1996

An optimistic concurrency control mechanism based on clock synchronization

Myoung Jin Park

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Computer Engineering Commons

Recommended Citation
https://scholarworks.lib.csusb.edu/etd-project/982

This Thesis is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
AN OPTIMISTIC CONCURRENCY CONTROL MECHANISM
BASED ON CLOCK SYNCHRONIZATION

______________________________
A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

______________________________
In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

______________________________
by
Myoung Jin Park
March 1996
AN OPTIMISTIC CONCURRENCY CONTROL MECHANISM
BASED ON CLOCK SYNCHRONIZATION

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Myoung Jin Park
March 1996

Approved by:

Dr. Arturo I. Concepcion, Chair, Computer Science

Dr. Josephine G. Mendoza

Dr. Kay Zemoundeh
ABSTRACT

A new approach for clock synchronization, called “clock rate synchronization”, was developed. Each local clock in a distributed system runs at a constant rate in each machine even though clock rates among them are different due to differences in the crystal frequency. However, if each local machine knows the difference of the clock rate between a known clock rate server and itself, periodic synchronization will not be necessary to synchronize clocks. Clock rate synchronization improves concurrency and eliminates the risk of failure of a single time server during transaction processing in a distributed database system. The clock rate synchronization algorithm requires 4N message exchanges, where N is the number of local machines.

Clock rate synchronization was used to develop an optimistic concurrency control mechanism. In Kung & Robinson’s optimistic concurrency control [6], processing of disordered concurrent transactions is allowed, even though this may be considered as a failure in some distributed database applications. The improved approach uses synchronized clock values so that it can guarantee that earlier requested job will commit first when there exist data intersections among concurrent transactions. The improved optimistic concurrency control mechanism was compared with Kung & Robinson’s and was found to be of comparable performance. Even though the concurrency efficiency was decreased by 1.76% in this method in comparison with Kung & Robinson’s, this method prevents the processing of concurrent disordered transactions.
TABLE OF CONTENTS

ABSTRACT .............................................................. iii
LIST OF FIGURES .................................................. vi

CHAPTER 1. INTRODUCTION AND MOTIVATION ............. 1
  1.1. Clock Synchronization .......................................... 1
  1.2. Concurrency Control Mechanisms .............................. 2
  1.3. Motivation ........................................................ 4
  1.4. Organization of Thesis .......................................... 6

CHAPTER 2. CLOCK SYNCHRONIZATION ..................... 7
  2.1. Lamport's Algorithm ............................................ 7
  2.2. Cristian's Algorithm ............................................ 11
  2.3. Berkeley Algorithm ............................................. 14
  2.4. A New Approach: Clock Rate Synchronization ............ 17
    2.4.1. Clock Rate Synchronization Algorithm .................. 17
    2.4.2. Analysis .................................................... 23

CHAPTER 3. OPTIMISTIC CONCURRENCY CONTROL MECHANISMS ...... 27
  3.1. Kung and Robinson's Method ................................. 27
  3.2. Schlageter's Method .......................................... 31
  3.3. An Improved Approach ........................................ 35
    3.3.1. The Algorithm ............................................ 36
3.3.2. Analysis .............................................. 39

CHAPTER 4. COMPARATIVE PERFORMANCE RESULTS ................. 43

  4.1. Design and Implementation of the Prototype .................. 43
  4.2. Simulation of Load on the System ............................ 45
  4.3. Performance Results ...................................... 46

CHAPTER 5. CONCLUSION AND FUTURE DIRECTIONS ................. 53

  5.1. Improvements to Existing Algorithms ........................ 53
  5.2. Proposed Improvements .................................... 55

APPENDIX A. DESIGN OF THE PROTOTYPE .......................... 57

APPENDIX B. MAINTENANCE OF THE PROTOTYPE ..................... 59

REFERENCES .................................................... 60
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Lamport’s Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Cristian’s Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Estimation of Transmission Time</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Berkeley Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Clock Rate Synchronization</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Generalized Clock Rate Synchronization</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Three Phases of a Transaction</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Validation of Transactions</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Schlageter’s Optimistic Concurrency Control</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison of Kung &amp; Robinson’s Method and Schlageter’s Method</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>An Improved Approach for Optimistic Concurrency Control</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Three Different Cases for Earlier Transactions</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>A Distributed Database Prototype</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Database Objects in Servers</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Test for Average Arrival of Transactions</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance Results for Kung &amp; Robinson’s Method</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Disorder of Transactions</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Changes in Ticks, Rollbacks and Disorders</td>
<td>49</td>
</tr>
</tbody>
</table>
FIGURE 4.7: Performance Results for Improved Algorithm .......................... 50

FIGURE 4.8: Comparison of Time Spent and Disorders .............................. 51

FIGURE 5.1: Four Phases of Optimistic Concurrency Control ...................... 55
CHAPTER 1. INTRODUCTION AND MOTIVATION

In a distributed database system, the data are scattered and replicated on multiple computers. High-speed networks or telephone lines are used to let computers communicate with one another. Each different machine has its own main memory and clock. A distributed database system consists of multiple sites and each site has a local database system. Each site is able to process local transactions as well as participate in the execution of global transactions which are all transactions that access data in other sites including local transactions. Since data are shared and accessed by multiple sites in the global transactions, these sites need to communicate among the sites [5]. The importance of distributed database system has been recognized in terms of data sharing, reliability and availability of data, and speedup of query processing. The ability to share and access data from multiple sites is the primary advantage of distributed database systems. On the other hand, distributed database system also has several disadvantages, including increased software development cost, greater potential for bugs, and increased processing overhead.

