions and suited for client-server systems with caching at client sides.

Chapter 3. The Proposed Scheme

In this chapter, we present the proposed scheme. We call our proposed scheme as VQ which stands for Validation Queue, because it uses a validation queue to synchronize the accesses to objects and to validate transactions[1]. The primary design goals of the proposed scheme are: (1) to increase the system performance by caching necessary objects at client side and (2) to reduce the amount of communication with the server.

In the design of the proposed scheme, we use some common assumptions as follows:

- We assume a single server system. Thus, multi-server issues, such as the use of two-phase commit protocol, are ignored. Chapter 5 describes the extension of our scheme to multi-server systems.
- We assume a client issues transactions one at a time. Our scheme can be extended to parallel transactions; one client can issues more than one transaction at a time. This extension is described in chapter 5.
- We assume no blind write. If a transaction wants to update object *x*, it has to read object *x*.
- We assume a transaction works on its own memory. When it requests accesses to objects, the objects are copied to its own memory. It can modify the objects in its own memory. When it is about to end its execution, its commit request including its updates is submitted. In other words, we use a deferred write technique for write operations.
- To focus the problems that are addressed in this thesis, we assume there are no network partitions. A message is always delivered to its destination. Furthermore, we assume that messages are received and processed at the client in the same order as they are sent from the server, with the network preserving the number and the order of messages.

[1] A preliminary version of the scheme appeared in 2012 IEEE 14th International Conference on High Performance Computing and Communications (Bukhari, F. and Shrivastava, S., 2012).

Since our scheme is an extension of ROCC algorithm, we start this chapter with the description of ROCC algorithm. Understanding of ROCC algorithm is helpful to understand our scheme. Afterward, we describe the system architecture and the validation algorithm. The design of the proposed scheme is described more detail in section 3.4. Finally, this chapter is closed with the correctness of the proposed scheme.

3.1 Read-Commit Order Concurrency Control (ROCC)

3.1.1 Read-Commit Queue

Shi and Perrizo describe a new concurrency control method for a centralized database system (Shi and Perrizo, 2002). The concurrency control method is called Read-commit Order Concurrency Control (ROCC). ROCC is a deadlock-free concurrency control method based on optimistic mechanisms. It employs a centralized queue called Read-Commit queue (RC queue) to record the access order of transactions. Along with the RC queue, an "intervening" validation algorithm is developed for execution validation. In addition to traditional operation conflict, they introduce a new concept; element conflict.

A client application executing a transaction sends one or more read request messages to the database system to fetch copies of the data objects; at commit time, the client sends a commit request message with new values of any fetched objects that have been updated. The database server performs validation to determine whether the transaction should commit or abort. If a transaction succeeds its validation process, it is committed. Otherwise, it is aborted.

Generally, a transaction may submit more than one request to the server. Shi and Perrizo define a *static transaction* as a transaction that submits only one request to the server.

Whenever the system receives a read request message, it generates a corresponding element and inserts it into the RC queue. An element contains the transaction identifier (TID), the element type, one or more object identifier fields (such a field contains the list of object identifiers to be accessed and other information) and links for queue management. The element and its fields will be depicted as follows:

TID	Element Type	Object Identifiers	Links

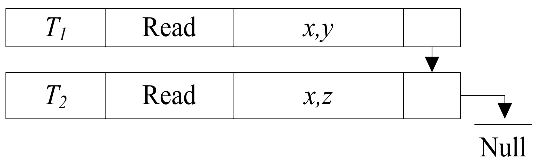
An RC queue may have four types of elements: Read element, Commit element, Validated element, and Restart element. A Read element is created and inserted into the RC queue whenever the system receives a read request message. Since a transaction may submit several read request messages, there could be several Read elements related to the transaction. All the objects that a transaction requests to write are contained in the writeset object identifier field of the Commit element. A Commit element also has a readset object identifier field (this is normally empty and used only in the VQ algorithm discussed later). The system executes data object accesses in the same order as they appear in the RC queue. The system traverses the RC queue to validate a transaction. If the transaction passes the validation process, then all of its elements are merged into a Validated element. Otherwise, a Restart element will be generated.

3.1.2 Examples of Cases

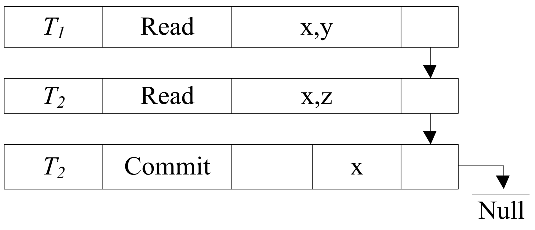
In this subsection, we present some simple cases. The purpose of this subsection is to motivate the studies of ROCC algorithm. We give some illustrations for the 'intervening' validation algorithm. The cases are independent, except the first and second cases.

Case 1:

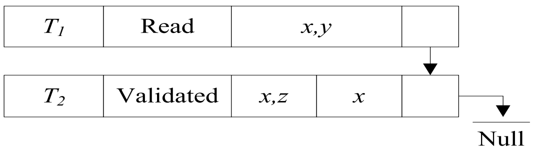
The following figure represents a structure of RC queue. Its first element (top of the RC queue) represents a Read element. It contains transaction identifier: T_1; element type: Read; the list of object identifiers (x, y). The second element is a Read element of transaction T_2.



Then transaction T_2 submits its commit request message containing a write operation on object x. The system creates a Commit element of transaction T_2 and inserts it into the RC queue. The RC queue will then be as follows:



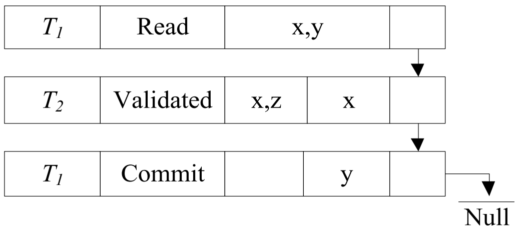
Before executing the commit request message of transaction T_2, the system validates transaction T_2. The validation process checks whether the execution of transaction T_2 interleaves with the execution of other transactions. If the execution of transaction T_2 does not interleave with the execution of other transactions, then transaction T_2 will pass the validation process, otherwise, it will fail the validation process. To validate transaction T_2, we need to examine transaction T_2's intervening elements from other transactions in the RC queue. In the above RC queue, we can see that there are no elements belonging to other transactions in between the Read element and Commit element of transaction T_2. Transaction T_2 passes the validation process and its elements are combined to form a validated element and the RC queue will now look as follows:



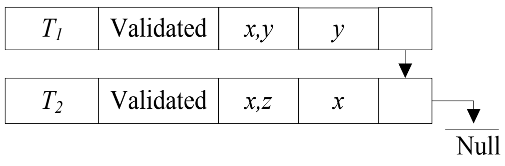
Now, transaction T_2 is represented by an element; which is the Validated element. It contains transaction identifier (T_2), element type (Validated), object identifiers to be read (x,z), and an object identifier to be written (x). The existence of the validated element of transaction T_2 in the RC is required queue for validation processes of other transactions.

Case 2:

Afterwards, transaction T_1 submits its commit request message which contains write operation on object y. The RC queue will be as follows:

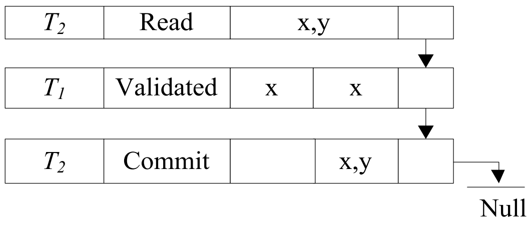


To validate transaction T_1, the system checks elements of transaction T_1 against 'in between' elements from other transactions. Transaction T_1 reads object x before transaction T_2 updating it. The read element of transaction T_1 conflicts with the validated element of transaction T_2. Now we examine the commit element of transaction T_1 and the validated element of transaction T_2 and find that they are not in conflict each other. We can therefore order the commit element of transaction T_1 before the validated element of transaction T_2. Therefore, transaction T_1 passes its validation process, and the RC queue will look as follows:



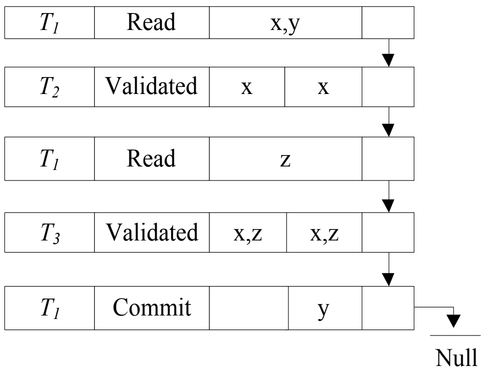
Case 3:

Let us consider another case, for example, after transaction T_2 submitting its commit element, the RC queue of a system is shown as follows:

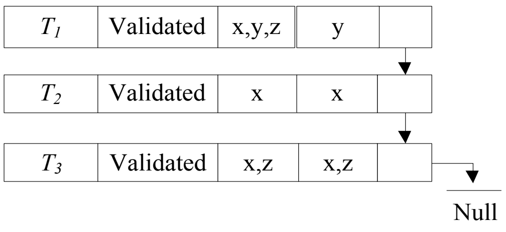


Transaction T_2 fails the validation process. Its read element conflicts with the validated element of transaction T_1. T_2's read element reads object x and T_1's validated element writes object x. They are in conflict. The read element of transaction T_2 cannot pass the validated element of transaction T_1. Now, we examine the Commit element of transaction T_2. The commit element of transaction T_2 contains a write operation on object x. The commit element of transaction T_2 conflicts with the validated element of transaction T_1 as well. It cannot pass the validated element of transaction T_1 as well. In this case, the execution of transaction T_2 interleaves with the execution of transaction T_1. Therefore transaction T_2 fails the validation process. Consequently, it is aborted.

Case 4: Let us consider the following RC queue:



The validation process of transaction T_1 is successful. Transaction T_1 reads objects x and y which is represented by the first read element of transaction T_1. Its first read element conflicts with the validated element of transaction T_2. Now, we examine the commit element of transaction T_1. It does not conflict with the validated element of transaction T_3. Therefore, it can pass the validated element of transaction T_3. When we move up, then we find another element of transaction T_1. We combine the commit element of transaction T_1 with its read element. The combined element of transaction T_1 does not conflict with the validated element of transaction T_2. Now we move up, we find the first read element of transaction T_1. Therefore, we can combine all elements of transaction T_1 to be one element which is validated element. As a result, transaction T_1 succeeds the validation process, and the RC queue will look as follows:



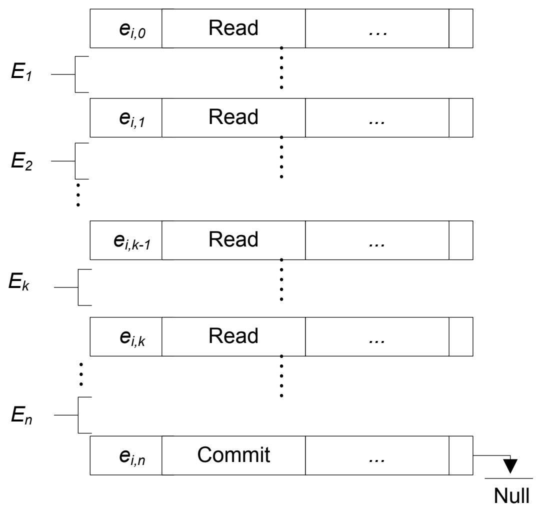
3.1.3 The Validation Algorithm

This section describes the ROCC validation algorithm in a rigorous manner. Two elements, element $e_{i,p}$ from transaction T_i and element $e_{j,q}$ from transaction T_j ($i \neq j$) are in conflict if at least one of the following condition is true,

- $ws(e_{i,p}) \cap ws(e_{j,q}) \neq \{\}$,
- $ws(e_{i,p}) \cap rs(e_{j,q}) \neq \{\}$,
- $rs(e_{i,p}) \cap ws(e_{j,q}) \neq \{\}$,

where the notation of *ws*(*e*) means the writeset of element *e* and the notation of *rs*(*e*) means the readset of element *e*. The j-th element of transaction T_i is represented by $e_{i,j}$. The notation E_i represents a sequence of elements. If two elements from the same transaction $e_{i,p}$ and $e_{i,q}$, are merged ($e_{i,p} \cup e_{i,q}$) to make a single compound element $e_{i,r}$, then $ws(e_{i,r}) = ws(e_{i,p}) \cup ws(e_{i,q})$ and $rs(e_{i,r}) = rs(e_{i,p}) \cup rs(e_{i,q})$. The compound element $e_{i,r}$ represents the existence of the element $e_{i,p}$ and $e_{i,q}$ in transaction T_i.

The validation process of transaction T_i is started after the transaction submits the commit request. Suppose transaction T_i has $n+1$ elements in RC queue, $e_{i,0}$, $e_{i,1}$, .., $e_{i,n}$; where $e_{i,j}$ is any Read element of transaction T_i ($0<=j<=n-1$); $e_{i,n}$ is the Commit element of transaction T_i. The structure of RC queue, from the first Read element of T_i to the Commit element of T_i (or the rear of RC queue) can be considered as follows:



E_1 is a (possibly empty) collection of elements from other transactions in between element $e_{i,0}$ and $e_{i,1}$; E_2 in between element $e_{i,1}$ and $e_{i,2}$; and E_j in between elements $e_{i,j-1}$ and $e_{i,j}$ of transaction T_i for $1<=j<=n$. Let e^* be an element in E_k that splits E_k into two parts *A* and *B*, therefore $E_k=A;e^*;B$ (where, *A* and/or *B* can be empty sequence). Transaction T_i passes its validation process, if one of the two following condition is satisfied;

1. An element or the compound element $e_{i,0} \cup e_{i,1} \cup \ldots \cup e_{i,j}$ **does not conflict** with any element in the sequence E_{j+1}, for all $j=0,1,\ldots, n-1$.

2. The compound element $e_{i,0} \cup e_{i,1} \cup \ldots \cup e_{i,j}$ **does not conflict** with any element in E_{j+1} for all $j=0,1,\ldots,k-1$ and any element in *A* but the compound element **conflicts** with element e^*, for $k=1,2, \ldots, n$. Then, the element $e_{i,n}$ or the

compound element $e_{i,n} \cup e_{i,n-1} \cup \ldots \cup e_{i,j}$ **does not conflict** with any element in sequence E_j for all $j=n,n-1,\ldots,k+1$, and the compound element $e_{i,n} \cup e_{i,n-1} \cup \ldots \cup e_{i,k-1} \cup e_{i,k}$ **does not conflict** with element e^* and any element in B.

In other words, condition 1 is true, only if no elements from other transactions in between the first read element and the commit element of transaction T_i conflict with the elements of T_i. Condition 2 is true, only if the first read element of T_i or its compound element (from its first read element forwards) conflicts with element e^* of other transaction in between the execution of transaction T_i, but the commit element of transaction T_i or its compound element (from its commit element backwards) does not conflict with any element from other transactions including element e^*. If transaction T_i fails the validation process, its elements are removed from the RC queue.

Let us see again the previous cases. The first case, transaction T_2 satisfies condition 1 so it passes the validation process. The second case, transaction T_1 satisfies condition 2, so it also passes the validation process. However, in the third case, transaction T_2 does not satisfy condition 1 or condition 2. Therefore, it fails the validation process. In the fourth case, transaction T_1 passes the validation process, because it satisfies condition 2.

The rest of this section describes briefly the pseudo code of the "intervening" validation algorithm of ROCC which is presented at Figure 3. The pseudo code shows how to traverse RC queue when the algorithm is validating a transaction. Initially, the algorithm sets the first read element of the validating transaction to FIRST and the commit element of the validating transaction to SECOND. From the position of FIRST, the algorithm traverses RC queue to the rear of the queue and checks if FIRST conflicts with its in-between elements of other transactions. If FIRST does not conflict with its in-between elements from other transactions, then merge FIRST and the read element of the validating transaction, let this merged element be FIRST and place FIRST to the position of the last read element. The algorithm continues checking FIRST against its between elements. If the algorithm reaches SECOND which is the commit element of the validating transaction, then the algorithm returns **true.** This condition is the same as condition 1 described above.

If a conflict is found, then move FIRST to the front of the conflicting element. Now traverse RC queue from SECOND toward FIRST and check if SECOND conflicts with its in-between elements of other transactions in the same way as described above. If SECOND conflicts with its in-between elements, then the algorithm returns **false** after it removing all elements of the validating transaction. Otherwise; SECOND does not conflict with its in-between elements and reaches FIRST, then the algorithm returns **true**. This is the same as condition 2 described above.

```
FIRST = get the first read element of the transaction;
PREV = NULL;
SECOND = get the commit element of the transaction;
NEXT = get element after FIRST element;
for (;;)
{
    If (NEXT is another read element of the transaction)
    {
        Remove FIRST element from the RC queue;
        FIRST = merging FIRST and NEXT then store at the position of NEXT element
                in the Queue;
        NEXT = get next element;
    }
    else if (NEXT is equal to  SECOND)
    {
        Remove FIRST element from the RC queue;
        Merging FIRST and SECOND store into SECOND as local validated element;
        Return success;
    }
    else if (FIRST conflict with NEXT)
    {
        /* move FIRST to the position before NEXT */
        Remove FIRST from the RC queue;
        Insert FIRST before NEXT in the RC queue;
        PREV = get previous element of SECOND;
        for (;;)
        {
            If (PREV is equal to FIRST)
            {
                Merging SECOND and FIRST store at the position of FIRST as local validated;
                Return success;
            }
            else if (PREV is another element of the transaction)
            {
                Remove SECOND from the RC queue;
                SECOND = merging SECOND and PREV store at the position of PREV ;
                PREV = get previous element of SECOND;
            }
            else if (SECOND conflict with PREV)
            {
                Remove FIRST, SECOND, and all the remainder elements of the transaction;
                Return failure;
            }
            else
                PREV = get previous element of PREV;
        }
    }
    else
        /* FIRST is not conflict with NEXT*/
        NEXT = get next element of NEXT;
}
```

Figure 3: The Validation Algorithm

3.2 System Architecture

We consider a system that shares a common object database over a large geographic area. We divide the infrastructure into *a server* and a set of *clients* connected by a network (see Figure 4). The server is specialized to hold persistent objects and to provide them to clients on request. The clients run applications that request accesses to the objects. The communication between a client and the server occurs only through explicit message across the network. To resolve the network latency problem, the client caches necessary objects.

3.2.1 Client Side Components

Client side consist of *application*, *Cache Manager*, and *Cache Object Manager*. They are independent modules and communicate with each other through explicit messages. In this research, Cache Manager together with Cache Object Manager is referred as a *cache* (or local cache). As an alternative architecture, a cache can be placed in a separate computer to the application.

Each client may have different applications. We do not restrict the application to a specific one. However, when an application wants to access database objects, it creates a transaction and accesses objects through the transaction. To access objects, a transaction submits requests to cache manager.

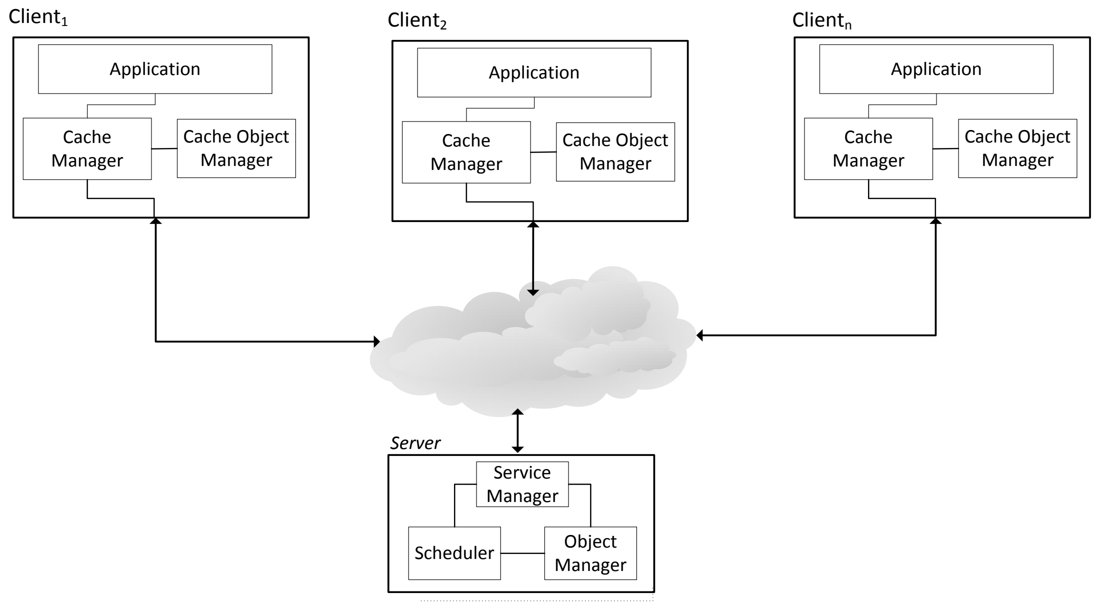


Figure 4: Client-Server Architecture

Cache Manager is a module which has a dual function in the system. At one hand, it represents the server to the client application. The client application submits its requests to the local cache manager. The client application does not know the existence of the server, and it does not need to know the server because the local cache manager acts like the server for the client application. The cache manager submits a request to the server for updating database objects on behalf client applications. It fetches or drops objects to the server on behalf the client.

To provide correct execution of local transactions, the local cache manager is required to take actions on committing local transactions. The actions taken by a local cache manager are governed by the outcome of *Cache Side Validation Algorithm*. If the outcome is negative, it aborts the transaction and tells the client about its decision and the client creates a new transaction. Otherwise, it commits the transaction; a read-only transaction can leave safely; an update transaction requires second validation at the server; a commit request message is created (containing the readset and the writeset with the new values) and forwarded to the server for global, server side validation.

3.2.2 Server Side Components

The server consists of *Service Manager*, *Scheduler*, and *Object Manager*. Service Manager coordinates incoming and outgoing messages at the server. Any access request to the database is submitted to Service Manager. Then it is directed to Scheduler. Scheduler has a responsibility to synchronize the access to the database. To execute the accesses, Scheduler submits them to Object Manager.

Besides coordinating incoming and outgoing messages, Service Manager is also responsible to produce a unique version number for a cache. A new cache version number is required if a client cache is initiated or updated by the server. In order to get services from the server, each transaction should be provided with a valid version number of its originated cache. Otherwise, the transaction would be rejected and aborted.

Scheduler is responsible to validate update transactions and to track client cached objects. Meanwhile, Object Manager executes Read, Fetch or Commit elements submitted by Scheduler.

Accesses to database are managed by the scheduler. To manage accesses to objects at the server, the scheduler may execute or reject an access request of transactions to objects. The actions taken by the scheduler are governed by an algorithm explained in the next section. If the scheduler takes the action to execute the request, it passes the request to object manager for execution and notifies the client about its decision. When object manager finishes executing the request, it informs the scheduler and eventually the scheduler propagates the updates to other clients. If the scheduler takes the action to reject the request, in which case it tells the client that its request have been rejected.

3.3 The Validation Algorithm

We now present the validation algorithm of the proposed scheme. The validation occurs in two sides; cache side and server side. Cache side validation is to validate local transactions. Meanwhile server side validation is to validate a transaction that updates any persistent object at the server. The validation algorithms at both sides are an extension of the validation algorithm described in section 3.1.

The main objective of validation algorithm is to provide serializability order to the committed transactions by not allowing interleaved transactions to commit. Therefore, sometime before a transaction finishes its execution, the system checks whether its execution interleaves with others'. To check whether the execution of a transaction interleaves with the execution of other transactions, we use a *validation queue*; we call it validation queue, because we use it for validation purpose only[2]. This validation queue is used to record the execution order of transactions. Since we use elements as the execution unit of a transaction, then this validation queue contains elements.

There are two kinds of validation queues in the system. The first validation queue is named as Cache Validation Queue (CVQ) because it is located at cache sides. CVQ is maintained by the local cache manager. It is used to record accesses to cached objects at client side. The second validation queue is Server Validation Queue (SVQ) located at the server and maintained by the scheduler. SVQ is used to record accesses to database objects at the server. Both cache and server sides make use of the same ROCC

[2] In section 3.1 the validation queue was termed RC queue.

of algorithm for validation (discussed in section 3.1), making use of some additional elements as we discuss below. Each client cache has a unique sequence number assigned by the server; this sequence number is included in all the messages sent by the clients to the server. The server generates a new, higher sequence number when it has to update a cache; the new sequence number is included in the update message from the server (this message is called the Update Propagation message).

3.3.1 Cache Side Validation Algorithm

The cache side validation algorithm described in this subsection is invoked by the local cache manager when it validates a local transaction. Its objectives are to prevent the commit of incorrect execution of transactions. It checks the correctness of a transaction execution by examining the execution order of the transaction. If it finds the execution of a validating transaction interleaves with others, it returns failure; otherwise it returns success.

The cache side algorithm uses CVQ as a tool to record the execution order of elements, in the same manner as the RC queue. In addition to Read, Commit, and Validated elements, CVQ contains Local Validated and Update Propagation elements. An Update Propagation element represents the execution of a remote update transaction. It contains the readset and writeset of the update transaction. It is inserted when the local manager receives an Update Propagation message from the server (as discussed in the next sub-section); Read or Commit elements are inserted into CVQ as a result the local manager receives read or commit request respectively from local transactions.

Whenever any transaction is about to end its execution, it submits its commit request to the local cache manager. Upon receiving a commit request, the local cache manager creates a commit element and posts it into CVQ. Then it validates the transaction. If the transaction succeeds the validation process, then, if it is a read-only transaction, all its elements are merged to be a Validated element. Otherwise, the locally validated transaction is an update transaction and the process is as follows: (i) all its elements are merged to be a Local Validated element; (ii) the local cache manager submits its commit request message to the server. A local validated element is turned in to a validated element if the response from the server is positive, else (the response is abort) the local element is discarded.

To validate a transaction, the local cache manager invokes the validation algorithm described in section 3.1. In ROCC algorithm, a transaction; read-only and update transaction, succeeds in the validation process if its elements in RC queue satisfies condition 1 or 2. However, the validation algorithm of our scheme in cache side is as follows:

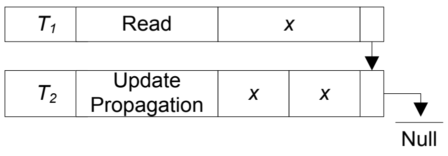
- A read-only transaction succeeds the validation process if it satisfies condition 1 or condition 2. Otherwise it fails.
- An update transaction succeeds the validation process if it satisfies condition 1 only. Otherwise it fails.

3.3.2 Examples of the Execution of Transactions at Cache Side

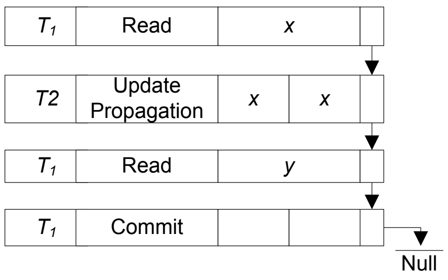
In this subsection, we present some simple examples. The purpose of this section is to motivate the studies of the cache side validation algorithm of the proposed scheme. We give some illustrations for the validation algorithm. The examples are independent.

Example 1:

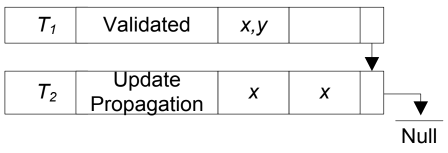
In the first example, we consider two transactions; $T_1=\{R_1(x),R_1(y)\}$ as a local read-only transaction and $T_2=\{R_2(x),W_2(x)\}$ as a remote update transaction (from other client). Initially, T_1 submits its first read request. Then the local cache manager inserts its read element into CVQ and executes the element. Eventually the local cache manager manages to send the value of object x to the client transaction. At the same time, transaction T_2 from another client commits at the server. The server sends an update propagation message to client cache that caches the object x. Upon receiving the update propagation message, the local cache manager creates the corresponding element; an Update Propagation element, and inserts it into CVQ, then it forwards the message to Cache Object Manager to refresh the cached objects. The following figure represents a structure of CVQ. Its first element (top of the queue) represents a Read element of T_1. It contains transaction identifier: T_1; element type: Read; the list of object identifiers (x). The second element is an update propagation element of T_2; it contains a read operation on x and a write operation on x. After executing the update propagation element of T_2 at cache side, the object x has two versions; the old version at T_1's working memory and the new version at the cache's memory. For the correctness of our scheme, the execution of update propagation element does not automatically reflect to T_1's working memory.



Therefore, transaction T_1 can keep running and sending its requests; a read request and a commit request. Eventually, CVQ looks as follows,



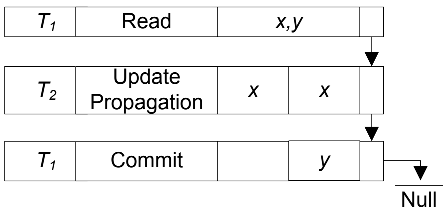
To commit transaction T_1, the local cache manager executes the cache side validation algorithm. Since T_1 is a read-only transaction and its elements satisfy condition 2 of the validation algorithm described in section 3.1, it succeeds the validation process at cache-side. Its elements are merged to be a validated element. After the validation process of transaction T_1, CVQ is shown as follows,



and the validation algorithm returns success. The execution order of these two transactions is considered as $T_1 \rightarrow T_2$ even T_1 commits after the commit of T_2. Eventually, these two elements are removed from CVQ because their existence in CVQ is not necessary anymore.

Example 2:

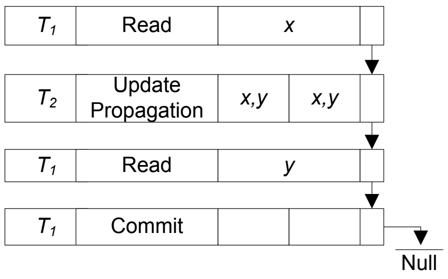
In this example, we want to show the case that a local update transaction reads stale objects. Let us consider a local update transaction $T_1=\{R_1(x),R_1(y),W_1(y)\}$ and a remote update transaction $T_2=\{R_2(x),W_2(x)\}$. Initially, the local transaction T_1 submits its read request which contains read operations on x and y. Then the local cache manager receives an update propagation message of T_2 from the server. Afterward, T_1 submits its commit to the local cache manager. Now, CVQ looks as follows,



Transaction T_1 is an update transaction and its elements do not satisfy condition 1 of the validation algorithm described in section 3.1. Therefore, transaction T_1 fails the validation process and it is aborted.

Example 3:

To show the execution of a transaction interleaving the execution of another transaction, let a local read-only transaction $T_1=\{R_1(x),R_1(y)\}$ and a remote update transaction $T_2=\{R_2(x),R_2(y),W_2(x),W_2(y)\}$. Briefly, CVQ shows as follows,



The execution of T_1 is interleaving with the execution of T_2 at local cache. Transaction T_1 sees object x before updated by T_2, but through object y T_1 sees it after being updated by T_2. If we apply the validation algorithm to transaction T_1, it returns **failure**, because transaction T_1 does not satisfy condition 1 and 2. Therefore, transaction T_1 is aborted.

3.3.3 Server Side Validation Algorithm

There are three main tasks for the Server Validation Algorithm: to validate an update transaction at the server, to propagate the updates to the caches, and to maintain *Cache elements*. A Cache element contains the information about the objects stored at a cache. This algorithm uses SVQ in the same manner as the RC queue. SVQ may contain Cache, Commit, or Validated elements.

Fetch requests from a cache are treated at the server side as requests from a *cache transaction*; which is a transaction associated with a cache; it is a long running transaction at the server; its life span is equal to the life of an associated client cache. When the server receives a fetch request from a cache-side, it creates a commit element of associated cache transaction and posts it into SVQ. This commit element contains, in its readset field, the names of the objects the cache-side is requesting; the writeset field is empty. Fetch operations are transactional operations. Therefore, after inserting the commit element of the associated cache transaction into SVQ, the server validates the cache transaction. If the cache transaction passes the validation process, then its fetch operations are submitted to the object manager. Otherwise, the commit element is removed from SVQ; to make sure that the fetch operation gets the committed values, it will be delayed and retried later. If the fetch request carries a *drop request*; note that a drop request contains a list of cached objects to be removed from the client cache, then the server extracts the list of dropped objects from the request and modifies the list of cached objects on the correspond cache element; note that drop operations are not transactional operations. Eventually, the requesting cache manager will get a positive response together with the requested object values from the server.

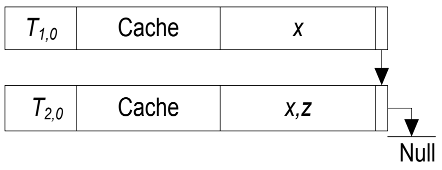
Whenever the server receives a commit request of a transaction, it behaves as follows: if the sequence number carried by the commit request message is not equal to the sequence number recorded on its cache transaction at the server, the commit request is sent back to its originated cache manager for verification; else the server creates two elements. These are a Read element and a Commit element. The Read element contains the list of object identifiers that have been read by the update transaction. The Read element will not be executed; it is needed for the validation purposes only. Meanwhile, the Commit element contains the list of object identifiers that the transaction wants to update. The Read element is inserted into SVQ at the position right after the position of the Cache element of the associated cache transaction in SVQ.

The validation process at the server-side adopts the validation algorithm described in section 3.1. It only requires any transaction to satisfy condition 1. Otherwise the transaction is considered failure in the validation process. The pseudo code of the server-side validation algorithm is provided in the following section.

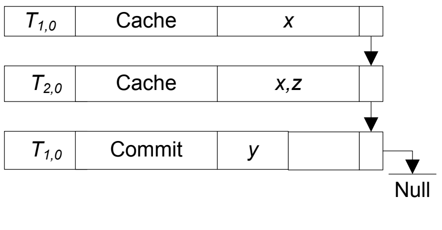
If the validation is successful, the server sends a commit acknowledgement message to the originating cache manager, executes the updates of the transaction, and refreshes other caches (holding stale versions) by sending Update Propagation messages, with new sequence numbers. If the validation is failure, the server removes the commit element from SVQ and sends an abort message to the originated cache manager.

3.3.4 Examples of the Execution of Transactions at Server Side

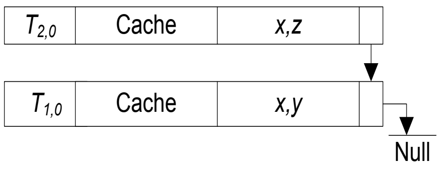
This section illustrates the Server Validation Algorithm with some simple cases. Let us consider two transactions from two clients in the system. The system has three objects *x, y,* and *z*. Each client has its own cache, with cache version numbers 1 and 2 respectively. Cache 1 currently stores object *x* and cache 2 stores object *x* and *z*. SVQ contains two Cache elements as follows:



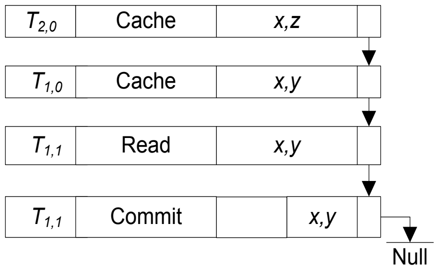
The first Cache element represents cache transaction $T_{1,0}$ from client with Cache 1. It caches object *x*. The second Cache element represents cache transaction $T_{2,0}$ from client with Cache 2. Consider client at cache 1 issues transaction $T_{1,1}$ = {R(x,y),W(x,y)}. Transaction $T_{1,1}$ submits its read request to its local cache manager. Its read request contains object *x* and *y*. Since cache 1 only stores object *x*, it needs to fetch object *y* to the server before it responses the read request of transaction $T_{1,1}$. It sends a fetch request to the server for object *y* on behalf of cache transaction $T_{1,0}$. When the server receives the fetch request, it inserts a commit element of cache transaction $T_{1,0}$, and SVQ is as follows:



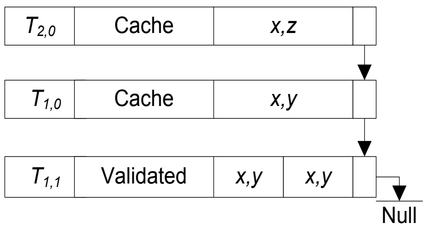
By considering the $T_{1,0}$'s cache element as the first read element of cache transaction $T_{1,0}$, cache transaction $T_{1,0}$ succeeds the validation process. Eventually, the server sends the value object y to the client *1* as a fetch acknowledgement message. Now, SVQ looks as follows:



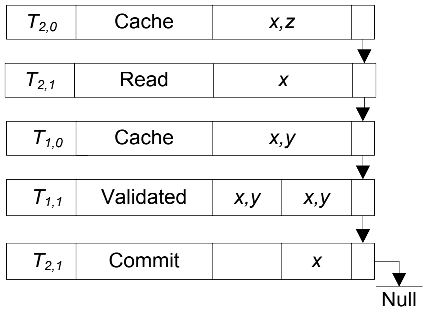
Shortly, the local cache manager of client *1* submits the commit request of transaction $T_{1,1}$ to the server. The server creates two elements of transaction $T_{1,1}$: Read and Commit element. Read element is inserted after the Cache element of cache transaction $T_{1,0}$. SVQ is as follows:



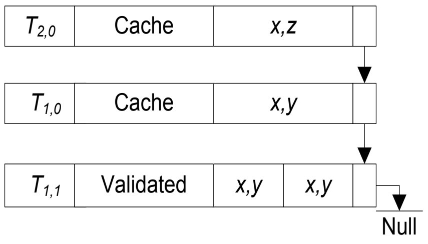
The elements of transaction $T_{1,1}$ satisfy the condition 1 of the validation algorithm described in section 3.1. Therefore, a commit acknowledgement message is sent to the cache *1* and SVQ is modified as follows:



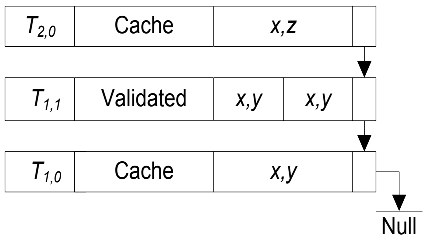
Now consider the commit request of transaction $T_{2,1} = \{R(x), W(x)\}$ from cache *2* arrives at the server. The server creates two elements; Read and Commit elements, and inserts them into SVQ as follows:



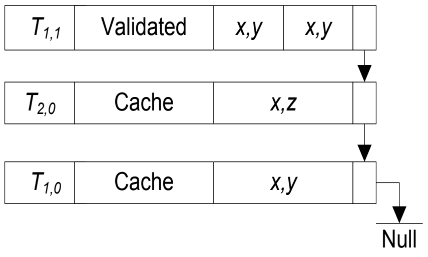
The read element of transaction $T_{2,1}$ conflicts with the validated element of transaction $T_{1,1}$; one reads object x, another writes object x. Thus, the elements of transaction $T_{2,1}$ do not satisfy the condition 1 of the validation algorithm. Therefore, transaction $T_{2,1}$ does not pass the validation process. Consequently, it is aborted and removed from SVQ. SVQ shows as follows,



As object x and y have been updated by transaction $T_{1,1}$, each cache side should be refreshed by creating and sending an update propagation element. Since cache element of $T_{1,0}$ is from cache side *1*, the server does not need to send the update propagation element, but it needs to normalize the cache element by updating the position of the cache element. Then SVQ will be like this,



The server creates and sends an update propagation element to cache *2*. The update propagation element for cache *2* contains the value of object x. Then the server updates the position of cache element of $T_{2,0}$. SVQ will be shown as follows,



Eventually, the server removes the validated element of $T_{1,1}$ from SVQ, because it is on the top of SVQ.