1.1. Clock Synchronization

In a distributed system, there exists no global clock since each distributed machine has its own clock. Clock synchronization has been widely recognized as an important requirement in distributed database systems. Various clock synchronization schemes were
used in many distributed system mechanisms, such as checkpointing, interprocess
communication, resource allocation, and transaction processing [8]. Since distributed
systems use distributed algorithms, synchronization is more difficult than in centralized
systems. In a centralized system, a process can make a system call to know the time and
the kernel tells it. However, in a distributed system, achieving agreement on time is not
simple because each different machine has its own physical clock. Various clock
synchronization schemes have been proposed earlier in [2], [3], [4], [7], and [10]. These
studies include logical clock synchronization as well as physical. Applications running on
a given computer that are interested only in the order of events, and not in the absolute
time at which they occurred, require only the value of the counter to timestamp events. On
the other hand, in some real time systems, actual clock time is important. For these
systems, external physical clocks are required.

1.2. Concurrency Control Mechanisms

When several transactions are executing concurrently in the database, the system
needs to control the interaction among the transactions to preserve the data consistency.
The types of concurrency control schemes include lock-based, timestamp-based, and
optimistic concurrency control. In the lock-based algorithm, when a process tries to read
or write a file, it first locks the file. Locks are used to determine the order of transactions
that access the same data items according to the order of arrival of their operations at the
data items. Locking can be done either by a single centralized lock manager or by a local
lock manager on each machine. When a process has already locked some particular files,
the lock manager rejects all further attempts to lock those files [5]. The two-phase
locking protocol allows a transaction to lock a new data item only if it has not yet
unlocked any data item. The lock-based algorithm ensures serializability but is not
deadlock free.

In a timestamp-based algorithm, each transaction is associated with a unique
timestamp. The timestamps of the transactions determine the serializability order. Thus, if
the timestamp of transaction $T_i$ is smaller than the timestamp of transaction $T_j$, then the
scheme ensures that transaction $T_i$ is processed before transaction $T_j$. This is done by
backing up a transaction whenever such an order is violated. This method does not suffer
from deadlocks. However, if transactions arrive too late, they must be aborted.

The lock-based and timestamp-based concurrency control prevent transactions that
make non-serializable schedule at run time. For another approach to concurrency control,
Kung and Robinson [6] proposed the optimistic concurrency control mechanism.
Optimistic concurrency control mechanism assumes that conflicts between transactions
will occur rarely [6]. When a transaction is executed concurrently with other transactions,
no synchronization check is performed. However, at the end of the transaction’s
execution, a validation phase is performed to determine if the transaction has conflicted
with other concurrently running transactions. If the transaction has conflicted, it should be
aborted and rolled back. Thus, the optimistic concurrency control is based on transaction
roll-backs rather than locking.
1.3. Motivation

Each clock in a processor is driven by its own crystal, and crystals can vary slightly in their frequency. Therefore, if a clock is not periodically reset, it will drift from the true time. Most clock synchronization methods set the clock time in either logical or physical methods. These methods may reduce the potential for concurrency when they are used in concurrency control mechanisms since the synchronization should be performed periodically. Therefore, the number of times that synchronization has to be performed is proportional to the number of transactions that have to be processed. One of the important goals of concurrency control is to maximize the concurrency efficiency among the transactions. If there exist a large number of transactions, the time spent for periodic clock synchronization should be seriously considered. For a more efficient concurrency control mechanism, the clock synchronization method that requires less time during the transaction processing is desirable.

Several problems exist in current optimistic concurrency control mechanisms. Processing transactions in correct order is very important in some systems, such as distributed banking system and process control system. In these systems, the transactions should be processed in order according to their starting time under the condition where there exist some data intersections among them. Current optimistic concurrency control mechanisms did not consider the transactions’ starting time so that disorder of transactions may happen. Even in optimistic concurrency control mechanisms, the accurate starting
time of a transaction should be considered. The improved approach uses synchronized
clock values so that it can guarantee that the earlier requested job will commit first when
there exist data intersections among concurrent transactions.

In summary, the following are the reasons for undertaking this study:

a) Synchronization of clocks is an important issue in distributed database systems.
b) Current clock synchronization methods reduce the potential for concurrency since they
   have to be performed periodically during transaction processing.
c) Current clock synchronization methods require a single time server to synchronize
clocks.
d) Current optimistic concurrency control mechanisms used transaction number rather
   than actual clock time so that transactions may not be processed in correct order.
e) Optimistic concurrency control mechanisms need to employ a clock synchronization
   scheme to prevent transactions from being disordered with others in terms of request
   starting time.

An assumption was established for new clock synchronization algorithm and
improved optimistic concurrency control mechanism. We assume that the transmission
time among distributed machines within LAN is equal. In our distributed database
prototype which is presented in Chapter 4, the database objects are not replicated but
partitioned in two servers.
1.4. Organization of Thesis

This paper is organized into five chapters. In Chapter 1, the basic concepts about distributed database system, clock synchronization, and concurrency control mechanisms are mentioned. In Chapter 2, several existing clock synchronization methods are considered in detail. These clock synchronization methods are Lamport’s algorithm, Cristian’s algorithm, and Berkeley algorithm. A new approach for clock synchronization is discussed in the last section of Chapter 2. In Chapter 3, two existing optimistic concurrency control mechanisms are discussed: Kung & Robinson’s and Schlageter’s. In the last section of Chapter 3, the problems of these current optimistic concurrency control mechanisms are discussed. Then, an improved algorithm that solves the drawbacks of current optimistic methods is presented. In Chapter 4, the improved optimistic concurrency control mechanism is compared with Kung & Robinson’s method using a distributed database prototype. Lastly, in Chapter 5, the conclusion and future directions for clock synchronization and optimistic concurrency control mechanism are presented.
CHAPTER 2. CLOCK SYNCHRONIZATION

Several existing clock synchronization algorithms are discussed in this chapter such as Lamport’s Algorithm [7], Cristian’s Algorithm [2], and Berkeley Algorithm [3]. For each of these algorithms, there are some drawbacks that can have an effect on concurrency control efficiency. The worst drawback of these current clock synchronization algorithms is that they have to be performed periodically. In the last section of this chapter, a new clock synchronization algorithm is proposed and analyzed.