3.4 The Design of the Proposed Scheme

3.4.1 Cache Transaction Model

A cache transaction is a transaction associated with a cache. It is a long running transaction at the server. Its life span is equal to the life of an associated client cache. The cache transaction is the representation of the client cache at the server. Furthermore, it can be considered as a parent transaction of all transactions from the client. Our cache transaction model is similar to the *envelope transaction* model of Wilkinson and Neimat, 1990.

There are some properties of a cache transaction. Those are listed as follows:

- Cache Transaction identifier: one may take its network address as the cache transaction identifier.
- Cache address: a network address of the local cache manager.
- Sequence number: an incremented number maintained by the scheduler for a cache transaction. This number is incremented when the scheduler creates an update propagation element for this cache transaction.
- List of active transactions: a pointer to linked lists of active transactions at the server.

3.4.2 Client Transaction Model

The transaction model used in our scheme is a flat model. It consists of begin of transaction (BOT), reads, writes, commits or end of transaction (EOT). BOT is operation to start a new transaction. The client application requires to submit BOT to the system (or local cache manager) because there are some actions that the system should do for a new transaction. After receiving a transaction identifier for a new transaction, the client application may submit some read and write operations with the transaction identifier. The actions of a transaction are ended by EOT. When a transaction submits its EOT, it means the transaction submits its commit request. All of its readset and writes are attached to the request.

The response EOT operation may be successful or failure. If the EOT response is failure (ABORT_REQ), then all operations of the transaction are undone. The client application may restart the transaction or create a new transaction. The restart transaction is treated the same as a new transaction. If the EOT response is successful (COMMIT_ACK), then all transaction operations are committed.

There are some important properties of a transaction:

- Transaction identifier: consists of a cache transaction identifier and incremented number maintained by local cache manager.
- Elements: point to the linked lists of the transaction elements.
- Client address: a network address of the client application.

Transactions which do not write objects are called as *read-only transactions*. Meanwhile transactions which update one or some objects are called *update transactions*.

3.4.3 Elements

An element contains the transaction identifier (TID), the element type, one or more object identifier fields (such a field contains the list of object identifiers to be accessed and other information) and links for queue management. The element and its fields will be depicted as follows:

TID	Element Type	Object Identifiers	Links

There are 7 elements in the system. Those are as follows:

- **Read element:** corresponds to read request of a transaction. It consists of a list of objects to be read (readset).
- **Commit element:** corresponds to commit request of a transaction. It may contain a list of objects to be written (if any).
- **Update Propagation element:** corresponds to the updates of a remote transaction. It consists of readset and writeset of a remote transaction.
- **Validated element:** represents a validated transaction. It consists of readset and writeset of a transaction.
- **Local Validated element:** It is the same as Validated element, but it represents a local validated transaction.

- **Cache element:** represents a cache transaction. It consists of list of objects cached by a client.

Elements are mutable and movable. Some elements are created by the system as corresponding to a client transaction's request; multiple read requests and a commit request, or corresponding to a cache transaction's request; cache, fetch and drop request. The other elements are produced from a combination of other elements of the same transaction; such as Validated element is a combination of some Read elements and one Commit element; two elements from the same transaction can be merged to be a compound element by merging their operations. An Update Propagation element is created at the server for a specific client. It is sent to the client through a message.

3.4.4 Cache Transaction Execution

To start a cache, cache manager sends a cache request (CACHE_REQ) to the server. The server responds the request with a cache acknowledge (CACHE_ACK) and a unique identifier of the cache transaction. This cache transaction identifier together with increment local identifiers builds a client transaction identifier. After a cache transaction has been created, it may issue some operations or actions.

A cache transaction may issue fetch and drop operations. A fetch operation is required by a cache transaction to fetch an object from the server. A drop operation is to remove a cached object from the client cache. These operations are sent to the server, and the server executes them. A fetch operation requires server to send a persistent object to the client that submits the fetch operation. A drop operation is necessary for enabling server to trace cached objects at clients. To keep the number of messages minimum, a client submits a fetch request together with its drop request (if any).

When a cache manager finds that a requested object is not found in its local cache, it creates a fetch request message for the object. At this time if the number of cached objects is greater than the size of cache, a drop request is created and added to the fetch request message. Fetch and drop requests basically contains a list of objects. A fetch request contains a list of objects to be cached and a drop request contains a list of objects to be removed from the client cache. Both requests can be sent to the server at once by creating a message with two lists of objects.

To have refreshed cached objects, the local cache manager may receive some *update propagation messages*. These messages are sent by the server as an update transaction has been committed at the server. An update propagate message contains a readset and writeset with the new values, and a *sequence number* of the cache transaction. This sequence number is an incremented number maintained by the server whenever the server creates an update propagation message for the cache transaction. It increments the sequence number and sends it to the local cache manager together with the update propagation message. The purpose of this sequence number is to make sure that when the local cache manager sends a commit request of a transaction, it has seen the most recent update propagation messages sent by the server. Therefore the local cache manager attaches the commit request messages with the most recent sequence number it has seen. If a commit request contains a sequence number too low, the server rejects the request, and asks the local cache manager to verify if the transaction should be committed and then resend the request with the most recent sequence number.

3.4.5 The execution of Cache Manager

The cache manager may receive requests from the client and the server. Those requests are listed as follows:

- Start of cache session: a request from a client to start a cache session.
- Start of transaction: a request from a client application to start a new transaction.
- Read request: a request from a client transaction to read some objects.
- Commit request: a request from a client transaction to commit transaction actions.
- Verify request: a request from the server to verify a commit of a transaction.
- Abort request: a request from the server to abort a transaction.
- Update Propagation request: a request from the server on behalf of remote update transaction. This request contents of readset and updates of the remote transaction.

The rest of this subsection describes each of these requests and actions taken by the local cache manager to respond the requests. We may write a pseudo code for some requests.

Start of cache session. This is automatically sent by a client when it starts to execute the application. This initiates Cache Manager and Cache Object Manager module to start their execution. At the beginning of its execution, the local cache manager needs to create a new cache transaction at the server. Therefore it sends a Cache Start request to the server. After the local cache manager is receiving an acknowledgement of Cache Start request, the client can start a new transaction.

Start of transaction. When the client application wants to create a new transaction, it should send a start of transaction request; other researchers may refer this request as Begin of Transaction (BOT), to the local cache manager. As a reply of this request, the local cache manager sends a transaction identifier; the cache transaction identifier and incremented number. The cache transaction identifier can be a network address of the cache side. To create a unique transaction identifier, the local cache manager maintains an incremented number. It increments this number whenever it creates a new transaction.

Whenever the local cache manager receives a request from a client transaction, it creates a corresponding element and posts it into CVQ. Afterward the element sent to Cache Object Manager for execution. The execution of cache manager and cache object manager is parallel. After the local cache manager sends any element to the cache object manager, it may serve another request from the client or server. However, local cache manager serves a request at a time.

```
Receive a read request message;
Create read element based on read request message;
if (requested objects available at local cache)
{
        Insert read element into CVQ;
        Record the object access for cache replacement strategy;
        Send read request message to Cache Object Manager;
}
else
{
        Insert read request message into Blocked list;
        Fetch necessary objects to the server;
}
```

Figure 5: Processing a Read Request at Cache Side

Read request. Figure 5 shows the process of a read request at cache side. When the local cache manager receives a read request message, it creates a Read element. If the requested objects are available at local cache, it inserts the element into CVQ, and records the access to associated objects for the purpose of cache replacement strategy. Afterward it sends the read element to the cache object manager for the execution of the read operations contained in the read element. Eventually, the client gets the values of the requested objects as soon as the local cache manager receives the result from cache object manager. Otherwise, if the requested objects are not found at local cache, the read request message is delayed and inserted into a blocked list. The local cache manager sends a request for fetching the not found requested objects to the server.

```
Create fetch request message
If (number of cached objects > cache size)
{       // Select objects to be dropped
        Sort cached objects based on time of last accessed
                          (from least to recently used);
        Select (number of cached objects - cache size) objects from the sorted objects;
        Put the selected objects in to the fetch request message;
}
Find out which objects needed to be fetched;
Put the objects needed to be fetched in the fetch request message;
Send the fetch request message to the server;
```

Figure 6: Creating Fetch Request Message

To fetch one or more objects to the server, the local cache manager submits a *fetch request message* to the server. Figure 6 shows the process of creating fetch request message. This message contains a list of objects to be fetched from the server. To reduce the number of round-trip message to the server, we design that a fetch request message may also contain a list of objects to be dropped from the cache. One or more objects are required to be removed from the cache. This is caused by the limitation of cache size. To decide which objects to be removed from the cache, any strategy, such as least recently use (LRU), can be used.

Whenever the local cache manager receives a fetch acknowledgement (FETCH_ACK) message, it forwards the message to the cache object manager for unloading the content of the message. Then the local cache manager updates its cached object information and examines the blocked list if any delayed request can be served.

```
Receive a commit request message;
Create Commit element based on commit request message;
Insert Commit element into CVQ;
If (validate())
{     // validation success
      If (is it a read-only transaction?)
      {     // read-only transaction
            If (ReadDirtyObjects?)
                  // the transaction read uncommitted objects
                  Set element type to Local Validated;
            else
                  // the transaction read committed objects
                  Set element type to Validated;

            Send commit acknowledgement to client;
      }
      else
      {     // update transaction
            Set element type to Local Validated element;
            If (is it No Wait Commit option or in disconnected mode ?)
                  Send Local Commit acknowledge to the client;

            If (can submit commit to the server?)
            {
                  Put the current sequence number to the commit request message;
                  Send commit request message to the server;
            }
            else
                  Suspend commit submission;
      }
}
else
{     // validation failure
      Remove all elements of the validating transaction;
      Send abort message to the client;
}
```

Figure 7: Processing Commit Requests at Cache Side

Update Propagation request. Whenever an update transaction is committed at the server, the server creates and sends an update propagation message to each client that caches any object updated by the transaction. The update propagation message contains a sequence number, the list of pair object id and its new value, and the client id.

When local cache manager receives an update propagation message, it creates a corresponding element, an Update Propagation element, replaces its old sequence number with the new one which is included in the message, and inserts the element into CVQ. Eventually, it forwards the message to Cache Object Manager for refreshing the cached objects. The update propagation elements in CVQ represent the execution order of remote update transactions at cache side. The existence of them in CVQ is important for the correctness. The Update Propagation element will be removed if it is on the top of CVQ.

Commit request. Commit request is a request from a client transaction to validate its actions; for read-only transactions, this is to validate whether its reads is in correct way; for update transactions, this is to validate and to make its updates permanent and available to others. The validation algorithm at cache side is described in subsection 3.3.1; see Figure 12 for the pseudo-code; here we define it as a function named it as **validate()**. In this subsection, we describe the execution of the cache manager whenever it receives a commit request from the client transaction.

When it receives a commit request message from the client transaction, the local cache manager creates a corresponding element; a Commit element (see Figure 7). After inserting the element into CVQ, it validates the transaction by invoking the function **validate()**. Suppose a transaction succeeds the local validation process. If it is a read-only transaction, then all its elements are merged to be a Validated element and its client is notified. Otherwise, if it is an update transaction, then the process is as follows:

i. It's all elements are merged to be a Local Validated element;

ii. a local commit acknowledge is sent to its client transaction if the client is under asynchronous commit strategy or under disconnected mode; otherwise nothing is sent to the client transaction;

iii. its commit request message is forwarded to the server for final validation, if it runs under connected mode or no conflict with suspended transactions (if any). Otherwise its commit is suspended. The suspended commits will be examined when the cache manager receives any response from the server.

Verify request. A cache manager receives a Verify request from the server whenever the server finds that the cache manager sends a commit request message with an invalid sequence number. We say the sequence number of a commit request is invalid if it is not equal to the sequence number of the update propagation element of the cache transaction at the server. This may happen when the local cache manager and the server are about the same time sending a commit request and update propagation respectively.

The purpose of the Verify message is to ask the local cache manager whether the commit request would be resent. The content of Verify request message is the same as the commit request sent by the local cache manager. Upon receiving the Verify request message, the local cache manager checks whether the commit transaction has been aborted. If the transaction has been aborted, then the local cache manager does not need to do anything; it just neglects the Verify message. Otherwise, it resends the commit request of the transaction with a new sequence number.

3.4.6 The Execution of Scheduler

Scheduler is a collection of programs that synchronize accesses to persistent objects at the server and maintain cache consistency. Similar to the local cache manager, the Scheduler uses a structured queue to synchronize accesses to persistent objects; we name the queue as Sever Validation Queue (SVQ). The accesses to persistent objects are recorded in SVQ. Therefore, whenever it receives a access request message, it creates a corresponding element and inserts it into SVQ. The scheduler may receive the following requests from the clients:

- Cache Start requests; requests to start a client cache session,
- Cache Finish requests; requests to finish a client cache session,
- Fetch requests; requests to fetch or to drop objects of the client cache,
- Commit requests; requests to commit an update transaction,

Therefore SVQ may content of Cache, Read, Commit, or Validated elements. **Cache**

Start request. When it receives a Cache Start request from a client; a Cache Start request is a request to start a cache session, the scheduler creates a new cache transaction (see sub section 3.2.1) and its new *Cache element* of the cache transaction, and then inserts the element into SVQ. A Cache element contains information about the objects stored at a cache represented by the cache transaction. This element is associated with a cache transaction.

```
Receive a fetch request message;
If (the fetch request contains a list of objects to be dropped)
{
    Drop objects in the list of writeset from the associate Cache element in SVQ;
    Remove the dropping list of objects from the fetch request;
}
Create a commit element of the cache transaction;
Insert the commit element into SVQ;
If (validate())
    Send the fetch request to Object Manager;
Else
    Put the fetch request into a blocked list;
```

Figure 8: Processing a Fetch Request at Server Side

Fetch request. When the scheduler receives a fetch request message from a client, it examines the message. As previously mentioned (see sub section 3.2.6) that a cache manager may submit a fetch request and a drop request in one request which is a fetch request. Therefore a fetch request contains a list of objects to be fetched and it may contain additional list of objects to be dropped. The fetch requests are considered as read operation of a cache transaction. Therefore, the scheduler needs to synchronize the fetch requests as follows (see Figure 8). First of all, it checks whether the fetch request contains a list of objects to be dropped. If the fetch request contains a list of objects to be dropped, then the server updates the list of objects in the Cache element of the cache transaction. Afterward, the server creates a Commit element of the cache transaction. The commit element contains, in its readset field, the list identifiers of the objects to be cached; its writeset field is empty. Then the scheduler inserts the Commit element into SVQ and validates the cache transaction by considering its Cache element as its first Read element. If the cache transaction fails the validation process, the fetch request will be delayed and retried later. Otherwise, the Cache element and the Commit element are combined to be a new Cache element of the cache transaction.

```
Receive a commit request message;
If (commit request message does not carry the most recent sequence number)
{
      Send Verify request message to originated cache manager;
}
else
{
      Create Read element of committing transaction;
      Insert the Read element right after the corresponding Cache element in SVQ;

      Create Commit element of committing transaction;
      Insert the Commit element into SVQ;
      If (validate())
      {
            Send the commit message to Object Manager;
      }
      else
      {
            Remove all elements of the committing transaction;
            Send abort message to originated cache manager;
      }
}
```

Figure 9: Processing Commit Request at Server Side

Commit request. A commit request at the server side is a request to validate an update transaction. When the server receives a commit request of a transaction, it behaves as follows (see Figure 9): if the sequence number carried by the commit request message is not equal to the sequence number recorded on its cache transaction at the server, the commit request is sent back to its cache manager for verification; else the server creates two elements. These are a Read element and a Commit element. The Read element contains the list of object identifiers that have been read by the update transaction. The Read element will not be executed; it is needed for the validation purposes only. Meanwhile, the Commit element contains the list of object identifiers that the transaction wants to update. The Read element is inserted into SVQ at the position right after the position of the Cache element of the associated cache transaction in SVQ. The Commit element is inserted at rear of SVQ. To validate an update transaction, the server invokes the validation algorithm (see in more detail in subsection 33.2); here we use function **validate()** to invoke the validation algorithm which its pseudo-code is written in Figure 13. If the validation is successful, the server sends a commit acknowledgement message to the originated cache and then executes the updates of the transaction by sending the commit element to Object Manager. Otherwise it aborts the transaction and sends an abort message to the originated cache manager.

```
Receive an execution acknowledgement of commit element  from Object Manager;
ValidatedElement = get corresponding validated element of the transaction
                                from SVQ;

for each CacheElement in front of ValidatedElement in SVQ
{
    If (ValidatedElement conflicts with CacheElement)
    {
        Create UpdatePropagationElement for originated client of CacheElement;

        If(CanProcessUpdatePropagation(CacheElement tid,
                                                ValidatedElement tid)
        {
            Create update propagation message for the client of CacheElement;
            Send the update propagation message to Object Manager;
        }
        Else
            Pending propagate update to the client
    }
}
```

Figure 10: The First Step of the Update Propagation Process

After the scheduler receiving an execution acknowledgement of the commit element, it has to propagate the updates to each cache side. The objectives of the update propagation process are not only to distribute the updates to each cache side, but also to execute the update transaction automatically at each cache side. Since each cache side caches objects differently, the update propagation element for each cache side is unique. Therefore, the scheduler creates a single update propagation element for each cache side. The scheduler executes the process of update propagation in two steps; the first step to create an update propagation element and to read the values; the second step is to update the related cache element in SVQ and to send the update propagation to the cache side. Figure 10 shows the pseudo-code of the first step of the update propagation process. The cache side in which its Cache element is located at the front of the validated element of the update transaction in SVQ will be sent an update propagation element with two conditions should be satisfied:

- it caches any object updated by the update transaction, and
- the function of CanCreateUpdatePropagation (see Figure 14) returns true.

If these two conditions are satisfied by the cache side, then the process of update propagation proceeds. Otherwise it is delayed.

The second step of the update propagation process is begun when the scheduler receives an acknowledgement of the read execution of the update propagation element from Object Manager. The pseudo-code of this step is shown in Figure 11. In the second step, the update propagation element has been loaded by the values of the objects. Now, the scheduler sends the element to the cache side after inserting the incremented sequence number of the associated cache transaction to the element, and updating the position of the cache element of the cache transaction in SVQ (see Figure 15). Afterward it sends the update propagation element to the cache side. If the client runs under disconnected environment, then the update propagation element is inserted to its cache transaction's list; otherwise, it is sent to the cache side.

```
Receive a read execution acknowledgement of update propagation
                                                from Object Manager;
If (CanProcessUpdatePropagation(Cache trans id, update trans id)
{
     Increment the sequence number of the associated cache transaction;
     Updating the position of CacheElement in SVQ;
     If (the associated cache transaction running under connected mode)
          Send update propagation message to the client;
     Else
          Put the update propagation element on the cache transaction's list;
}
Else
     Pending the process of update propagation in second step;
```

Figure 11: The Second Step of the Update Propagation Process

Cache Finish request. This request is submitted to the server when the client wants to finish its session. To finish a client cache session, the server needs to assure that that all requests from the client have been finished. Then, it deletes the associated cache transaction and sends a Cache Finish acknowledgement message to the client.

3.4.7 The Pseudo Code of Cache Side Validation Algorithm

The rest of this subsection describes the pseudo code of the cache-side validation algorithm. Figure 12 shows the pseudo code of the cache side validation algorithm. The pseudo code shows how to traverses CVQ in validating a transaction. It checks the elements of the validating transaction against its intervening elements. Initially, it sets the first read element of the validating transaction to FIRST and the commit element of the validating transaction to SECOND. From the position of FIRST, it traverses RC queue toward to the rear of the queue and checks if FIRST conflicts with its in-between elements of other transactions. If FIRST does not conflict with its in-between elements from other transactions, then merge FIRST and the read element of the validating transaction, let this merged element be FIRST and places FIRST to the position of the last read element. The algorithm continues checking FIRST against its in-between elements. If it reaches SECOND which is the commit element of the validating transaction, then it returns **success.**

If a conflict founds and the validating transaction is an update transaction, then the validation algorithm returns failure; otherwise move FIRST to the front of the conflicting element. Now traverse CVQ from SECOND toward FIRST and check if SECOND conflicts with its in-between elements of other transactions in the same way as described above. If SECOND conflicts with its in-between elements, then the algorithm returns **failure** after it removes all elements of the validating transaction. Otherwise; SECOND does not conflict with its in-between elements and reaches FIRST, then the algorithm returns **success**.

```
FIRST = get the first read element of the transaction;
SECOND = get the commit element of the transaction;
NEXT = get element after FIRST element;
for (;;)
{
   If (NEXT is another read element of the transaction)
   {
         Remove FIRST element from the CVQ;
         FIRST = merging FIRST and NEXT then replace NEXT element with FIRST;
         NEXT = get next element of NEXT;
   }
   else if (NEXT is equal to  SECOND)
   {
         Remove FIRST element from the CVQ;
         Merging FIRST and SECOND store into SECOND as local validated element;
         Return success;
   }
   else if (FIRST conflict with NEXT)
   {
         If (it is an update transaction)
               Return failure;

         /* the transaction is read-only transaction */
         /* move FIRST to the position before NEXT */
         Remove FIRST from the CVQ;
         Insert FIRST before NEXT in the CVQ;
         PREV = get previous element of SECOND;
         for (;;)
         {
               If (PREV is equal to FIRST)
               {
                     Merging SECOND and FIRST store at the position of FIRST as local
                              validated;
                     Return success;
               }
               else if (PREV is another element of the transaction)
               {
                     Remove SECOND from the Queue;
                     SECOND = merging SECOND and PREV store at
                                                    the position of PREV ;
                     PREV = get previous element of SECOND;
               }
               else if (SECOND conflict with PREV)
               {
                     Remove FIRST, SECOND, and all the remainder elements of the
                                 transaction;
                     Return failure;
               }
               else
                     PREV = get previous element of PREV;
         }
   }
   else
         /* FIRST is not conflict with NEXT*/
         NEXT = get next element of NEXT;
}
```

Figure 12: The Cache Side Validation Algorithm

3.4.8 The Pseudo Code of Server-Side Validation Algorithm

The main objective of server side algorithm is to validate update transactions at the server and to maintain cache consistency. To validate an update transaction, the server checks the execution of the update transaction whether interleaves with the execution of other update transactions at the server. To maintain cache consistency, the server should distribute the updates of an update transaction to each cache side which caches any object updated by the update transaction. In this sub section, we describe the function **validate()** and CanCreateUpdatePropagation() which are mentioned in subsection 3.2.7.

```
FIRST = get the first read element of the transaction;
SECOND = get the commit element of the transaction;
NEXT = get element after FIRST element;
for (;;)
{
  if (NEXT is equal to  SECOND)
  {
        Remove FIRST element from the CVQ;
        Merging FIRST and SECOND store into SECOND as local validated element;
        Return success;
  }
  else if (FIRST conflict with NEXT)
  {
        Remove FIRST and SECOND from SVQ;
        Return failure;
  }
  else
        /* FIRST is not conflict with NEXT*/
        NEXT = get next element of NEXT;
}
```

Figure 13: The Server Side Validation Algorithm

Figure 13 shows the pseudo-code of the function **validate()**. As mentioned before in this chapter, a commit request contains the readset and writeset of the update transaction. The server creates and inserts the read element of the commit request into right after the position of the cache element of the associated cache transaction in SVQ, and creates and inserts the commit element of the update transaction at rear of SVQ. The validation algorithm checks whether the read element can be combined with the commit element to be a validated element. If the execution of update transaction interleaves with other updates transaction, then there must be any validated element of other update transactions in between the read element and the commit element which conflicts with the read element. Therefore the read element and the commit element of the update transaction cannot be combined to be one element. In this case, the update transaction fails the validation process. Otherwise, the update transaction succeeds the validation process.

The rest of this subsection describes the pseudo-code of the function CanCreateUpdatePropagation and updating the cache element position in SVQ. These two pseudo-codes are invoked at the process of update propagation which is described in subsection 3.2.7. The first pseudo-code (Figure 14) is to check whether the cache element can be moved to the position after the validated element or the validated. Meanwhile the second pseudo-code (Figure 15) is to update the position cache element of the cache transaction to the position right after the validated element.

```
Function CanCreateUpdatePropagation(cache trans id, update trans id)
{
    Cache Element = get the cache element of the associated cache transaction
                                                                    in SVQ;
    ValidatedElement = get the validated element of the update transaction in SVQ;

    NEXT = get element after CacheElement in SVQ;
    For (;;)
    {
        If (NEXT is equal to ValidatedElement)
            Return true;
        Else (NEXT conflict with CacheElement)
        {
            PREV = get the previous element of ValidatedElement in SVQ;
            For (;;)
            {
                If (PREV conflicts with ValidatedElement)
                    Return false;
                Else if (PREV is equal to NEXT)
                    Return true;
                Else
                    PREV = get the previous element of NEXT in SVQ;
            }
        }
        Else
            NEXT = get the next element of NEXT;
    }
}
```

Figure 14: The function of CanCreateUpdatePropagation

```
CacheElement = get the cache element of the associated cache transaction in SVQ;
  ValidatedElement = get the validated element of the update transaction in SVQ;
  NEXT = get the next element of CacheElement;
  For (;;)
  {
        If (NEXT equal to ValidatedElement)
        {
              Move CacheElement to right after the position of ValidatedElement
                                                                    in SVQ;
              Break;
        }
        Else if (CacheElement conflicts with NEXT)
        {
              Move CacheElement at the position before NEXT;
              Move ValidateElement to the position before CacheElement;
              Break;
        }
        Else
              NEXT = get the next element of NEXT;
  }
```

Figure 15: Updating Cache Element position in SVQ.

3.5 The Serializability of The proposed Algorithm

The correctness of the proposed algorithm is described in this section. This section is divided in three subsections. The first subsection describes some fundamental concepts that are used to prove the correctness of the proposed scheme. The second subsection is designated to describe the correctness of ROCC algorithm; note that the proposed scheme is an extension of ROCC, therefore it is necessary to describe the correctness of ROCC before proving the correctness of the proposed scheme. The following subsection is to prove the correctness of the proposed scheme.

3.5.1 Definitions

A transaction T_i may request either read or write access to data objects x, denoted as $r_i(x)$ or $w_i(x)$ respectively. The requests submitted to the system can be considered as a collection of the accesses. Without lack of generalization, we can consider the collection of the accesses as an *element*. Note that an element may contain one operation (read or write). The j-th element of transaction T_i is denoted as element e_{ij}. In this research, we define the formal definition of element as follows,

Definition 1 An element e_{ij} is the j-th element from transaction T_i where:

1. e_{ij} is a subset of $\{r_{ij}(x), w_{ij}(x) \mid x \text{ is an object}\}$
2. $rs(e_{ij}) \cap ws(e_{ij}) = \varnothing$, where $rs(e_{ij})$ is a readset and $ws(e_{ij})$ is a writeset.

In words, condition (1) defines the kinds of operation in the element. Condition (2) says that read and write operations on the same object cannot be in the same element. Two or more elements of the same transaction can be merged to build a compound element. The compound element is not executed by the system, but it is used for concurrency control purposes. The merge operation on elements is defined as follows,

Definition 2 If element e_{ir} is a compound element, built by merging element e_{ip} and element e_{iq}, then $ws(e_{ir}) = ws(e_{ip}) \cup ws(e_{iq})$ and $rs(e_{ir}) = rs(e_{ip}) \cup rs(e_{iq})$.

The compound element is not necessary to satisfy the condition (2) of an element (see Definition 1). Note that two elements from different transaction cannot be merged.

Definition 3 Element e_{ij} conflicts with element e_{pq} if and only if $i \neq p$ and one of the following statements is true:

- $rs(e_{ij}) \cap ws(e_{pq}) \neq \{\}$, or
- $ws(e_{ij}) \cap ws(e_{pq}) \neq \{\}$, or
- $ws(e_{ij}) \cap rs(e_{pq}) \neq \{\}$.

In other words, element e_{ij} conflicts with element e_{pq} if and only if they are not from the same transaction ($i \neq p$) and both access the same object (at least one object) and at least one of them is write operation.

Definition 4 A transaction T_i is partial order with ordering relation $<_i$ where:

1. $T_i = \{ e_{i1}, e_{i2}, \ldots, e_{in} \} \cup \{a_i, c_i\}$,
2. a_i (abort) is member of T_i only if c_i (commit) is not member of T_i,
3. if t is c_i or a_i, for any element e_{ij} in T_i, $e_{ij} <_i t$,
4. if $r_{ij}(x) \in e_{ij}$ and $w_{ik}(x) \in e_{ik}$, then $e_{ij} <_i e_{ik}$.

Informally, (1) transaction T_i is a set of element and abort or commit operations. (2) If the transaction executes an abort operation, then it does not execute a commit operation. (3) If an operation *t* is abort or commit operation, then the ordering relation defines that for all elements precede operation *t* in the execution of the transaction. (4) If both read and write operations are executed to the same object, then the ordering relation defines the order of the execution of the correspondent element.

Definition 5 Transaction T_i conflicts with transaction T_j if and only if one of T_i's elements (or compound elements) conflicts with one of T_j's elements (or compound elements).

Definition 6 A *complete history H* over *T* is a partial order with ordering relation $<_H$ where:

1. $H = \bigcup_{i=1}^{n} T_i$;
2. $<_H \supseteq \bigcup_{i=1}^{n} <_i$; and
3. for any two conflicting elements *p, q* member of *H*, either $p <_H q$ or $q <_H p$.

Condition (1) says that the execution represented by *H* contains the elements submitted by transaction T_1, T_2, ..., T_n. Condition (2) says the global ordering relation supersets the ordering relation specified within each transaction. Condition (3) says that ordering every pair of conflicting elements is determined by ordering relation $<_H$. A *history* is simply a prefix of a complete history. Note that a complete history (or history) is defined over a set of committed transactions (Bernstein, P. A., Hadzilacos, V. and Goodman, N. 1987).

Definition 7 Let *H* be a complete history over $T = \{T_1, T_2, \ldots T_n\}$. The *Serialization Graph* (*SG*) for *H,* denoted as *SG*(*H*), is a directed graph whose nodes are the transactions in *T* and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i's elements precedes and conflicts with one of T_j's elements in *H*.

We can determine whether a history is serializable by analyzing the serialization graph. Suppose *H* is a complete history over $T = \{T_1, T_2, \ldots T_n\}$. The history *H* is serial if and only if the serialization graph *SG*(*H*) is acyclic (Bernstein, P. A., Hadzilacos, V. and Goodman, N. 1987).

Definition 8: Distributed serialization order (Bernstein and Goodman, 1981). A global history H is serializable if there is exist a total ordering of T such that for each pair of conflicting elements e_i and e_j from distinct transactions T_i and T_j (respectively), e_i precedes e_j in any $H_1, H_2, \dots, H_n$ if and only if T_i precedes T_j in the total ordering.

Intuitively, an execution is serial if there is a total order of transactions such that if T_i precedes T_j in H, then T_i's elements precedes T_j's elements in every local history H_i (where $i=1,2, \dots n$) where both appear. In other words, this says transactions execute serially and in the same order at all clients.

3.5.2 The Correctness of ROCC Algorithm

We now present the correctness of ROCC algorithm. To prove ROCC algorithm is correct, we have to prove that all histories representing executions that could be produced by it is serializable. Any history of ROCC algorithm can be proved by using the serialization graph.

To prove the correctness of ROCC algorithm, we must characterize the set of ROCC history, that is, those that represent possible executions of transactions that are synchronized by ROCC algorithm. ROCC records executions of transactions in RC queue. When ROCC executes an element of a transaction, it inserts the element into RC queue. The transaction may submit multiple Read elements and end its execution by submit Commit element. If the transaction succeeds the validation process, all transaction elements are united to be Validated element.

Proposition 1: Let H be a history produced by ROCC. If T_i's element is in H, it has only one element which is Validated element.

Using this properties, we must show that every ROCC history H has an acyclic $SG(H)$. Note that transactions in H are committed transactions. Therefore a transaction T_i in H has one element, e_i.

Lemma 1: Suppose there are a set of transaction $T = \{T_1, T_2, \dots, T_n\}$. A complete history H over T is produced by ROCC algorithm. A serialization graph SG is defined over H. If $T_i \rightarrow T_j$ is in $SG(H)$, then e_i; the Validated element of T_i conflicts with e_j; the Validated element of T_j, in H, $e_i <_H e_j$.

Poof: Since $T_i \rightarrow T_j$ is in $SG(H)$, then T_i conflicts with T_j and T_i precedes T_j. Transaction T_i conflicts with T_j if and only if e_i conflicts with e_j, such that $e_i <_H e_j$.

Lemma 2: Let H be a complete history produced by ROCC algorithm, and let $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ be a path in $SG(H)$, where $n > 1$. Then e_1 conflicts with e_n in H, $e_1 <_H e_n$.

Proof: The proof is by induction on n. The basis step, for $n=2$, follows immediately from Lemma 1. Suppose the lemma holds for $n=k$, for some $k \geq 2$. We will show that it holds for $n = k+1$. By induction hypothesis, the path $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k$ implies that T_1's element e_1 and T_k's element e_k in H, such that $e_1 <_H e_k$. By $T_k \rightarrow T_{k+1}$ and Lemma 1, T_k's element e_k conflicts with T_{k+1}'s element e_{k+1} such that $e_k <_H e_{k+1}$. By the last three precedences and transitivity, $e_1 <_H e_{k+1}$ as desired.

Theorem 1: Every ROCC algorithm history H is serializable.

Proof: Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$, where $n>1$. By Lemma 2, one T_1's element conflicts with another T_i's element in H. This contradicts Proposition 1 that the execution of transaction T_1 is equivalent with single element. Thus $SG(H)$ has no cycles and so H is serializable.

3.5.3 The Correctness of VQ Algorithm

To prove the correctness of VQ algorithm, we have to characterize the set of histories produced by VQ algorithm, that is, those that represent possible executions of transactions that are synchronized by VQ algorithm. To characterize VQ histories, we need to model VQ history. Let $T = \{T_1, T_2, \dots\}$ be a set of transactions in the system and H be a global history over T. There are n clients in the system. Each client caches necessary objects. Each client has a local cache manager which manages local requests. We define a *local history* H_k for client k as a partial order set over T.

Definition 9: Let H_k a complete history at cache side k; $k=1,2,\dots,n$, is a partial order over $T_k=\{T_{k,1},T_{k,2},\dots, T_{k,nk}\}$ with ordering relation $<_{Hk}$ where:

1. $H_k = T_{k,1} \cup T_{k,2} \cup \dots \cup T_{k,nk}$;
2. $<_{Hk} \supseteq <_1 \cup <_2 \cup \dots \cup <_{nk}$;
3. for any two conflicting elements $p,q \in H_k$, either $p <_{Hk} q$ or $q <_{Hk} p$.

In other words, condition (1) says that the execution represented by H_k involves precisely the elements submitted by $T_{k,1}$, $T_{k,2}$, …, $T_{k,nk}$. Condition (2) says that the execution honours all element orderings specified within each transaction. Finally, Condition (3) says that the ordering of every pair of conflicting elements is determined by $<_{Hk}$.

Suppose T_i participates at cache side and its elements are in H_k. If T_i is a local transaction, then its execution is equivalent to a single element in H_k and it is a validated element. If T_i is a remote transaction, then its execution is equivalent to a single element in H_k and it is an update propagation element. Therefore, if T_i participates at cache side k, then its execution is equivalent to a single element; for the simplicity, we denotes the element as e_i.

Proposition 1: Let H_k be a local history at cache side k produced by cache side algorithm of the proposed scheme. If T_i participates at cache side k, then the execution of T_i's elements at cache side k is equivalent to a single element, e_i.

Definition 9: Let T=$\{T_1, T_2, \ldots\}$ be a set of transactions, H is a complete history produced by VQ algorithm, and there are n cache sides in the system. History H is defined as a partial order over T with ordering relation $<_H$ where:

1. H=$H_1 \cup H_2 \ldots \cup H_n$, where H_k is a complete history at cache side k; H_k is partial order over T.
2. $<_H \supseteq <_{H1} \cup <_{H2} \cup \ldots \cup <_{Hn}$;
3. for any two conflicting elements $p, q \in H$, either $p <_H q$, or $q <_H p$.

In other words, condition (1) says that the execution represented by H involves the elements executed at H_1, H_2, …, and H_n. Condition (2) says that H honours all elements orderings specified within each cache side. Finally, condition (3) says that says that the ordering of every pair of conflicting elements is determined by $<_H$.

Note that conflicting update transactions from cache side k are submitted to the server one at a time. For example, if two conflicting transactions T_i and T_j run parallel at cache side k and T_j has been submitted first to the server. Then the commit of T_i to the server is delayed until the server responds the commit of T_j.

Proposition 2: Let H_k be a local history at cache side k; $k=1,2,\ldots, n$, an $T=\{T_1, T_2, \ldots\}$, H is a global history, and T_i and T_j are from cache side k. If $e_i <_{Hk} e_j$, then $e_i <_H e_j$.

Lemma 1: Suppose the number of clients is n, a set of transactions $T = \{T_1, T_2, \ldots\}$, and each client executes serial local history $H_1, H_2, \ldots, H_n$ based on VQ scheme. A VQ global history H is defined over T. If $e_i <_H e_j$, then $e_i <_{Hk} e_j$ in client k in which both transactions appear, $k=1..n$.

Proof: Suppose client i and j creates T_i and T_j respectively. If $e_i <_{Hi} e_j$, then e_i conflicts with e_j at client i. There three cases that e_i conflicts with e_j.