2.1. Lamport’s Algorithm

When a distributed system considers only the internal consistency of clocks for synchronization, it is called logical clock synchronization. Physical clock synchronization considers not only the internal consistency of clocks, but also deviation from the real time. Lamport pointed out that clock synchronization need not be absolute [7]. Currently we cannot use the physical true time for ordering any pair of events because it is impossible to generate really accurate synchronized clocks in a distributed system. All processes do not have to agree on exactly what time they have. What they really have to know is the order in which events occur. Lamport defined the “happen before” relation, denoted by “→”, as follows [7].
- If \( a \) and \( b \) are events in the same process, and \( a \) comes before \( b \), then \( a \rightarrow b \).
- If \( a \) and \( b \) are sender and receiver in different processes, then \( a \rightarrow b \).
- If \( a \rightarrow b \) and \( b \rightarrow c \), then \( a \rightarrow c \). If \( a \leftrightarrow b \) and \( b \leftrightarrow a \), then \( a \) and \( b \) are to be concurrent.

Lamport defined a function which assigns a number to event \( a \) in process \( P_i \) as \( C_i(a) \). The entire system of clocks is denoted by the function \( C \) which assigns to any event \( b \) the number \( C(b) \), where \( C(b) = C_j(b) \) if \( b \) is an event in process \( P_j \). Therefore, if \( a \rightarrow b \), then \( C(a) < C(b) \). Each process \( P_i \) increments \( C_i \) between any two successive events. If \( a \) is the sender with message \( m \) by process \( P_i \), \( m \) contains timestamp such as \( T_m = C_i(a) \). \( P_j \), which is receiver of a message \( m \), sets \( C_j \) greater than or equal to its present value and greater than \( T_m \). Lamport's logical clock is a monotonically increasing software counter which means \( C \) must always go forward, never backward. Clock synchronization should be made by adding a positive value, never by subtracting one [13].

Consider the four processes in Figure 2.1.
The processes run on different machines, each with its own clock, running at its own speed. In Figure 2.1, \( P_i \)'s clock ticks 2 times, \( P_j \)'s clock ticks 5 times, \( P_k \)'s clock ticks 3 times, and \( P_l \)'s clock ticks 2 times. Even though the clock runs at a constant rate in each machine, the rates among them are different due to differences in the crystal frequency. Process \( P_i \) sends a message to process \( P_j \) at time 2. Then, \( P_j \) receives the message at time 10 so that \( P_j \) interprets that it took 8 ticks to transfer a message from \( P_i \) to \( P_j \). Now, \( P_j \) sends a message to process \( P_k \) at time 15 and \( P_k \) receives it at time 12. That value is certainly impossible. According to the happen-before relation, \( P_k \) must receive the message from \( P_j \) at time 15 or later. At this point, correction of time should be
applied to $P_k$, which is simply changing the time value from 12 to 16 as shown in the array below it, see Figure 2.1. Every time values in $P_k$ are corrected according to the current time value. The message from $P_k$ to $P_I$ is sent at time 19 and arrived at time 12. The time correction adjusts the time in $P_I$ to 20.

Logical clocks impose only a partial order on the set of all events. The partial orderings on the events in the distributed system can be converted to a total ordering by using the partial ordering of the local clocks. If an event occurs at $P_i$ with local timestamp $T_m$, and another event occurs at $P_j$ with local timestamp $T_n$, we define the global logical timestamps for these events to be $(T_m, P_i)$ and $(T_n, P_j)$ respectively. And, $P_i$ and $P_j$ put those global logical timestamps on their request queues in order. $P_i$ is granted the resource if and only if there is a $(T_m, P_i)$ request resource in its request queue which is ordered before any other request in its queue by the happen-before relation and $T_m$ is the smallest value in every timestamp. The Lamport's algorithm is shown below.

- Lamport's Algorithm

At resource requesting process $P_i$:

- $P_i$ sends $(T_m, P_i)$ request resource to every other processes, and puts that message on its request queue
- $P_i$ receives a timestamped acknowledgment from $P_j$

At receiving process $P_j$:

when $P_j$ receives $(T_m, P_i)$ request resource, it places it on its request queue and sends a timestamped acknowledgment to $P_i$
Lamport’s algorithm has a crucial drawback. Even though Lamport’s algorithm can manipulate the advancements of distributed clocks by exchanging messages, it cannot control those clock values by occurring of internal events. Only message exchanges build paths in Figure 2.1 among distributed processes [12]. Therefore, in order to make event ordering by Lamport’s algorithm in a distributed system, the messages among the distributed machines should be always broadcasted.

2.2. Cristian’s Algorithm

Although Lamport’s algorithm gives a total event ordering, the actual clock time is important in some systems such as real-time systems. In these systems, physical clock synchronization is required. To provide UTC (Universal Coordinated Time) to the systems for precise time, the National Institute of Standard Time (NIST) operates a shortwave radio station with call letters WWV. In order to have all distributed machines stay synchronized with the actual time, at least one of those distributed machines has to have a WWV receiver. Cristian [2] used a central time server which has WWV receiver to synchronize physical clocks.
In the Figure 2.2, when a process $P_i$ requests the time with a message $Req$, the time server replies as fast as it can with a message containing its current time $T_{UTC}$. However, if $P_i$ just sets its clock to $T_{UTC}$, the clock time must be wrong. Since it takes a nonzero amount of time for transferring a message from the time server to $P_i$, the transfer time should be considered. With the simple principle, $P_i$ should set its clock to the time $T_{UTC} + T_{TRANS}$, where $T_{TRANS}$ is the time taken to transmit $T_{UTC}$ from the time server to $P_i$. The $T_{TRANS}$ can be defined as $T_{TRANS} = T_{\text{min}} + T\alpha$, where $T\alpha \geq 0$. The $T_{\text{min}}$ can be obtained when no other processes executed and no other network traffic existed. Unfortunately, $T\alpha$ is subject to variation. To estimate the transmission time between the time server and a process $P_i$, Cristian proposed a method as shown in Figure 2.3.
Cristian suggested to measure the time when $Pi$ starts requesting, $T_{SEND}$, and the time when $Pi$ receives the time server's reply, $T_{REC}$. Therefore, we can see that the message propagation time is $(T_{REC} - T_{SEND}) / 2$. When the time server's reply comes in, the value in the message should be $T_{UTC} + (T_{REC} - T_{SEND}) / 2$. If the time taken by the time server to process the incoming message is known by some method, his method can be improved. With the knowledge of $T_{INT}$ (Interrupt time) in the time server, the transmission time can be defined as $(T_{REC} - T_{SEND} - T_{INT}) / 2$. Cristian's algorithm is shown as follows:

- **Cristian's Algorithm**

At time requesting process $Pi$:

Send a message to time server and Set $T_{SEND} :=$ current time
Receive the interrupt time $T_{INT}$ from time server and Set $T_{REC} :=$ current time
Calculate $T_{TRANS} := (T_{REC} - T_{SEND} - T_{INT}) / 2$
Send $T_{TRANS}$ to time server
Receive the current time from time server and Set its clock
At time server:

- Receive a message from \( P_{i} \)
- Send interrupt time \( T_{\text{INT}} \) to \( P_{i} \)
- Receive time request from \( P_{i} \) with \( T_{\text{TRANS}} \)
- Send \( (T_{\text{UTC}} + T_{\text{TRANS}}) \) to \( P_{i} \)

Cristian’s method suffers from the problem that the single time server machine has all the responsibility for clock synchronization in a distributed system. If the time server fails, then clock synchronization cannot be done. Election algorithm is used to select a new time server in his method. But this increases the complexity of the database system and the cost of software development for clock synchronization. Even after a successful execution of an election algorithm, we cannot guarantee that the time server will not fail again.

### 2.3. Berkeley Algorithm

Gusella and Zatti [3] proposed an algorithm for internal synchronization which is for logical clock synchronization. The time server intends to be active in Berkeley algorithm [3], while it is passive in Cristian’s algorithm. The \textit{master} which is called the time server in Cristian’s algorithm periodically polls the other computers whose clocks are to be synchronized, called \textit{slaves}. If no machine has a WWV receiver in a distributed system, this method will be suitable. The method is illustrated in Figure 2.4.
In Figure 2.4(a), at 2:05, the master sends the other machines, which are slave 1, slave 2, and slave 3, its time and asks for theirs. In Figure 2.4(b), the slaves reply with the time difference from the master’s time. The master calculates the global time as the average of the times provided by the slaves [3]. A clock is considered faulty if its value is away from the specified range of time values of the majority of the other machines. The clock of slave 3 in Figure 2.4 is considered as faulty. In Figure 2.4(c), the master tells each slave how to adjust its clock with the average time value. Every clock in a distributed system is now synchronized in Figure 2.4(d). The Berkeley algorithm is shown as follows:
• **Berkeley Algorithm**

There exist $n$ slaves in a distributed system: \{ *slave 1, slave 2, slave 3, \ldots, slave n* \}

At the master process:

$$T_{TOTAL} := 0 \quad \text{// difference of clock time from master itself}$$

$m := n$

for $i$ from 1 to $n$ do

[ Send its time to *slave i*

  Receive the time difference from *slave i* and Set this to $T_i$

  if ($T_i$ is not faulty) then

    $$T_{TOTAL} := T_{TOTAL} + T_i$$

  else then

    $m := m - 1$

  ]

$T_{AVG} := T_{TOTAL} / (m+1) \quad \text{// m slaves + master process}$

for $i$ from 1 to $n$ do

[ Send ($T_{AVG} - T_i$) to *slave i*

  Set its clock by (current time + $T_{AVG}$) ]

At slave process $S_i$:

Receive the time from *master* and Set this to $T_{MASTER}$

Send (current time - $T_{MASTER}$) to *master*

Receive ($T_{AVG} - T_i$) from *master* and Set this to $T_{ADJ}$

Set its clock by (current time + $T_{ADJ}$)

By using average time value of distributed machines, it can prevent the individual clocks from running too fast or too slow. The accuracy of the protocol depends on a nominal maximum round-trip time between the *master* and the *slaves* [1]. The important difference in this method with the previous ones is that the *master* sends the amount by which each individual *slave*’s clock requires adjustment instead of sending the current time value. The adjustment value can be either a positive or negative value. With this method, the uncertainty of transmission time from *master* to *slaves* can be reduced.

Even though Berkeley algorithm improves clock synchronization in terms of transmission time, the failure of *master* is still a serious problem. It is suggested that if the
master fails, then another can be elected to take over and function exactly as its predecessor. However, we cannot be sure that a new master is elected in bounded time. Also, it is still possible for a new master to fail again during the execution of the election algorithm.

2.4. A New Approach: Clock Rate Synchronization

In previous algorithms for clock synchronization, two important drawbacks can be found. The first problem is that the use of clock synchronization algorithm in a distributed database system can reduce the potential for concurrency among concurrent transactions since synchronization is performed periodically. The other problem is that there exists too much risk due to employing a single time server during the processing of transactions in Cristian's algorithm [2] and Berkeley algorithm [3]. For the first problem, the clock synchronization method that requires only constant time should be established so that the concurrency efficiency is not affected by clock synchronization. For the second problem, to eliminate the risk of failure of a single time server, any machine in a distributed system can synchronize its clock with the help of other machines.

2.4.1. Clock Rate Synchronization Algorithm

In the new approach for clock synchronization, the rates of the local clocks are adjusted by a randomly chosen clock rate server’s rate. Since this method is concerned
with local area network (LAN) without any network bridge, we assume that the transmission time among them within LAN are equal. The transmission time within a LAN is tested in section 2.4.2. The new approach for clock synchronization is illustrated in Figure 2.5.

In Figure 2.5, three processes are involved in the clock rate synchronization. Any process can be chosen to be a \( P_{CRS} \) (Clock Rate Server) temporarily at the beginning of the clock rate synchronization. In order to know the clock rate difference between \( P_{CRS} \) and \( P_i \), we use another process called \( P_c \) which is also chosen randomly. This illustration shows how \( P_c \) finds the difference of clock rates between \( P_{CRS} \) and \( P_i \). In Figure 2.5, process \( P_c \) sends a message to \( P_{CRS} \) and sets its clock as \( T_a(P_c) \) which means the time \( T_a \) by \( P_c \) 's clock. After \( P_{CRS} \) receives the message from \( P_c \), it halts for a fixed amount of time by \( P_{CRS} \) 's clock. \( T_{HALT}(P_{CRS}) \) means that \( P_{CRS} \) holds a message for a fixed amount of time by its clock. After the fixed time later, \( P_{CRS} \) replies to \( P_c \). As soon as \( P_c \) receives
PcRs's reply, it sets its clock as Tb(Pc). At this point, we can get the transmission time between the Pcrs and PC. The \( T_{\text{TRANS}}(\text{Pcrs}, PC) \) refers to the transmission time between the Pcrs and PC and is defined in (2.1).

\[
T_{\text{TRANS}}(\text{Pcrs}, PC) = \left[ Tb(PC) - Ta(PC) - T_{\text{HALT}}(\text{Pcrs}) \right] / 2
\] (2.1)

However, from the point of view of PC, \( T_{\text{HALT}}(\text{Pcrs}) \) is unknown since it is based on Pcrs's clock. From (2.1), the time period from \( Ta(PC) \) to \( Tb(PC) \) includes twice of \( T_{\text{TRANS}}(\text{Pcrs}, PC) \) and \( T_{\text{HALT}}(\text{Pcrs}) \) as follows:

\[
Tb(PC) - Ta(PC) = 2 \times T_{\text{TRANS}}(\text{Pcrs}, PC) + T_{\text{HALT}}(\text{Pcrs})
\] (2.2)

Now, PC sends a message to Pi and sets its clock as Tc(PC). Pi also halts for the same amount of time as Pcrs did, but by Pi's clock which can be written as \( T_{\text{HALT}}(Pi) \). According to the assumption for transmission time in the above, we have to assume that \( T_{\text{TRANS}}(\text{Pcrs}, PC) \) is equal to \( T_{\text{TRANS}}(Pi, PC) \) since our environment is only for LAN.

\[
T_{\text{TRANS}}(Pi, PC) = \left[ Td(PC) - Tc(PC) - T_{\text{HALT}}(Pi) \right] / 2
\] (2.3)

\[
Td(PC) - Tc(PC) = 2 \times T_{\text{TRANS}}(Pi, PC) + T_{\text{HALT}}(Pi)
\] (2.4)
Compare (2.2) with (2.4). Since $T_{\text{TRANS}}(P_{\text{CRS}}, P_{C})$ is equal to $T_{\text{TRANS}}(P_{i}, P_{C})$ by the assumption, the difference between $T_{b}(P_{C}) - T_{a}(P_{C})$ and $T_{d}(P_{C}) - T_{c}(P_{C})$ may be only due to the disagreement of $T_{\text{HALT}}(P_{\text{CRS}})$ and $T_{\text{HALT}}(P_{i})$. Therefore, we can get the clock ratio between $P_{\text{CRS}}$ and $P_{i}$ simply by comparing the value of $T_{b}(P_{C}) - T_{a}(P_{C})$ with $T_{d}(P_{C}) - T_{c}(P_{C})$ as follows:

$$
\text{Clock Ratio}(P_{\text{CRS}}, P_{i}) = \left[ \frac{T_{b}(P_{C}) - T_{a}(P_{C})}{T_{d}(P_{C}) - T_{c}(P_{C})} \right]
$$

Consider the following example. $P_{C}$ sends a message to $P_{\text{CRS}}$ at time 220, $T_{a}(P_{C}) = 220$. $P_{\text{CRS}}$ holds for 100 ticks, $T_{\text{HALT}}(P_{\text{CRS}}) = 100$ and replies to $P_{C}$. $P_{C}$ gets $P_{\text{CRS}}$'s reply at time 330, $T_{b}(P_{C}) = 330$. Then, $P_{C}$ sends a message to $P_{i}$ at time 332 and gets $P_{i}$'s reply at time 452, $T_{c}(P_{C}) = 332$, $T_{d}(P_{C}) = 452$. By (2.2) and (2.4), we get

$$
330 - 220 = 2 \times T_{\text{TRANS}}(P_{\text{CRS}}, P_{C}) + T_{\text{HALT}}(P_{\text{CRS}}), \quad \text{and}
$$

$$
452 - 332 = 2 \times T_{\text{TRANS}}(P_{i}, P_{C}) + T_{\text{HALT}}(P_{i}).
$$

The difference of $T_{\text{HALT}}(P_{\text{CRS}})$ and $T_{\text{HALT}}(P_{i})$ gives the difference of clock rate of $P_{\text{CRS}}$ and $P_{i}$. This difference can be obtained by (2.5).

$$
\text{Clock Ratio}(P_{\text{CRS}}, P_{i}) = \frac{330 - 220}{452 - 332} = 0.917
$$

By informing $P_{i}$ of the clock ratio, $P_{i}$ may reset its clock rate to make it equal to $P_{\text{CRS}}$ logically. Consider Figure 2.6 for a generalized clock rate synchronization.
In Figure 2.6, we have $n$ processes, $P_1$ through $P_n$. $P_{CRS}$ and $P_C$ are chosen randomly among these processes. In this example, $P_2$ is chosen to be $P_{CRS}$ and $P_3$ is selected to be $P_C$. Consequently, the clocks in every process will be set to the same clock rate of $P_{CRS}$. The role of $P_C$ is to simply compare $P_{CRS}$'s clock rate with other clocks' rates and to inform them of the ratio. With the discussion we have so far, the algorithms for the clock rate synchronization can now be written as follows:

**Clock Rate Synchronization Algorithm**

$\text{Process Set} = \{P_1, P_2, P_3, \ldots, P_n\}$

At initiator process $P_{INT}$:

- Choose at random a clock rate server and a coordinator, say $P_{CRS}$ and $P_C$
- Delete $P_{CRS}$ and $P_C$ from Process Set
- Inform $P_{CRS}$ and $P_C$ with Process Set

At coordinator process $P_C$:

- Call Procedure $\text{Sync\_Request}(P_{CRS}, \text{rate1})$  // send request to $P_{CRS}$
- for $i$ from 1 to $n$ do
  - if $P_i \in \text{Process Set}$ do
    - Call Procedure $\text{Sync\_Request}(P_i, \text{rate2})$  // request to $P_i$
Calculate ratio := \((T_b - T_a) / (T_d - T_c)\)

Send ratio to \(P_i\)

Choose one process in Process Set, \(P_k\)

Send a message to \(P_k\)\hspace{1em} // inform \(P_k\)

Call Procedure Sync_Wait(\(P_k\))\hspace{1em} // receive request from \(P_k\)

Receive ratio from \(P_k\)

Receive current time \(T_{CURRENT}\) from \(P_{CRS}\)

At clock rate server process \(P_{CRS}\):

Call Procedure Sync_Wait(\(P_C\))\hspace{1em} // receive request from \(P_C\)

Call Procedure Sync_Wait(\(P_k\))\hspace{1em} // receive request from \(P_k\)

Receive a message from \(P_k\)

for \(i\) from 1 to \(n\) do
[ if \(P_i \in \text{Process Set}\) do
[ Receive a message from \(P_i\)
Send current time \(T_{CURRENT}\) to \(P_i\) ]
]

Send current time \(T_{CURRENT}\) to \(P_C\)

At \(P_i\) in Process Set:

Call Procedure Sync_Wait(\(P_C\))\hspace{1em} // receive request from \(P_C\)

Receive ratio between \(P_{CRS}\) and \(P_i\) from \(P_C\)

if \(P_i = P_k\) do
[ Receive a message from \(P_C\)
Call Procedure Sync_Request(\(P_{CRS}, rate1\))\hspace{1em} // send request to \(P_{CRS}\)
Call Procedure Sync_Request(\(P_C, rate2\))\hspace{1em} // send request to \(P_C\)
Calculate ratio := \((T_b - T_a) / (T_d - T_c)\)
Send ratio to \(P_C\)
Send a message to \(P_{CRS}\) ]

Send a message to \(P_{CRS}\)

Receive \(T_{CURRENT}\) from \(P_{CRS}\)

Procedure Sync_Request(\(P, Rate\))

if \(Rate = rate1\) do
Send request to \(P\), and Set \(T_a := \text{current time}\)
Receive reply from \(P\), and Set \(T_b := \text{current time}\)
else if \(Rate = rate2\)
Send request to \(P\), and Set \(T_c := \text{current time}\)
Receive reply from \(P\), and Set \(T_d := \text{current time}\)
end if
Procedure *Sync_\_Wait* (*P*)

- Receive request from *P*
- Wait for a fixed amount of time *T_{HALT}*
- Send reply to *P*

### 2.4.2. Analysis

**Theorem 2.1:** The clock rate synchronization algorithm correctly computes the clock rate of all processes.

**Proof:** Let *Process Set* = \{ *P_1*, *P_2*, *P_3*, ......, *P_n* \}

A clock rate server *P_{CRS}* and a coordinator *P_C* are chosen to be a clock rate server and a coordinator randomly.

Let \( T_{\text{TRANS}}(P_{\text{CRS}}, P_C) \) = transmission time from *P_{CRS}* to *P_C*,

\( T_{\text{CRS}}(P_C) \) = time when *P_C* sends a message to *P_{CRS}*,

\( T_{\text{CRS'}}(P_C) \) = time when *P_C* gets reply from *P_{CRS}*,

\( T_{\text{HALT}}(P_{CRS}) \) = amount of time halted by *P_{CRS}’s* clock.

Using (2.1) and (2.2), we have

\[ T_{\text{TRANS}}(P_{\text{CRS}}, P_C) = \frac{[T_{\text{CRS'}}(P_C) - T_{\text{CRS}}(P_C) - T_{\text{HALT}}(P_{\text{CRS}})]}{2} \]

\[ T_{\text{HALT}}(P_{\text{CRS}}) = T_{\text{CRS'}}(P_C) - T_{\text{CRS}}(P_C) - [2 \times T_{\text{TRANS}}(P_{\text{CRS}}, P_C)] \]

and then by (2.3), we can compute the transmission time between the coordinator *P_C* and any process *P_i*.
\[ T_{\text{TRANS}}(P_1, P_C) = \frac{[T_1'(P_C) - T_1(P_C) - T_{\text{HALT}}(P_1)]}{2} \]

\[ T_{\text{TRANS}}(P_2, P_C) = \frac{[T_2'(P_C) - T_2(P_C) - T_{\text{HALT}}(P_2)]}{2} \]

\[ T_{\text{TRANS}}(P_3, P_C) = \frac{[T_3'(P_C) - T_3(P_C) - T_{\text{HALT}}(P_3)]}{2} \]

\[ \ldots \ldots \]

\[ T_{\text{TRANS}}(P_n, P_C) = \frac{[T_n'(P_C) - T_n(P_C) - T_{\text{HALT}}(P_n)]}{2} \]

and for the time halted at each process, we have

\[ T_{\text{HALT}}(P_1) = T_1'(P_C) - T_1(P_C) - [2 \times T_{\text{TRANS}}(P_1, P_C)] \]

\[ T_{\text{HALT}}(P_2) = T_2'(P_C) - T_2(P_C) - [2 \times T_{\text{TRANS}}(P_2, P_C)] \]

\[ T_{\text{HALT}}(P_3) = T_3'(P_C) - T_3(P_C) - [2 \times T_{\text{TRANS}}(P_3, P_C)] \]

\[ \ldots \ldots \]

\[ T_{\text{HALT}}(P_n) = T_n'(P_C) - T_n(P_C) - [2 \times T_{\text{TRANS}}(P_n, P_C)] \]

These are summarized in the following table.