- e_i reads some objects at client i into which e_j subsequently updates ($rs(e_i) \cap ws(e_j) \neq \{\}$). This means that the update propagation of T_j is received by the local cache manager after T_i locally committed at client i. The commit of T_i must have preceded the commit of T_j at the server (see Figure 16). Otherwise transaction T_i carries an invalid sequence number to the server; consequently it is aborted. The execution of Update Propagation of T_i at client j must have preceded the commit of T_j at client j. Otherwise transaction T_j is aborted at the server. Therefore $e_i <_{Hj} e_j$ is hold at client j. Since we assume that messages delivered in first come first served basis, then $e_i <_{Hk} e_j$ for client k which both transactions appear, $k=1\ldots n$.

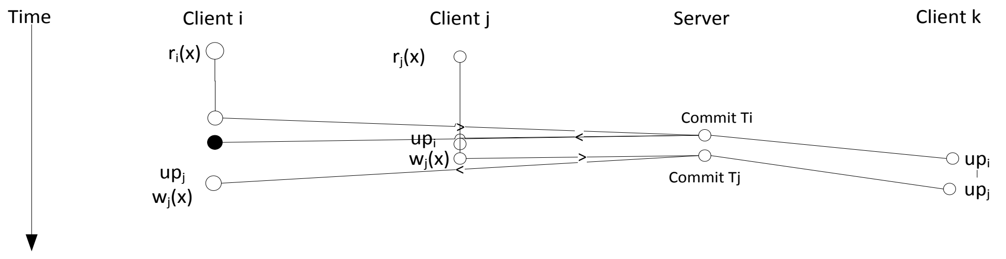


Figure 16: Case $rs(e_i) \cap ws(e_j) \neq \{\}$, $e_i <_{Hi} e_j$ at Client *i*

- e_i writes some objects at client i into which e_j subsequently reads at client j ($ws(e_i) \cap rs(e_j) \neq \{\}$). This means that the Update Propagation of T_i is received and executed by cache manager at client j before T_j reads the conflicting objects (see Figure 17.); note that an update transaction is not allowed to read stale objects. Consequently, the commit of T_i precedes the commit of T_j at the server. Therefore, $e_i <_{Hk} e_j$ for client k in which both transactions appear, $k=1..n$.

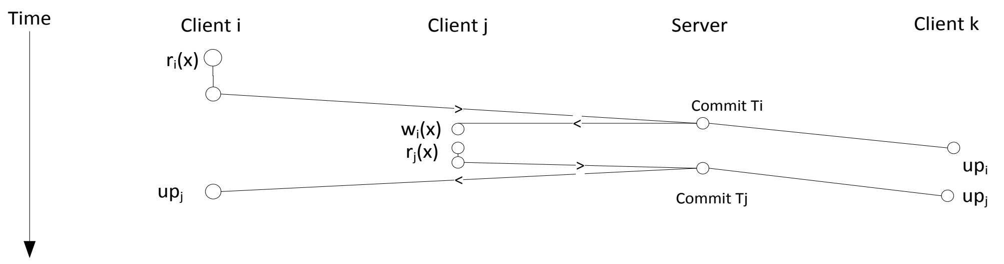


Figure 17: Case $ws(e_i) \cap rs(e_j) \neq \{\}$, $e_i <_{Hi} e_j$ at Client *i*

- e_i writes some objects at client i into which e_j subsequently updates ($ws(e_i) \cap ws(e_j) \neq \{\}$). This means that Update Propagation element of T_i must precede the T_j's reads of the conflicting objects at client j (see Figure 18). Consequently, the commit of T_i precedes the commit of T_j at the server. Therefore, $e_i <_{Hk} e_j$ for client k in which both transactions appear, $k=1..n$.

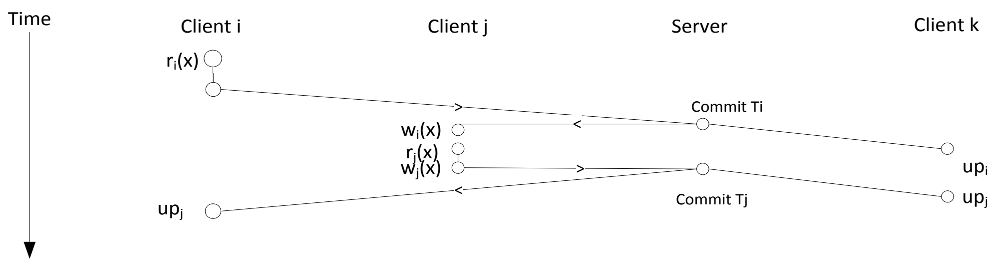


Figure 18: Case $ws(e_i) \cap ws(e_j) \neq \{\}$ of $e_i <_{Hi} e_j$ at Client *i*

Since all cases above show that if $e_i <_{Hi} e_j$ at client i then $e_i <_{Hk} e_j$ at client k for $k=1..n$. Therefore, if $e_i <_H e_j$, then $e_i <_{Hk} e_j$ in client k in which both transactions appear, $k=1..n$ is hold.

Lemma 2: Suppose there are a set of transaction $T = \{T_1, T_2, \ldots\}$. A complete history H over T is produced by VQ algorithm. A serialization graph SG is defined over H. If $T_i \rightarrow T_j$ is in $SG(H)$, then e_i; the Validated element of T_i conflicts with e_j; the Validated element of T_j, in H, $e_i <_H e_j$.

Proof: If $T_i \rightarrow T_j$ is in $SG(H)$, then based on Definition 8 there exist e_i conflicts with e_j and e_i precedes e_j. Consequently $e_i <_H e_j$.

Lemma 3: Let H be a complete history produced by VQ algorithm, and let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ be a path in $SG(H)$, where $n > 1$. Then e_1 precedes e_n in H, $e_1 <_H e_n$.

Proof: The proof is by induction on n. The basis step, for $n=2$, follows immediately from Lemma 4. Suppose the lemma holds for $n=k$, for some $k \geq 2$. We will show that it holds for $n = k+1$. By induction hypothesis, the path $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k$ implies that T_1's element e_1 and T_k's element e_k in H, such that $e_1 <_H e_k$. By $T_k \rightarrow T_{k+1}$ and Lemma 4, T_k's element e_k precedes T_{k+1}'s element e_{k+1} or $e_k <_H e_{k+1}$. By the last three precedences and transitivity, $e_1 <_H e_{k+1}$ as desired.

Theorem 2: Every VQ history H is serializable.

Proof: Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, where $n>1$. By Lemma 3, one T_1's element conflicts with another T_1's element in H. This contradicts Proposition 1 that the execution of transaction T_1 is equivalent with single element. Thus $SG(H)$ has no cycles and so H is serializable.

Chapter 4. Performance Evaluation

This chapter describes the experimental framework for evaluating the concurrency control scheme in client-server database systems which is discussed in the previous chapter. The experiments were performed with the help of a simulator. We used a simulation technique to evaluate and to compare the performance of our proposed scheme to other schemes in the presence of a large number of clients and varying the percentage of read-only transactions.

For the purpose of the comparison, we also implement Adya algorithm and MVCC algorithm which are described in chapter 2. The reason to choose Adya algorithm is that it is considered currently the best algorithm in client-server database systems with caching at client side, and it provides one-copy serializability or degree consistency 3 (see Franklin et al, 1997). Meanwhile, MVCC is chosen to represent Snapshot Isolation algorithms which are used and implemented by many commercial systems, such as INFINISPAN (http://www.jboss.org/infinispan). Note that MVCC does not provide one-copy serializability, its degree consistency is 2.

Originally, Adya algorithm employs *invalidation* to maintain cache consistency among clients. In invalidation, the server sends an invalidation message to each client that caches any object updated by the transaction to drop the object from the client cache. Here, we implement Adya algorithm; as the same with VQ and MVCC, with using *propagation* as their cache consistency protocol. In propagation, the server sends a propagation message to each client that cache any object updated by the transaction to update object at the client cache. Therefore, the client can keep caching the object. This does not make a significant change to Adya performance results.

To use simulation study to compare the proposed algorithm to other algorithms, it is necessary to model system components, such as client, server, database, and network. We refer to the model of system components as a *system model*, while the *workload model* captures the way that transactions run against database objects, and the nature of these transactions. Each model has a set of parameters, to allow us to vary, e.g. the number of clients, or the percentage of read-only transactions. Subsection 4.2 describes our system model

We constructed our simulation study and the workload from earlier concurrency control study Gruber (1997). His study was performed for a single-server, multi-client system. The simulator scheduling model has been borrowed from his study.

4.1 Simulation Tool

We use the Objective Modular Network Test-bed (OMNET++) simulation engine to implement the simulation model. It is a public source, component-based, modular and open architecture simulation environment with strong GUI support. Its main application area is the simulation of communication networks, but because of its generic and flexible architecture, it has been successfully used in other areas.

The OMNET++ model consists of hierarchically nested modules. The top level model is the system model, which covers the complete simulation model and is referred to as the “networks”. The system contains sub-modules which themselves may have sub-modules. Thus the modules can be described to any depth of nesting as a result able to describe complex system models as a combination of a number of simple modules. Modules that contain sub-modules are called compound models. Simple modules contain the algorithms in the modules and form the lowest level of module hierarchy. The user implements the simple modules in C++, using the OMNeT++ simulation class library. Modules communicate by message passing which may be a complex data structure.

Modules may send messages directly to their destination or through a series of gates and connections to other modules. The messages can represent frames or packets in a computer network simulation. The local simulation time advances when the module receives messages from other modules or from the same module as selfmessages, which is the representation of timers in simulation world. These self messages are used to schedule events to be executed by itself at a later time. Each of the modules has input and output interfaces called Gates through which message passing between modules is achieved. Messages are sent out through the out-Gate and received through the in-Gate. Connections are created between the sub-modules or between sub-module to compound module depending on the requirement of the system or the topology.

The description of the topology, the structure and specification of the modules, the Gates and connections are specified through the Network Description Language (NED). NED files are not used directly: they are translated into C++ code by the

NEDC compiler, then compiled by the C++ compiler and linked into the simulation executable. The actual behavior of the modules is written in C++ code using the OMNeT++ simulation library and the description of the modules:- parameters, Gates , connections between different modules, is specified by the NED language. In this way, there is a separation of behavior and interface definition. This allows reusability of module interfaces defined by NED code. For the implementation of the simple modules OMNeT++ offers an API consisting of a simple module interface, a message interface and a rich simulation library providing support for essential functions, as a lot of routines for the simulation purposes as e.g. I/O-functions, statistics-classes for gathering the achieved results, etc. but also more general stuff like statistical distributions, random numbers generators and even container classes like queues, stacks, containers, etc. The simulation tool allows the collection of the final results and also the statistics of the performance of the simulation transparently into scalar and vector files.

4.2 Assumptions for Simulation

Followings are the assumptions that are adopted for our simulation study:

- We assume the client-server database system with a single server and many clients. Clients are connected to the server through a network.
- Each client application issues a single transaction at a time.
- We assume that clients store objects in main memory. Client-side disk caching is not considered in our study. Presence of disks at client will affect the local data capacity and response time, but it is not expected to alter the relative performance of the different concurrency control schemes.
- We assume the server memory is large enough memory to keep all objects. This assumption holds for many applications and systems; current memory trend and recent technology (memcached; http://www.memcached.org/).
- We assume propagation updates than invalidation objects. Adya algorithm originally uses invalidation objects. In our simulation study, we implement Adya algorithm with propagation updates. Invalidation forces clients to drop the objects which are listed on the invalidation message. While propagation allows

clients keep caching the objects. This is not expected to alter the performance of Adya algorithm significantly.

- MVCC transactions are always aborted if a conflict is detected with concurrent update transactions.

4.3 System Model

To simulate a client-server database system accurately, it is necessary to model all components that can affect performance in a significant manner. We model each system component, such as Scheduler, Cache Manager, Client, etc. as a module in our simulation model. A service request to the system component is submitted through an explicit message. Upon receiving any request message, the module of the system component inserts the message into a queue if it is busy; otherwise it serves the request message. The module serves one message at a time. When it finishes serving any message, it takes another message from the queue; if there is no message in the queue, then it sets itself as an idle.

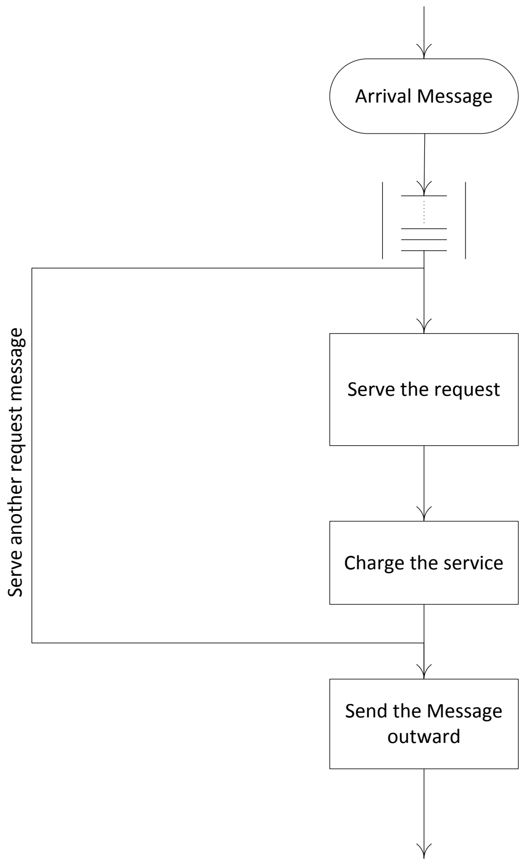


Figure 19: The Flowchart of a Module

Table 1: System Parameter Setting

Database Parameters	
Parameter	**Value**
Object Size	1 mb
Database Size	1000
The number of locality region	5
Client and Server Parameter	
Client CPU Speed	50 000 MIPS
Server CPU Speed	100 000 MIPS
Client Cache Size	25% of database size
Server Cache Size	100% of database size
Network Parameters	
WAN Network	100 MBPS
Fixed Network Cost	36000 instr
Variable Network Cost	43000 instr/KB
WAN Propagation Delay	Exponential(50ms)

A module may serve vary messages; for examples Cache Manager module may serve read request messages, commit request messages, update propagation messages, etc. To serve a request message, a module of a system component may invoke some functions. To charge the service of any request, we count the number of instructions executed by the module to serve the service; and convert this number of instructions to the CPU time consumption based on the speed of CPU. Then to emulate the CPU time of the service, we set the module in busy mode as long as the CPU time. The flowchart of the module is shown in Figure 19. We assume that a processor is always available for the module.

4.3.1 Database

We model database as a collection of objects. An object could be anything; it might contain data and procedures (or codes). Kim, Won in (Kim, 1990) defines object-oriented database very well. An object is an entity which has a unique identifier. We assume an object does not have a specific relation with other objects. For the sake of simplicity, we use in-memory object-oriented database at the server, and all objects are already in the server memory when the simulation is initiated.

We assume that an object size is 1 MB, and a request may contain 1 to 5 objects. We use objects as the smallest granularity locks for MVCC scheme.

4.3.2 Client

We model a client as a collection of modules; Application module, Cache Manager module, and Cache Object Manager module. Application module is a module to represent a client application. The main task of the Application module is to create a transaction, to emulate the client application, and to collect statistics. The Cache Manager module is a module to emulate Cache Manager. Meanwhile, the Cache Object Manager module is a module to emulate Cache Object Manager.

These three modules execute concurrently. They communicate each other through an explicit message. They form a compound module; Client Cache module. Each module occupies one processor. Thus, CPU is always available for the modules whenever they require CPU computations.

The processor speed at the client is chosen to be 50,000 MIPS (http://en.wikipedia.org). This speed corresponds to the amount of processing power can be devoted by the client for activities at the client side. We charge a requested service by calculating the CPU time. We calculate the CPU time with following formula:

CPU time = (the number of instructions for the service) / (Client CPU Speed)

We count the number of instructions for each service by using an approximation presented by Ahuja and Orlin (1992).

We choose the client cache size relative to the database size so that the concurrency control costs are not dwarfed by a high number of cache misses. The relative cache size is 25% of the database size. Gruber, et al. (1997) uses the cache size relative to the objects accessed by a client in the simulation. He uses the cache size 25% of the objects accessed by a client. Cache management is done using LRU.

4.3.3 Server

The same as clients, the server is modelled as a collection of modules; Service Manager module, Scheduler module, and Object Manager module. These modules emulate Service Manager, Scheduler, and Object Manager respectively in our simulation model. These three modules execute concurrently. They communicate each other through an explicit message.

We assume the server memory is big enough to hold all objects. This assumption is reasonable in the current computer technology (Perez-Sorrosal, et al., 2011). The server CPU speed is chosen to be 100,000 MIPS. The server CPU speed corresponds to the amount of processing power can be devoted by the server for its activities. We charge a requested service at the server by calculating the CPU time with following formula:

CPU time = (the number of instructions for the service)/(Server CPU speed)

4.3.4 Network

Instead of modelling in a particular type of network, we have modelled a network in a more abstract manner. Each network message has latency; processing cost for sending and receiving the message, network bandwidth, and propagation delay. CPU costs at each end consist of a fixed number of instructions and a variable number of instructions; these numbers shown in Table 1 are obtained from von Eicken, et al. (1995). We assume each client and servers is connected by a network channel which is facilitated by OMNET++. For sending a message between clients and servers, the time charged to each network channel is determined by processing time for sending and receiving plus network bandwidth multiplied by the message size plus propagation delay.

The propagation delay covers a delay due to the actual physical distance traversed by the message and a queuing delay caused by network congestion; we set the propagation delay as a number from the exponential distribution with mean 50ms (Pucha, et al., 2007).

We use a network bandwidth of 100 MBPS. We obtain this value from Wikipedia (http://en.wikipedia.org/wiki/Bandwidth_(computing)).

The propagation delay and communication transmission are neglected for sending a message from one module to another module within a single client or within servers. However, CPU costs and queuing delay are taken place.

4.4 The Workload Model

In this section, we describe our workload model adopted for our simulation experiments. We also describe how accesses for a transaction are generated. The workload models a realistic system with low contention. Each client has its locality region. We set 5 locality regions. Each locality region maps to 20% of databases. We assign a locality region randomly to each client. Any transaction from a client has accesses as follows: 80% objects from its locality region and 20% from the whole database.

Transactions are generated in the Application module. This module runs a single transaction one at a time. To generate a transaction, we do the following steps:

1. Set a unique transaction identifier.
2. Determine the transaction type.
3. Select the number of elements.
4. Select the number of objects to be accessed on each read element. If it is an update transaction, determine element type; read or write element. The last element of a transaction is a commit element.
5. Determine the objects to be accessed on each element.

Step 1 assigns a unique identifier to the transaction. The transaction identifier consists of a cache transaction identifier and a local identifier. The cache transaction identifier is issued by the server at the start of cache transaction; note that each client has a cache transaction running all the time at the server. Meanwhile, the local identifier is created by the local cache manager. Therefore, the Application module needs to send a request of a transaction identifier to the local cache manager before it generates a transaction.

In step 2, we determine the transaction type which is read-only transaction or update transaction. We determine the transaction type by using Bernoulli distribution with the probability of read-only transaction as a parameter.

To select the number of elements in step 3, we create a uniform random number from 2 to 6. A transaction at least has two elements; the first is read element and the second is commit element. Read element is an element containing read operations only. Meanwhile, commit element is an element containing write operations if it is an update transaction. Otherwise, if it is a read-only transaction, its commit element is an empty element. A transaction has only one commit element. It is the last element of a transaction.

In step 4, we determine the number of objects to be accessed for an element. We generate the number of object to be accessed by using the uniform random number from 1 to 5. If it is a read-only transaction, all its elements are read elements except the last element which is a commit element. If it is an update transaction, the element type of each element is determined at this step, except the last element. Each element of an update transaction has a probability equal to 0.5 to be a read or write element. If an update transaction does not have any write element, then its transaction type will be changed to a read-only transaction. Since there is no blind write, each write operation on object *x* will be converted to a read operation to object *x* and a write operation on object *x*. The write operation on object *x* will be added to the transaction's commit element.

The objects to be accessed by a transaction are determined in step 5. Objects to be accessed by a transaction are selected by using non-uniform access described in Tay, et al. (1985) which is called a b-c access; b is the percentage of a transaction's accesses and c is the percentage of database objects. The b-c access means b percent of a transaction's accesses fall within c percent of database objects. For example, under 80-20 access, 80 percent of a transaction's accesses fall within 20 percent of database objects.

To approach the b-c access, we divide database into five categories. Each category contains 20 percent of database objects. Each transaction selects one of five categories. Then 80 percent of a transaction's accesses are selected from its selected category and 20 percent of the transaction's accesses are selected from the whole database objects.

4.5 Simulation Results

This section presents the result of our simulation study that compares our scheme with other schemes in a client-server database system with caching at client-side. For the sake of simplicity, we assume a single server with multiple clients and an in-memory database at the server. Given a single server, there is no need for distributed two-phase commit. We believe that this simplification does not affect the relative performance comparison of VQ, Adya, and MVCC. Adding a distributed two-phase commit only adds a delay to commit times for all the algorithms. With the in-memory database object assumption, we eliminate the need for disk-latency simulation from our study.

Table 1 shows the parameter settings for generating transaction workload. The system maintains 1000 objects from the start to the end of the simulation time. We assume there are no delete and add object operations in the system. The size of each object is the same, which is 1 Mb for each object. The database is split into 5 regions. These database regions are used to model the locality reference pattern of client accesses. A client selects its locality region randomly; its subsequent accesses to its locality region are determined by the probability of locality reference parameter. If the probability of locality reference is 100%, a client accesses objects in its locality region only. If the probability of locality reference is 80%, each client accesses its locality region with 80% probability and the whole database (including its locality region) with 20% probability. *We refer 100% locality of reference as high locality of reference and 80% locality of reference as moderate locality of reference.*

Clients execute transactions continuously. Each transaction is a sequence of access requests as determined by the workload generator. Each request is either for read or write, determined randomly. If it is read access, all objects attached to the request are to be read; if it is write access, all objects attached to the request are to be updated. A write request of a transaction is executed at the transaction commit time. If a transaction aborts, a new transaction is started immediately. For each read request, some "thinking-time" is charged. This models the delay caused by the client before it proceeds to the next requests. *We define a workload with 80% of read only transactions as low to moderate contention workload.*

Table 2: Experiment Setting

Parameter	Setting
Number of requests for each transaction	Uniform(1,5)
The number of objects for each request	Uniform(1,5)
Observation time	100 hours (5th-105th hours)
Transaction inter-arrival time	Exponential(300 sec)
Thinking time	Exponential(150 sec)

Clients and server are connected by a 100 Mb per sec WAN. It has a propagation delay Exponential with mean 50 msecs for each message. A simulator run involves 5 repeats of 100 hour simulation.

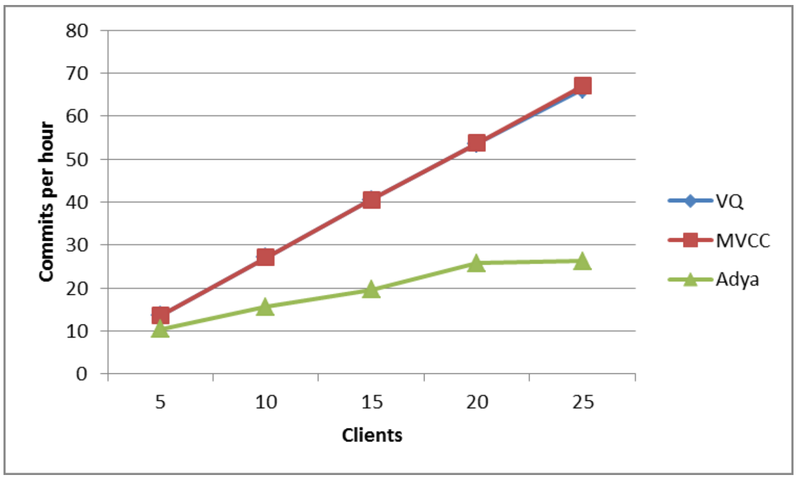
4.5.1 Number of Clients

The demands on the shared objects increase directly with more clients, and therefore, the effect on system performance of increasing clients is an important scalability issue. In the following set of experiments, we vary the number of clients in the system from 5 to 25 with 80% of read-only transaction, keeping all other parameters fixed at their default values. Figure 20 presents the results of these experiments, showing the effects on the system throughput, the message traffic, and the abort rate. The results show that VQ has scaled better that Adya in all experiments, and VQ has matched MVCC in the system throughput and the message traffic.

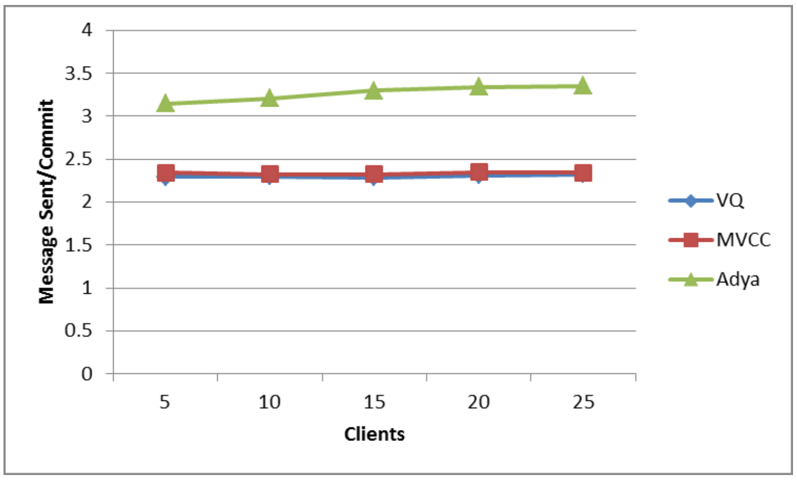
As can be seen from the first graph (Figure 20.a), the system throughput of all algorithms increases with larger number of clients. However, VQ and MVCC scale much better with increasing the number of clients. The reason is that Adya algorithm requires all transactions validate at the server. Therefore, Adya algorithm congestion problem at the server and consequently each transaction of Adya algorithm requires much more time to commit a transaction. As a result the system throughput (commit per hour) of Adya algorithm is less than others at all number of clients.

Figure 20.b shows the number of messages sent to the server per commit. A message is sent to the server by a local cache manager whenever it submits a commit of a transaction or a fetch request. In this graph, VQ has the number of message sent to the server less than Adya's, but it is equal to MVCC's. The reason is that Adya algorithm requires all transactions validating at the server. Meanwhile VQ and MVCC algorithm validates transactions at local cache manager except update transactions which have final validation at the server.

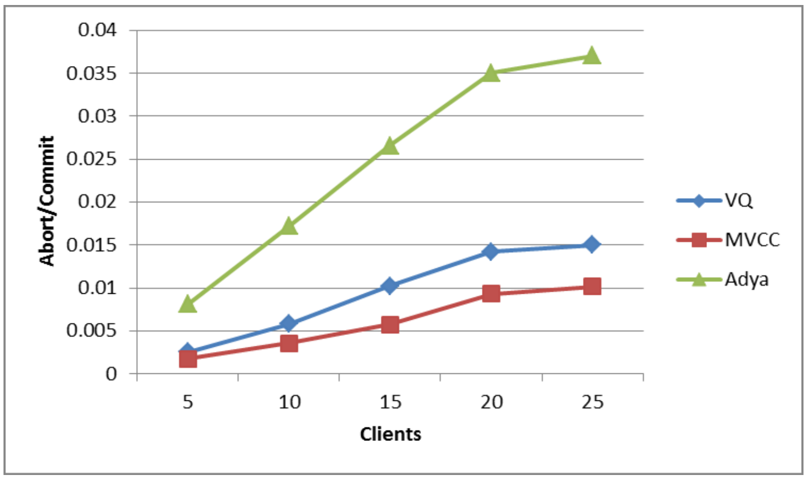
Adya algorithm orders transactions on timestamp basis. If two conflicting transactions are not ordered based on their timestamp, then one of them must be aborted even they are not interleaved each other. On the other hand, VQ is more direct to the problem than Adya. VQ ensures that no interleaving transaction is allowed to commit. If the execution of a transaction is interleaved with the execution of another transaction, then it is aborted. Therefore VQ aborts only necessary transactions. This is a reason why VQ has scaled better than Adya’s (see Figure 20.c).



(a)



(b)



(c)

Figure 20: (a) The System Throughput; (b) Message Traffic; (c) Abort Rate

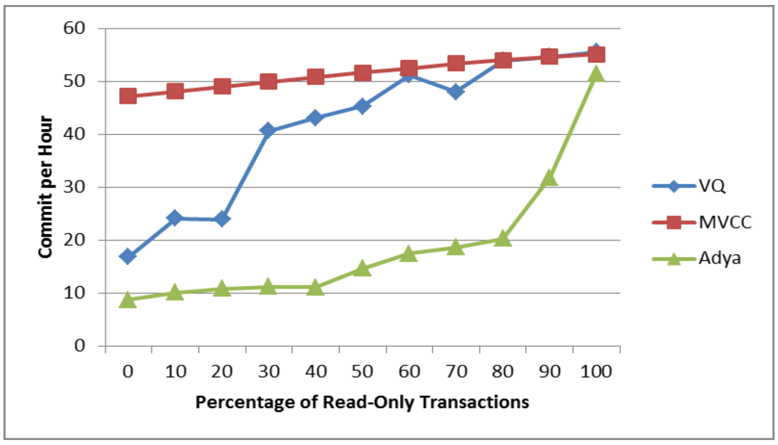
4.5.2 The Effect of Read-Only Transactions

To examine the effect of read-only transactions to the performance, we exercise the following set of experiments. We vary the percentage of read-only transactions from 0% to 100% with 20 clients and keeping all other parameters fixed at their default values.

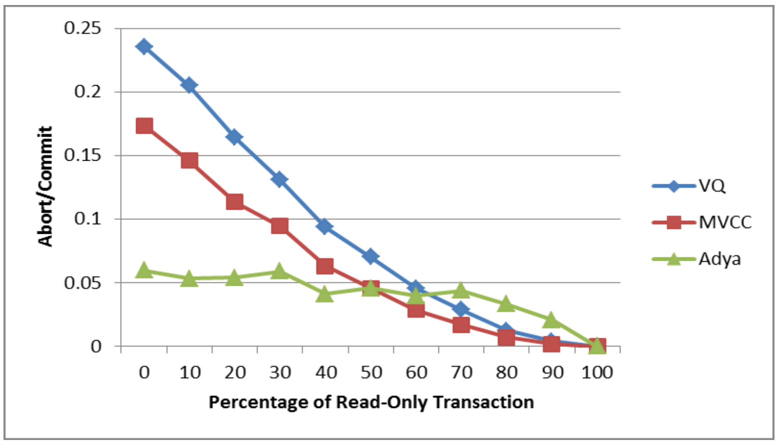
Figure 21 presents the results of these experiments, showing the effect of read-only transactions on the system throughput, the abort rate, and the message traffic. The results show that VQ outperforms Adya on all performances for the percentage of read-only transactions greater than 60%. Again the performance of VQ matches to the performance of MVCC on all performances for the percentage of read-only transactions greater than 60%.

Figure 21.a shows that VQ outperforms Adya in system throughput for all percentages of read-only transactions, and the performance of VQ matches to the performance of MVCC for the percentage of read-only transactions greater than 60%. Again the reason is that Adya requires transactions to validate at the server. On the other hand, VQ validates transactions at the client side and shares the validation process of update transactions between the client and the server. Meanwhile MVCC does not require read accesses to get read lock; only write accesses are required to get write lock at the server. This reason is also used to justify the following results.

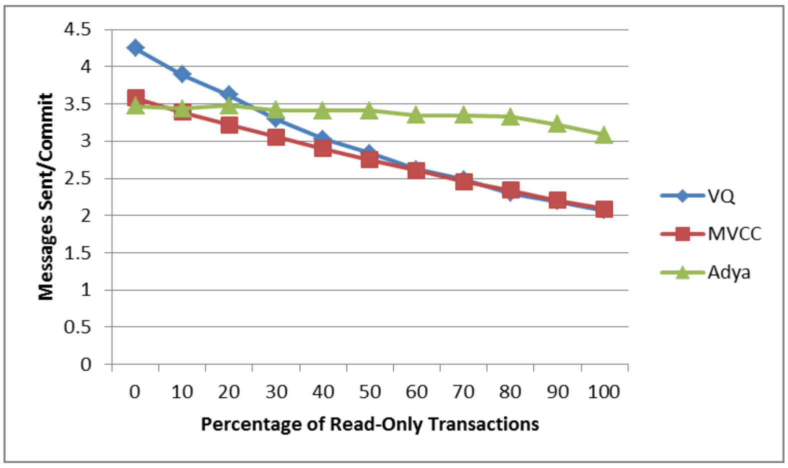
For the percentage of read-only transactions greater than 60%, VQ has better abort rate than Adya (see Figure 21.b). VQ's performance matches MVCC's for percentage of read-only transactions greater than 80%. VQ outperforms Adya in the number of messages sent to the server for the percentage of read-only transactions greater than 30%, and VQ matches MVCC for the percentage of read-only transactions greater than 60% (see Figure 21.c).



(a)



(b)



(c)

Figure 21: The Effect of Read-Only Transactions: (a) The System Throughput; (b) Abort Rate; (c) Message Traffic

Chapter 5. Extensions to the Proposed Scheme

This chapter is designated to describe a few extensions to the proposed scheme. The scheme described in chapter 3 is the basics of the proposed scheme. Now we present the proposed scheme with some extensions. The first extension we described is disconnected operation. It is described in section 5.1. Disconnect operation is a mode of operation in which a client uses cached objects to work while disconnected with the server. This ability is very useful for mobile clients even when connectivity is available. For example, it can extend battery life by avoiding wireless transmission and reception. It can reduce network expense and it allows radio silence to be maintained, a vital capability in military operations.

In section 5.2 we present an extension of our scheme to multiple database systems requiring multiple servers.

In the section 5.3, we describe two extensions; those are concurrent transactions and the extension to support server-side MushUps and edge-server configuration. In modern computation, client applications are to be complex and they may require running multi transactions at a time. The client application of our scheme so far runs a single transaction at a time. Subsection 5.3.1 describes the extension of our scheme to concurrent transactions. Meanwhile in subsection 5.3.2, we describe the extensions of server-side MushUps and edge-server configuration.

5.1 Disconnected Operation

Disconnected operation refers to the ability of a client to continue working on local cached objects in spite of disconnections. Disconnected operation is very useful feature for mobile clients. Mobile clients may have an intermittent or low bandwidth connection to the server. To enhance the performance, clients may disconnect to the server and work offline. There are other reasons for clients to disconnect their connection network. For examples, clients may disconnect to the server for saving the battery life, for reducing network charges, or for maintaining radio silence in military operations (Jin, 1999).

To provide disconnected operation, our scheme requires some modifications to its design. In this section we describe the modifications of our scheme in order to make our scheme allowing clients to run disconnected operation.

In disconnected environment, the positive response of EOT (End of Transactions) is locally validated (LOCALCOMMIT_ACK). Any read-only transaction also receives a locally validated reply, unless it does not read any dirty objects. A read-only transaction may read dirty objects from locally validated update transactions.

When a disconnected client submits a connect request, all the locally validated update transactions are submitted to the server. Some of these transactions may receive positive response (COMMIT_ACK), others may receive abort response (ABORT_REQ). An aborted transaction may cause a cascading abort for other transactions.

5.1.1 Cache Transaction Model

There are some additional properties of a cache transaction. Those are listed as follows:

- isConnected: flag of connection to network; true for connected, false for disconnected.
- List of update propagation elements: a pointer to linked lists of delayed update propagations.

In the basic scheme of our algorithm, it is required that an update transaction waits for the response of its final validation process. This scheme is most suited when the probability of conflict is high. However, if the probability of conflict is low or the aborts are rare in the system, then the performance can be improved by relaxing this requirement. We design two strategies for the commit process of update transactions. These are:

- *Synchronous commit:* an update transaction has to wait for the response of its final validation process from the server.
- *Asynchronous commit:* an update transaction does not have to wait for the response of its final validation process from the server. In this strategy, the update transaction can leave the system or the client application can create a

new transaction after it passes local validation. We assume that an update transaction succeeds its final validation process with high probability. However, if it fails the final validation process, then it may cause cascading aborts.

Since read-only transactions do not need the final validation process, the above strategies are only for update transactions. However, Asynchronous commit strategy can cause cascading aborts of read-only transactions. When an update transaction has been locally validated, its updates are available for local transaction. The abort of the update transaction at final validation causes the abortion of transactions read objects from the update transaction.

A client which runs under disconnected mode executes the commit request of a transaction with asynchronous commit strategy.

5.1.2 The execution of Cache Manager

The cache manager may receive additional requests from the client and the server.

- Disconnect request: a request from the client application to switch from running under connected mode to disconnected mode.
- Connect request: a request from client application to switch from running under disconnected mode to connected mode.
- Refresh request: a request from client to refresh the cached objects. It is available while running under disconnected environment.

Disconnect request. Whenever the local cache manager receives a disconnect request from the client, it forwards the request message to the server. Upon receiving the disconnect request message, the server checks whether any transactions from the requesting cache side have yet to be validated at the server. The server will send a disconnect acknowledgement message to the local cache manager after all validations are complete; and the server marks the associated cache transaction as running under disconnected mode. From now on the update propagation requests for this cache side are held at the server. They are ordered based on first in first out order on a list attached to the associated cache transaction.

When the local cache manager receives a disconnect acknowledgement, it forwards the message to the client and marks itself that from now on the cache side runs under disconnected mode. Two things should be noted while running in disconnected mode:

- the client runs on asynchronous commit strategy,
- the final validation of commit transactions is delayed until the client runs under connected mode. Commit update transactions are set as locally validated. Read-only transactions are set as validated if they do not read dirty objects (or uncommitted objects); otherwise they are set as locally validated.

Connect request. A connect operation is executed by the local cache manager when the client wishes to break the disconnected mode. The purpose of connect operation is to switch the environment from disconnected to connected. However, before it switches to the connected environment, the system should execute a reconciling process. Reconciling process is to execute all operations which were postponed under disconnected environment. The reconciling process is carried in two phases:

- The first phase: This phase is started when the server receives a connect request from the client. In this phase, the server sends all update propagations which were held at the server side to the cache manager. After sending all update propagations, the server sends a connect acknowledgement message to the cache manager. When the cache manager receives an update propagation, it checks the update propagation against the locally validated elements of update transactions in CVQ. Any conflicting transaction is aborted, and abort message sent to the client. Any transaction reads from the aborted transaction will be aborted as well.
- The second phase: This phase is started whenever the local cache manager receives a connect acknowledgement. In this phase the local cache manager examines CVQ whether any locally validated update transaction can be submitted to the server; if any, it submits the commit request of the update transactions to the server. Eventually, it marks itself from now on as running under connected mode.