<table>
<thead>
<tr>
<th>Process #</th>
<th>Time halted by each process clock</th>
<th>Ratio with ( P_{CRS} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{CRS} )</td>
<td>( T_{\text{HALT}}(P_{CRS}) = \frac{T_{CRS}'(P_C) - T_{CRS}(P_C) - [2 \times T_{\text{TRANS}}(P_{CRS}, P_C)]}{2} )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>( T_{\text{HALT}}(P_1) = \frac{T_1'(P_C) - T_1(P_C) - [2 \times T_{\text{TRANS}}(P_1, P_C)]}{2} )</td>
<td>( T_{\text{HALT}}(P_1) / T_{\text{HALT}}(P_{CRS}) )</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>( T_{\text{HALT}}(P_2) = \frac{T_2'(P_C) - T_2(P_C) - [2 \times T_{\text{TRANS}}(P_2, P_C)]}{2} )</td>
<td>( T_{\text{HALT}}(P_2) / T_{\text{HALT}}(P_{CRS}) )</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>( T_{\text{HALT}}(P_3) = \frac{T_3'(P_C) - T_3(P_C) - [2 \times T_{\text{TRANS}}(P_3, P_C)]}{2} )</td>
<td>( T_{\text{HALT}}(P_3) / T_{\text{HALT}}(P_{CRS}) )</td>
</tr>
<tr>
<td>( \ldots \ldots \ldots )</td>
<td>( \ldots \ldots )</td>
<td>( \ldots \ldots )</td>
</tr>
<tr>
<td>( P_n )</td>
<td>( T_{\text{HALT}}(P_n) = \frac{T_n'(P_C) - T_n(P_C) - [2 \times T_{\text{TRANS}}(P_n, P_C)]}{2} )</td>
<td>( T_{\text{HALT}}(P_n) / T_{\text{HALT}}(P_{CRS}) )</td>
</tr>
</tbody>
</table>
Since \( T_{\text{TRANS}}(P_{CRS}, P_C) \) is equal to \( T_{\text{TRANS}}(P_i, P_C) \) in the assumption,

\[
\frac{T_{\text{HALT}}(P_i)}{T_{\text{HALT}}(P_{CRS})} = \left[ T_{i}'(P_C) - T_i(P_C) \right] / \left[ T_{CRS}'(P_C) - T_{CRS}(P_C) \right].
\]

Therefore, \( \left[ T_{i}'(P_C) - T_i(P_C) \right] / \left[ T_{CRS}'(P_C) - T_{CRS}(P_C) \right] \) gives the clock ratio between \( P_{CRS} \) and \( P_i \). □

The assumption for transmission time was tested within a LAN. Three machines were chosen to be tested to measure the transmission time among them. The test was performed by exchanging messages for 100,000 times among these machines during a day time which is when the transmission time is relatively long and not very predictable. Then, the total time spent is averaged to get the transmission time for transferring a message. These machines include blaze, indigo, and aviion and the test results are shown as follows:

\[
\begin{align*}
\text{blaze:} & \quad \text{indigo} - 0.001190, \text{aviion} - 0.001680 \\
\text{indigo:} & \quad \text{blaze} - 0.001140, \text{aviion} - 0.0011240 \\
\text{aviion:} & \quad \text{blaze} - 0.001690, \text{indigo} - 0.001170
\end{align*}
\]

These time values are written by ticks. Even though these transmission times are not equal, they are very short periods of time so that they cannot be measured by system call. Since these transmission time are much less than one tick, we have to assume that they are all equal. The clock rate synchronization algorithm was tested below.
This test was performed by measuring 1000, 2000, and 3000 ticks using synchronized clock in each local machine, reporting the current time of these machines to clock rate server (blaze), and checking if the reported time are equal to the time in clock rate server. By running clock rate synchronization algorithm in each local machine in parallel, the algorithm can be optimized.

As mentioned earlier, the existing clock synchronization methods have several problems such as having a crucial effect on concurrency efficiency, and employing a single time server until every transaction is finished. Cristian’s algorithm requires $4N \times M$ message exchanges where $N$ is the number of local machines and $M$ is the number of periodic synchronization performance. Berkeley algorithm requires $3N \times M$ message exchanges. The number of message exchanges in Lamport’s algorithm is subject to the number of events among distributed processes. The clock rate synchronization requires 4N message exchanges since synchronization is performed once at initialization. Therefore, clock rate synchronization does not affect concurrency efficiency. This method employs a randomly chosen CRS (Clock Rate Server) and a coordinator only during the synchronization period at the beginning temporarily. After a certain time period when synchronization has been made, CRS and coordinator are not to be maintained any more since now every clock knows the global clock rate in a distributed system.