Refresh request. Refresh operation is provided while running under disconnected mode. Refresh operation is exactly the same as the connect operation except the client keeps running under disconnected mode after the reconciling process. Initially, the cache manager executes the connect operation. Afterwards, it executes disconnect operation. Note that all delayed update propagations for the client can be sent to the cache manager at this time, but not all locally validated update transactions at the client side can be submitted in the refresh operation. Because the cache manager cannot send a commit request of a transaction which conflicts with the current committing transactions. Then, the cache manager sends all commit requests of locally validated update transactions that can be submitted before it submits a disconnect request to the server.

5.1.3 The Execution of Scheduler

Beside the regular requests described in chapter 3, the scheduler may receive the following additional requests from the clients in order to provide disconnected operation:

- Disconnect requests; requests to switch from connected to disconnected environment,
- Connect requests; requests to switch from disconnected to connected environment.

Disconnect request. When the scheduler receives a disconnect request from a local cache manager, it examines whether any transaction from the local cache manager is in the middle of its commit process at the server; If any, the scheduler delays the disconnect process until the commit process of the transactions is finished. Otherwise, it sends a disconnect acknowledgement message to the local cache manager.

```
Receive a connect request message;
Get the associated cache transaction;
Get an update propagation from the cache transaction;
While  (an update propagation is not null)
{
    Send the update propagation to the client;
    Get another update propagation from the cache transaction;
}
Send a connect acknowledgement to the client;
```

Figure 22: Processing Connect Request at Server Side

Connect request. Figure 22 shows the pseudo-code of the connect operation process at the server side. As mentioned in the previous subsection that a connect request sends by the client if it wants to finish its running under disconnected mode. Therefore, the connect process at the server side is to send all delayed update propagations to the local cache manager; the delayed update propagations are sent in first in first out basis. Then, it sends a connect acknowledgement to the local cache manager.

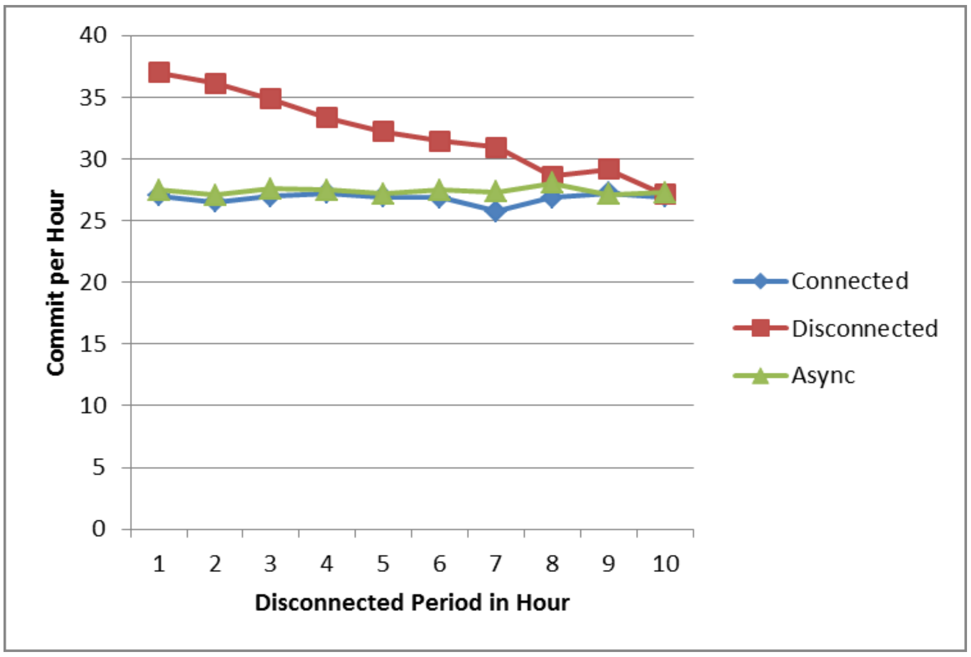
5.1.4 Performance

To evaluate the performance of the proposed scheme under disconnected mode, we set the following experiments. There are two experiments; the first experiment is to evaluate the performance of VQ under three modes; disconnected mode (Disconnected), connected mode with synchronous commit options (Connected), and connected mode with asynchronous commit options (Async). For this experiment, we set 10 connected clients and 10 disconnected clients. We run the first experiment by the variation of disconnected period; from 1 hour to 10 hour disconnected period with 80% of read-only transactions. The second experiment is to evaluate the effect of read-only transaction percentage to the performance of VQ under those three modes. We vary the percentage of read-only transactions from 0% to 100% with 10 hour disconnected period.

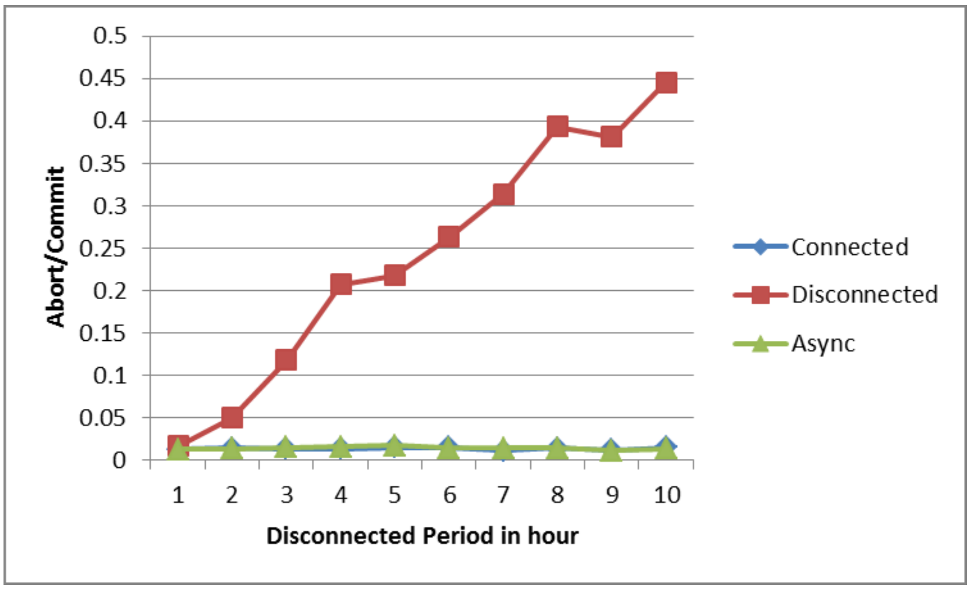
VQ under disconnected mode outperforms VQ under connected mode in both commit options; synchronous and asynchronous commit options, for all disconnected period; from 1 hour to 10 hours, (see Figure 23.a). The system throughput of disconnected clients is significantly higher than the system throughput of the connected clients; with synchronous and asynchronous commit options, at disconnected period 1. The rationale behind this is that disconnected clients are not interfered by other clients and read-only transactions are allowed reading stale objects. Meanwhile the connected clients are interfered by other clients and the cached objects are always refreshed immediately after persistent objects updated at the server. However, the advantages of the disconnected clients over connected clients decrease as disconnected period increases. The reason is that the disconnected clients introduce cascading aborts. Figure 23.b shows the abort rate of disconnected clients and connected clients. The abort rate of disconnected clients increases as disconnected period increases. Meanwhile, the abort rate of connected clients is not affected by disconnected periods.

Figure 24.a and Figure 24.b show the effect of read-only transaction percentages under disconnected environment (10 hour disconnect period). Our simulation results show that our scheme perform reasonable well under disconnected environment (10 hour disconnected period) with mostly read load (read-only transaction percentage 80-100%).

In Figure 23 and Figure 24, we can see that the performance of the asynchronous commit options (Async) and the synchronous commit option (Connected) is nearly the same. This fact indicates that the advantages of our scheme; read-only transactions can be validated at cache-side, boost the performance of the synchronous commit option. Therefore, the difference between the asynchronous commit and synchronous commit performance is not significant.

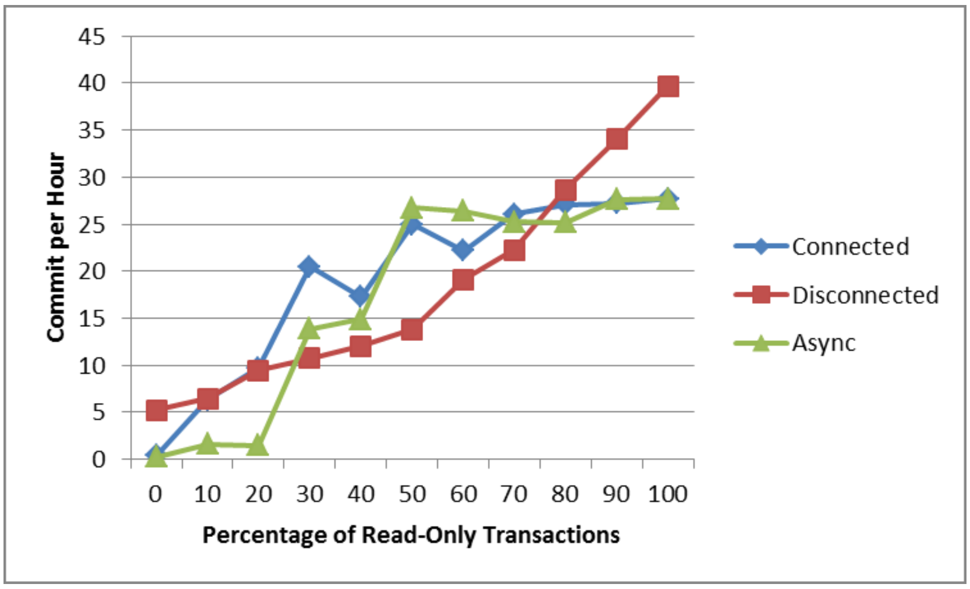


(a)

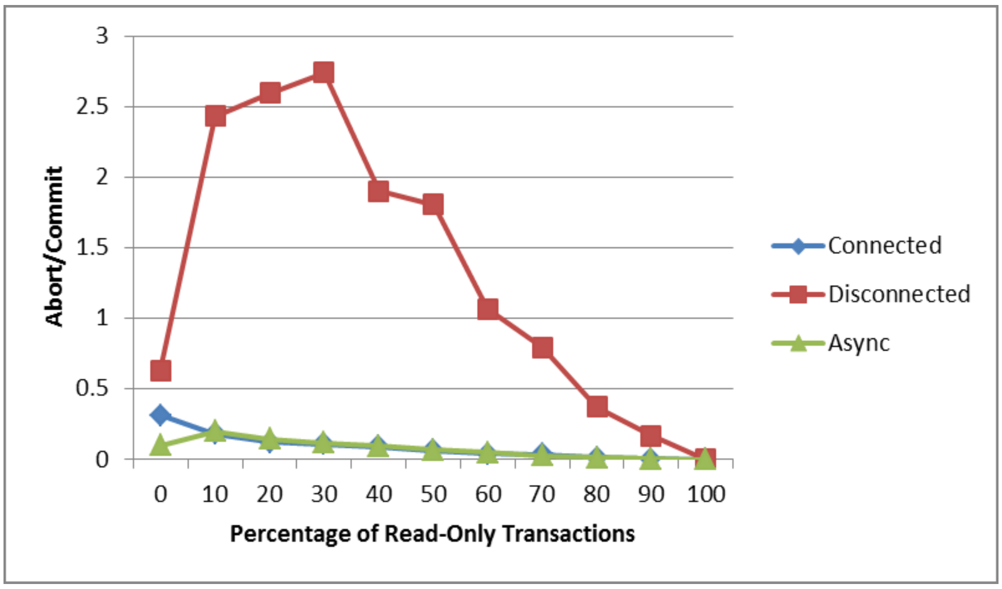


(b)

Figure 23: The performance of Connected and Disconnected Clients: (a) The System Throughput; (b) The Abort Rate.



(a)



(b)

Figure 24: The Performance of Connected and Disconnected Clients: (a) The System Throughput; (b) The Abort Rate.

5.2 Multiple Server System

Our scheme so far assumes a single server system. The use of a single-server greatly simplifies the validation processes of update transactions at the server. Here, we generalize our scheme to multiple server system. Each server maintains its own database. We describe now the necessary extension to our original scheme. Note that our scheme does not support nested transactions.

Figure 25 shows the configuration of the client-server database systems with multiple servers. For each client, one of the servers (typically the closest) is chosen as a *host server* and the rest servers are as *participant servers*. To cache objects, the local cache manager at a client submits a fetch request to its host server. To respond to a fetch request, the host server may forward the request to other participant servers; especially the servers that store the requested objects. The server that stores an object, we call it as the *owner* of the object. Thus, we assume that each server has knowledge about database stored at other servers. The cache manager initially submits a Cache Start request to the host. To response to the request, the host executes a new cache transaction associate with the cache side.

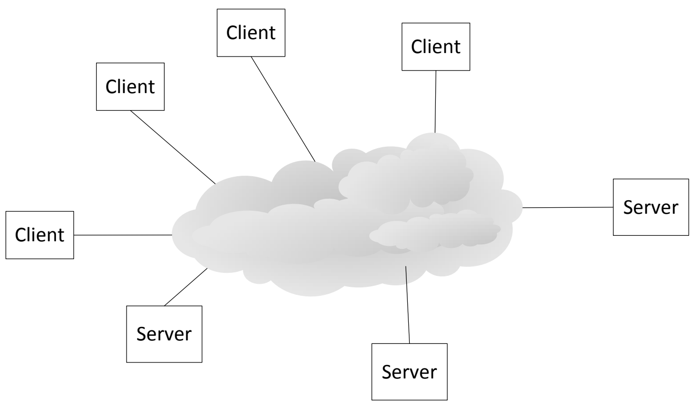


Figure 25: The Configuration of the Multiple Server Systems

Upon receiving the fetch request forwarded by the host, a participant server may create a new sub cache transaction. Thus, a cache manager may associate one cache transaction at the host server. A cache transaction in turn may have sub cache transactions at participant servers. If any persistent objects are updated at the owner, then the owner sends an update propagation request to the host and then the host forwards the request to the cache manager.

Each cache transaction has a unique identifier; we suggest using the host server identifier and client identifier (or cache identifier) to form the cache transaction identifier. The sub cache transaction identifier may consist of the host server identifier, the cache identifier, and the participant server identifier. The cache manager is only aware the cache transaction that runs at the host server.

As before, each cache transaction has a *sequence number* associate with it. This sequence number is maintained by the server where the cache transaction is executed. Whenever a local cache manager submits a request to the server, it should provide the request with the sequence number of the cache transaction. If the cache transaction sequence number associated with the request does not match the cache transaction sequence number stored at the server, then the server rejects the request and sends a *verification* request to the local cache manager.

To commit an update transaction, we use the standard *two-phase commit* protocol (Bernstein et al, 1987). A cache manager submits a commit request of the update transaction to its host server. Upon receiving the request, the host server may forward the request to other participants that own the objects accessed by the update transaction. Whenever a participant server receives any commit request from the host server of the transaction that acts as coordinator of a commit protocol, it creates the corresponding element, inserts the element into SVQ, and validates the transaction. The result of the validation is sent to the coordinator. Then the participant server waits for *commit trigger* or abort message from the coordinator.

If all validation results of each participant servers are positive and the validation process at the host server (or coordinator) is also positive, then the coordinator decides commit and sends commit trigger message to all participant servers. Otherwise, the coordinator decides abort and sends abort message to all participant servers that reply positive; those that reply negative already decided abort.

5.3 Other Extensions

5.3.1 Concurrent Transactions from a Client

Our scheme described in chapter 3 assumes that any client application executes transactions one at a time. If a client application is multithreaded, it has to coordinate its thread as part of a single transaction. Users may prefer to execute multiple transactions in parallel. In the basic design of our scheme, they have to start multiple cache sides for the same application. However, this is expensive since it leads to excessive duplication of data. It is more desirable to have a scheme in which a single cache side allows an application to execute multiple transactions simultaneously.

This is easily performed in our scheme. The local cache manager has to be modified so that it can serve more than one transaction. For the correctness of the execution of transactions, it is required that a single local cache manager submits the commit of conflicting update transactions to the server one at a time.

5.3.2 Supporting Server-Side MushUps and Edge Server Configuration

In many web applications it is necessary to fuse two or more resources from other web applications or tailor their own resources with one or more resources from other web applications. These web applications are called server-side MushUps (Auinger, et al. , 2009; Palfrey and Gasser, 2007).

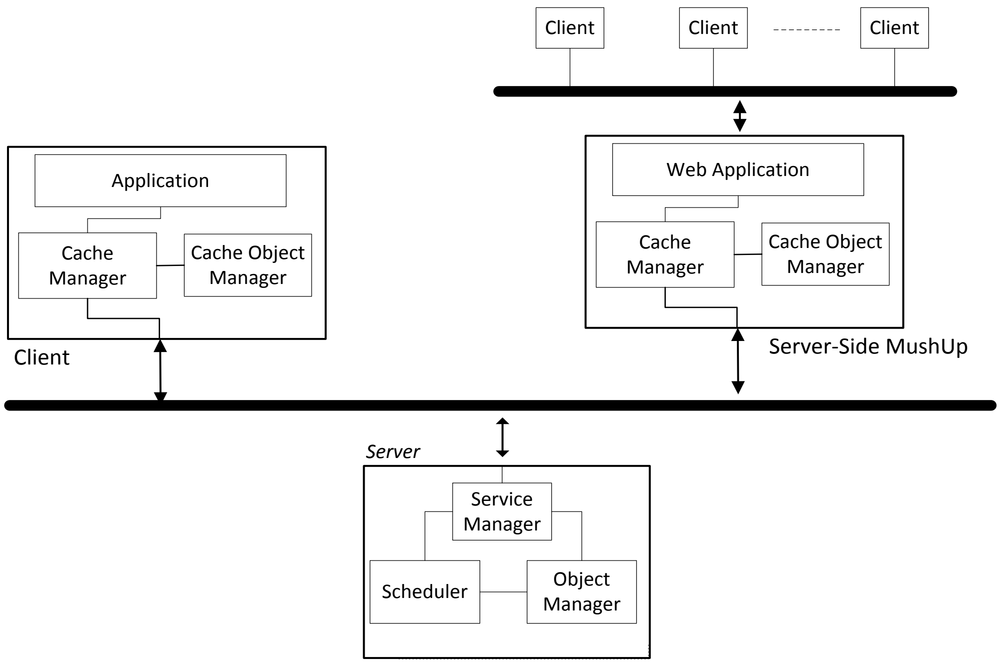


Figure 26: The Configuration of Server-Side MushUp

Figure 26 shows the configuration of server-side MushUps in client-server database systems. From the server's point of view, a server-side MushUp is just another client. It consists of our cache manager and our cache object manager. The clients of the server-side MushUp web applications can read and write our objects by creating transactions through the cache manager. Then the cache manager communicates to the server for providing accesses to the clients of the server-side MushUp web applications.

Our scheme can also support edge-server configuration which is very similar to server-side MushUp. The edge server is treated as a client. It can cache some objects.

The same as the server-side MushUp, the edge server may serve some clients (see Figure 27).

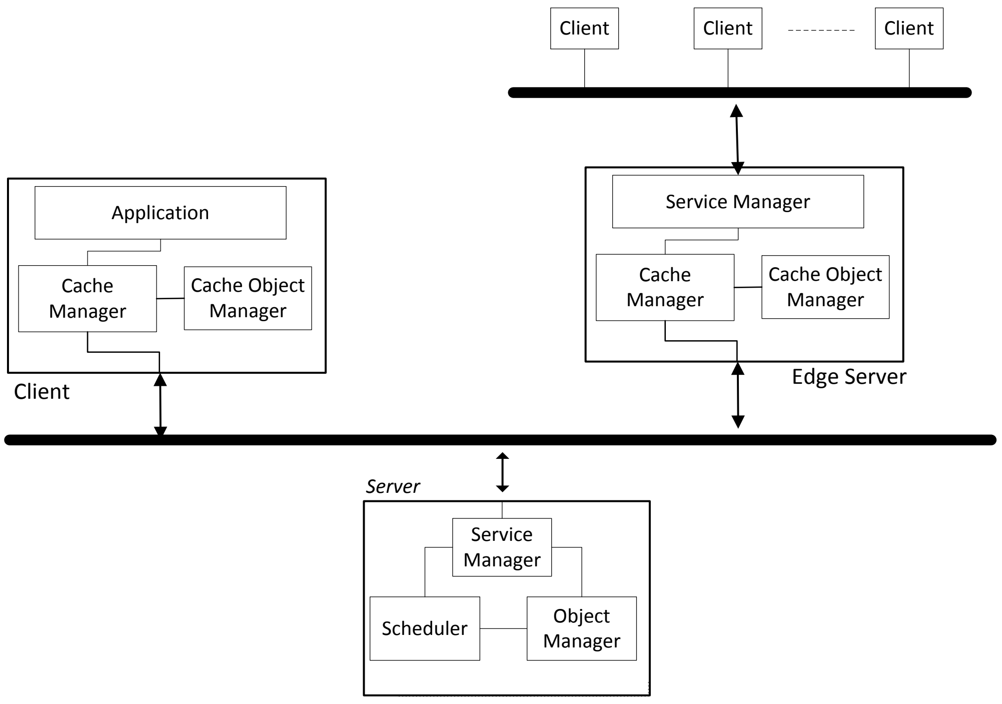


Figure 27: The Configuration of Edge Server

Chapter 6. Conclusions

In this thesis we have presented a new concurrency control algorithm for client-server database system with caching at the client side. We have demonstrated that the proposed algorithm outperforms the currently best known algorithm (Adya algorithm) through our simulation study. We have also proved the correctness of the proposed algorithm. Furthermore, we have presented some extensions of the proposed algorithm to permit disconnected operations and multiple servers. In this chapter we summarize our work and also present interesting problems for future research.

6.1 Summary of Thesis Contributions

This thesis has presented a transactional cache consistency scheme for client-server database systems with caching at the client side. It is based on the optimistic approach to concurrency control. We choose the optimistic approach over the pessimistic approach because we assumed low data contention environment (predominantly read-only load), precisely where caching would be most effective.

The proposed scheme uses a validation queue to record and to order accesses. This is the reason that we name our scheme as VQ; it stands for Validation Queue. We put a validation queue at each client side; we name it as Cache Validation Queue (CVQ), to record and to order accesses at client side. At the server side, we put another validation queue; we name it as Server Validation Queue (SVQ), to record and to order accesses at the server side. The proposed scheme employs these queues to validate transactions.

Our scheme consists of two validation algorithms; the validation at cache side and the validation at server side. The validation at client side is to check the client accesses against the updates of other clients sent by the server to the client. Meanwhile the validation at server side checks the client accesses against the accesses of other clients at the server. Therefore, the validation process of read-only transactions can be carried out at client side without communicating with the server and the validation process of update transactions is in two stages. The first stage is at client side. The second stage is at server side. Consequently, incorrectness of transaction execution can be detected earlier at client side. Incorrect transactions detected at client side will be aborted. Therefore, transactions submitted to server are more likely to succeed the validation process. Consequently, the number of abortions at server side is low. Meanwhile, other schemes, CBL and Adya validate transactions at server side only. All transactions including all incorrect transactions are submitted to servers. This causes the number of abortions at the server to be higher. Since MVCC does not validate read-only transactions and validates update transactions at server side, we consider the number of abortions at server for MVCC to be medium.

Our simulation work shows that the proposed scheme outperforms the scheme considered best (Adya) and compares favourably with the MVCC scheme that only provides shot isolation. We have evaluated the studying scheme in some parameters; the system throughput, the number of aborted transactions per committed transactions (the abort rate),

We have also proved the correctness of the proposed scheme. We prove that the proposed algorithm provides committed transactions serializability.

Some additional features of our proposed scheme have been described in this thesis. Those additional features are as follows:

- **Disconnected Operation:** This feature enables clients to continue working on local cached objects in spite of disconnections. It is very useful for mobile clients in which may have an intermittent or low bandwidth connection.
- **Multi-Server System:** This additional feature enables our system to have more than one server. Persistent objects are distributed over several servers and a transaction can cache objects from several servers.
- **Other Features:** With the extensions described at section 5.3, our scheme allows clients to issue more than one transaction at a time. Another additional feature of our proposed scheme is to support server-side MushUp applications and edge-server configurations.

Table 3 shows the features of current transactional cache consistency schemes; Callback locking (CBL), Adya algorithm, Multiversion Concurrency Control algorithm (MVCC), and our scheme VQ. Our scheme; VQ, has the best features compare to the rest. Since our scheme has a cache manager at each cache side and it acts as a server to transactions, then our scheme allows a client to execute transactions simultaneously.

Furthermore, it supports disconnected operations, server-side MushUp applications, and edge-server configuration.

Table 3: The Features of Current Transactional Cache Consistency Schemes

Features	CBL	Adya	MVCC	VQ
Technique used	Locking	Timestamp	Locking	Non Blocking
Deadlock-free	No	Yes	No	Yes
Degree of Consistency	Serial	Serial	Snapshot Isolation	Serial
Validation of Read-Only Transactions	No Validation	Server Side	No Validation	Client Side
Validation of Update Transactions	Server	Server	Server	Share
The number of abortions at the server	High	High	Medium	Low

6.2 Future Work

We suggest two areas of additional work. The first area is to provide the ability for a group of clients to share their caches in a distributed manner to form a single cache. For example, if each client in the group has 1G cache, then a group of 10 clients will form a 10G cache. Memcached (http://www.linuxjournal.com/article/7451) is a good example of such scheme. Extending this scheme to work with our algorithm will be a very useful extension.

Figure 28 shows the expected client-side architecture. The architecture is similar to the current scheme, except there is a new component named *Space Manager*. Space Manager is a module to manage memory for the client cache. The memory managed by the Space Manager is distributed physically among clients. It can be considered as a memcached server. It serves Cache Object Manager for read or write operations. It may serve local cache object manager or remote cache object manager from other clients in the group.

A single cache object manager can be thought as a Memchaced client. By using memcached library, such as "memcached_get" and "memcached_set", the client can get and update the object value respectively wherever the object is stored in the distributed cache. Since there would be more than one client in a group, then the Memcached concepts should be converted to distributed Memchached concepts.

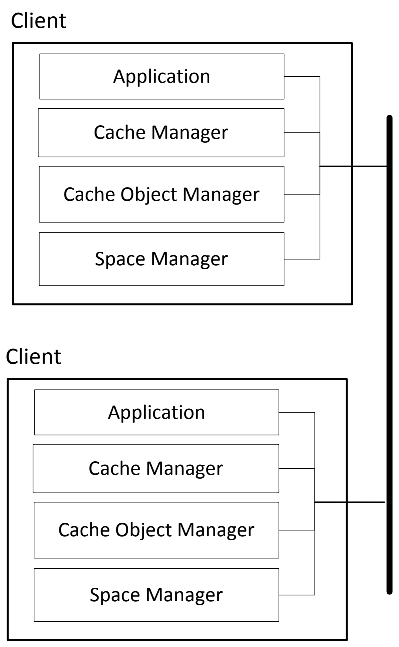


Figure 28: Distributed Caching at Client Side

The second area of work is to investigate how replication can be incorporated in our system. There are two reasons to employ database replication to improve performance and to increase availability. Although middleware-based replication scheme that transparently replicate data have been studied extensively in the literature, practical workable solutions are still not available (Cecchet et al, 2008). The main reason being the scheme for consistency, availability, and performance can interact in subtle ways so performing tradeoffs is difficult and requires much experimental work and tuning (Cecchet et al, 2008).

Finally we suggest an interesting generalization of our scheme consisting of integrating our VQ clients to servers which are employing other concurrency control schemes such as two-phase locking or timestamp concurrency control. Such an integrating is possible if at the server side enhancement to include cache update propagation functionality can be incorporated.

References

Adya, A., Gruber, R., Liskov, B. and Maheshwari, U. (1995) ‘Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks’, the ACM SIGMOD Conference on Management of Data.

Ahuja, R.K. and Orlin, J.B. (1992) *Use of Representative Operation Counts in Computational Testings of Algorithms*, Unpublished paper, Massachussetts Institute of Technology.

Alonso, R., Barbara, D. and Garcia-Molina, H. (1990) ‘Data Caching Issues in an Information Retrieval System’, ACM Transactions on Database Systems, Vol. 15, No. 3, pp. 359-384.

Auinger, A., Ebner, M., Nedbal, D., and Holzinger, A. (2009) ‘Mixing Content and Endless Collabolration – MashUps: Towards Future Personal Learning Environments’ In Stephanidis, C. (ed.) *Universal Access in Human-Computer Interaction. Applications and Services*, Lecture Note in Computer Science, Springer Berlin / Heidelberg, vol. 5616, pp. 14-23.

Bayer, R., Heller, H. and Reiser, A. (1980) ‘Parallelism and Recovery in Database Systems’, ACM Transactions on Database Systems, Vol. 5, No. 2, pp. 139-156.

Bhargava, B. (1999) ‘Concurrency Control in Database Systems’, IEEE Transactions on Knowledge and Data Engineering, VOL. 11, NO. 1.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O’Neil, E., and O’Neil, P. (1995) ‘A Critique of ANSI SQL Isolation Levels’, in *SIGMOD’95*: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 1-10.

Bernstein, P.A. and Goodman, N. (1981) ‘Concurrency Control in Distributed Database Systems’, *Computing Surveys*, Vol. 13, No. 2

Bernstein, P.A. and Goodman, N. (1983) ‘Multiversion Concurrency Control – Theory and Algorithms’, *ACM Transactions on Database Systems*, Vol. 8, No. 4, pp. 465-483.

Bernstein, P. A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R. and Tamma, P. (2006) 'Relaxed Currency Serializability fo Middle-Tier Caching and Replication', Proc. ACM SIGMOD, Chicago, IL., pp. 599-610.

Bober, P.M. and Carey, M.J. (1991) 'On Mixing Quries and Transactions via Multiversion Locking', Computer Science Department, University of Wisconsin, Madison, WI.

Bukhari, F. and Shrivastava, S. (2012)'An Efficient Distributed Concurrency Control Scheme for Transactional Systems with Client-Side Caching', in *Proceeding of 2012 IEEE 14th International Conference on High Performance Computing and Communications*, Liverpool, UK, pp. 1074 - 1081.

Bukhari, F. (1990) 'Two Fully Distributed Concurrency Control Algorithms', Master of Science Thesis, Department of Computer Science, the University of Western Ontario.

Bukhari, F. and Osborn, S. (1997) 'Two Fully Distributed Concurrency Control Algorithms', IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 5.

Cahill, M.J. (2009) *Serializable Isolation for Snapshot Databases*, PhD Thesis, The School of Information Technologies, the University of Sydney.

Carey, M. and Muhanna, W.A. (1986) 'The Performance of Multiversion Concurrency Control Algorithms', ACM Transactions on Computer Systems, Vol. 4, No. 4, pp. 338-378.

Carey, M., Franklin, M. and Zaharioudakis, M. (1994) 'Fine-Grained Sharing in a Page Server OODBMS', Proc. ACM SIGMOD , pp. 359-370.

Castro, M., Adya, A., Liskov, B., and Myers, A. (1997) 'HAC: Hybrid Adaptive Caching for Distributed Storage Systems', In *Proc. of ACM SIGMOD International Conference on Management of Data*, Washington D.C., pp. 102-115.

Cecchet, E., Candea, G., and Ailamaki, A. (2008) 'Middleware-based Database Replication: The Gaps Between Theory and Practice', In *Proc. of ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada

Delis, A. and Roussopoulos, N. (1991) 'Performance and Scalability of Client-Server Database Architectures', Proceedings of the 18th VLDB conference Vancouver, BC, Canada.

Eswaran, K., Gray, J., Lorie, R. and Traiger, I. (1976) 'The Notions of Consistency and Predicate Locks in a Database System', *Communications of the ACM*, 19(11), pp. 624-633.

Franaszek, P. and Robinson, J.T. (1985) ' Limitations of Concurrency in Transaction Processing', ACM Trans. Database Syst. 10, 1 , pp. l-28.

Franklin, M.J. and Carey, M.J. (1992) 'Client-Server Caching Revisited', In Poceedings International Workshop on Distributed Object Management, Edmonton, Canada, pp. 57-78.

Franklin, M.J. (1996) *Client Data Caching*, Kluwer Academic Publishers, Dordrecht.

Franklin, M.J., Carey, M.J. and Livny, M. (1997) 'Transactional Client-Server Cache Consistency: Alternatives and Performance', ACM Transactions on Database Systems, 22 (3), pp. 315-363.

Gray, J., Lorie, R., Putzolu, G. and Traiger, I. (1976) 'Granularity of Locks and Degrees of Consistency in a Shared Database', In Modeling in Data Base Management Systems. Amsterdam: Elsevier North-Holland.

Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA.

Gruber, R. (1997) *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases.* PhD Thesis, Massachussetts Institute of Technology.

Gruber, R., Kaashoek, F., Liskov, B. and Shrira, L. (1994) 'Disconnected Operation in the Thor Object-Oriented Database System', In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications', Santa Cruz, CA.

Guo, H., Larson, P., Ramakrishnan, R. and Goldstein, J. (2004) 'Relaxed Currency and Consistency: How To Say "Good Enough" in SQL', ACM SIGMOD 2004, Paris, France.

Herlihy, M. (1990) 'Apologizing Versus Asking Permission: Optimistic Concurency Control for Abstract Data Types', ACM Transactions on Database Systems, Vol. 15, No. 1, pp 96-124

Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J. (1988) 'Scale and Performance in a Distributed File System', ACM Transactions cm Computer Systems, Vol. 6, No. 1, pp. 51-81.

Jing, J., Helal, A. and Elmagarmid, A. (1999) 'Client-Server Computing in Mobile Environment', ACM Computing Surveys, Vol. 31, No. 2.

Kim, W. (1990) 'Object-oriented databases: definition and research directions', *Knowledge and Data Engineering, IEEE Transactions on*, 2(3), pp. 327-341.

Kistler, J.J. and Satyanarayanan, M. (1992) 'Disconnected Operation in the Coda File System', ACM Transaction on Computer Systems, Vol 10, No. 1.

Kistler, J.J. (1993) *Disconnected Operation in a DistributedFile System*, PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Kleijnen,J.P.C. (1995) 'Verification and Validation of Simulation Models.Theory and Methodology', *Eurpean Journal of Operation Research, vol. 82, pp. 145-162.*

Kossman, D., Kraska, T., and Loesing, S. (2010) 'An Evaluation of Alternative Architectures for Transaction Processing in the Cloud', ACM *SIGMOD'10,* Indianapolis, Indiana, USA

Kung, H. T. and Robinson, J. T. (1981) 'On Optimistic Methods for Concurrency Control', *ACM Transactions on Database Systems*, 6(2), pp. 213-226.

Kuo, T., Kao, Y. and Kuo, C. (2002) 'Two-Version Based Concurency Control and Recovery in Real-Time Client/Server Databases', IEEE Transactions on Computers, Vol. 52, No. 4.

Larson, P., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., and Zwilling, M. (2011) 'High-Performance Concurrency Control Mechanisms for Main-Memory Databases', In *Proceedings of the VLDB Endowment*, vol. 5, no. 4.

Leff, A. and Rayfield, J.T. (2004) 'Alternative Edge-Server Architectures for Enterprise JavaBeans Applications' Lecture Notes in Computer Science 3231, pp. 195-211.

Lewandowski, S.M. (1998) 'Frameworks for Component-Based Client/Server Computing', ACM Computing Surveys, Vol. 30, No. 1.

Mummert, L.B. (1996) *Exploiting Weak Connectivity in a Distributed File System,* PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

OMNET++ (2010) User Manual Ver 4.1, András Varga and OpenSim Ltd.

Özsu, M.T. and Valduriez, P. (1999) *Priciples of Distributed Database Systems*. Prentice Hall

Palfrey, J. and Gasser, U. (2007) 'Mashups Interoperability and eInnovation. Case Study', Berkman Publication Series, Harvard University Research Center for Information Law and University of St. Gallen, St. Gallen

Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R. and Kemme, B. (2011) 'Elastic SI-Cache: consistent and scalable caching in multi-tier architectures', The VLDB Journal, DOI 10.1007/s00778-011-0228-8.

Pitoura, E. and Bhargava, B. (1999) 'Data Consistency in Intermittenly Connected Distributed Systems', IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 6.

Podlipnig, S. and Böszörmenyi, L. (2003) 'A Survey of Web Cache Replacement Strategies', ACM Computing Surveys, Vol. 35, No. 4, pp. 374-398.

Pucha, H., Zhang, Y., Maou, Z.M., and Hu, Y.C. (2007) 'Understanding Network Delay Changes Caused by Routing Events', in *SIGMETRICS'07*, San Diago.

Satyanarayanan, M., Kistler, J., Mummert, L., Ebling, M., Kumar, P. and Lu, Q. (1993) 'Experience with Disconnected Operation in a Mobile Computing Environment', In *Proc. 1993* Usenix Symposium on Mobile and Location Independent *Computing*, pages 11–28, Cambridge, MA.

Shi, V. T. S. and Perrizo, W. (2002) 'A New Method for Concurrency Control in Centralized Database Systems', Computers and Their Applications (CATA-2002), ISCA 17th Int'l. Conference, ISBN: 1-880843-42-0.

Tay, J.C., Suri, R., and Goodman, N. (1985) 'A Mean Value Performance Model for Locking in Databases: The No-Waiting Case', Journal of ACM, vol. 32, No. 3, pp. 618-651.

Thomasian, A. (1998) 'Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing', IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 1.

Unland, R. (1994) 'Optimistic Concurrency Control Revisited', Institut für Wirtschaftsinformatik, Münster, Westfalen.

von Eicken, T., Basu, A., Buch, V., and Vogels, W. (1995) 'U-Net: A User-Level Network Interface for Parallel and Distributed Computing', in *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp. 40-53.

Voruganti, K., Özsu, M.T., and Unrau, R.C. (2004) 'An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems', in Distibuted and Parallel Databases, Kluwer Academic Publisher, 15, pp. 137-177.

Wikipedia available at: http://en.wikipedia.org/wiki/Instructions_per_second (Accessed: 10 Jully 2012).

Wikepedia available at: http://en.wikipedia.org/wiki/Bandwidth_(computing) (accessed: 30 Jully 2012).

Wilkinson, K. and Neimat, M.A. (1990) 'Maintaining Consistency of Client-Cached Data', In Proceedings of the Conference Very Large Data Bases (VLDB), Brisbane, Austalia, pp. 122-134.

Zaharioudakis, M., Carey, M.J., and Franklin, M.J. (1997) 'Adaptive, Fine-Grained Sharing a Client-Server OODBMS: A Callback-Based Aproach', ACM Transactions on Database Systems, Vol.22, No. 4, pp. 570–627