<table>
<thead>
<tr>
<th>Local Machine</th>
<th>Initial Tick</th>
<th>1000 ticks later</th>
<th>2000 ticks later</th>
<th>3000 ticks later</th>
</tr>
</thead>
<tbody>
<tr>
<td>blaze ($P_{CRS}$)</td>
<td>818189261</td>
<td>818190261</td>
<td>818191261</td>
<td>818192261</td>
</tr>
<tr>
<td>indigo ($P_C$)</td>
<td>818189261</td>
<td>818190261</td>
<td>818191261</td>
<td>818192261</td>
</tr>
<tr>
<td>aviion ($P_I$)</td>
<td>818189261</td>
<td>818190261</td>
<td>818191261</td>
<td>818192261</td>
</tr>
</tbody>
</table>
CHAPTER 3. OPTIMISTIC CONCURRENCY CONTROL MECHANISMS

Two existing optimistic concurrency control mechanisms will be discussed in this chapter. These mechanisms are proposed by Kung & Robinson [6] and Schlageter [11]. The algorithms will be presented in detail and they will be compared with each other. Since both methods did not consider clock synchronization schemes, they have crucial drawbacks in terms of ordering transactions. In the last section of this chapter, an improved approach is presented to show how the drawbacks of the current optimistic concurrency control algorithms can be solved.

3.1. Kung and Robinson’s Method

Kung and Robinson [6] proposed a method called “optimistic” for concurrency control. This method can be regarded as centralized database system since there is no global clock for ordering events. They assume that conflicts among concurrent transactions are very rare so that locking may be necessary only in the worst case. A transaction always executes concurrently with other transactions without any synchronization check, but it is validated before its writes are written in the database [12]. Consider Figure 3.1 for the three phases of a transaction.
As shown in Figure 3.1, any transaction consists of three phases: a read phase, a validation phase, and a possible write phase. During the read phase, each transaction has a tentative version of the data items that it updates. All updates are made on local copies of the database objects. With these local copies of data items, we can abort or roll back a certain transaction that fails validation with no effect on the original database. Transactions that involve only reading are performed immediately. If the local copies for that transaction already exist, a read transaction accesses it, otherwise it accesses the most recently committed value of the data item [1]. Write transactions record the new values of the data items as tentative values. Therefore, a database system may have different values of a certain database object among the concurrent transactions. In addition, each transaction has two records which are a read set and a write set. A read set contains the data items which are read by the transaction. A write set contains the data items which are written by the transaction. After the read phase, the transaction is validated to check if its operations conflict with other transactions’ operations on particular data items. If the transaction fails in validation phase, it needs to be rolled back, otherwise it is committed. During the write phase, all updates recorded in the local copies
are made permanent. Read only transactions may commit immediately after the validation phase. Write transactions are ready to commit once the local copies of the data items have been recorded in permanent storage. This method is illustrated in Figure 3.2.

<table>
<thead>
<tr>
<th>T&lt;sub&gt;i&lt;/sub&gt;</th>
<th>Read &lt;15&gt; Valid</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>&lt;16&gt;</td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;CURRENT&lt;/sub&gt;</td>
<td>&lt;17&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Validation Phase and Write Phase are in Critical Section shown by "".

**FIGURE 3.2: Validation of Transactions**

Each transaction is assigned a transaction number when it enters the validation phase. In some optimistic concurrency control mechanisms, timestamps are used rather than transaction number [9]. However, using timestamps doesn’t make any difference in the algorithm since the timestamps are obtained from local machine. If the transaction is validated, it retains this number, otherwise it is aborted. If the transaction is for read only, the number is released for re-assignment. In Figure 3.2, three concurrent transactions are shown: T<sub>i</sub>, T<sub>2</sub>, T<sub>CURRENT</sub>. T<sub>i</sub> and T<sub>2</sub> are previously committed transactions and T<sub>CURRENT</sub> is the current active transaction. T<sub>i</sub>, T<sub>2</sub>, and T<sub>CURRENT</sub> are assigned transaction numbers as 15, 16, 17 respectively at the beginning of the validation phase. In the validation phase of
the transaction $T_{CURRENT}$, all transactions $T_i$ which had their write phase of $T_{CURRENT}$ are considered. In this illustration, $T_1$ and $T_2$ can be committed during the read phase of $T_{CURRENT}$. Next, the validation of transaction $T_i$ checks whether its read set intersects with any of the write sets of earlier overlapping transactions $T_1$, $T_2$. Let $start\ tn$ be the highest transaction number at the start of transaction $T_{CURRENT}$, let $finish\ tn$ be the highest transaction number at the beginning of the validation phase of $T_{CURRENT}$. Then, the validation phase of optimistic concurrency control algorithm is performed as follows:

- **Kung & Robinson’s Algorithm**

  ```
  <start CS>  // start of CS
  finish tn := tnc;
  valid := true;
  for $T_i$ from $start\ tn + 1$ to $finish\ tn$ do
    if (write set of $T_i$ intersects read set of $T_{CURRENT}$) then valid := false;
    if valid then ((write phase); $tnc := tnc + 1$; $tn := tnc$) <end CS>;  // end of CS
    else then (backup)
  </end CS>  // end of CS
  ``

The assignment of transaction number, validation phase, and the write phase are all in a critical section. The transaction is assigned a transaction number in the execution of $tnc := tnc + 1$; $tn := tnc$. The transaction numbers are assigned only if validation is successful [6].
We can find several problems in Kung & Robinson's method for concurrency control. First of all, with their method, it is impossible for a server to know when a client really starts requesting a transaction. There might be some transmission delay when a client sends the server a request. The read phase is supposed to start when a client begins requesting. However, in this method, the read phase starts when the server receives the client's request. Since Kung & Robinson did not consider clock synchronization among the distributed machines, this drawback cannot be solved. Another problem is due to the critical section for the validation and write phases. Even in the case where there are no data intersections among the concurrent transactions, this method still requires a critical section for the validation and write phases. This can decrease concurrency efficiency.

3.2. Schlageter's Method

In Kung and Robinson's method, read transactions that do not have a write phase have to be validated. The main difference between read transactions and write transactions is that validation of read transactions need not be done in a critical section. Schlageter [11] proposed another optimistic approach for concurrency control. In Schlageter's method, read transactions are free from all burden of concurrency control. Write transactions are assigned all responsibility for concurrency control. Kung and Robinson treated read transactions in the same way as write transactions in terms of validation. However, in a query dominant system, it is desirable to treat read transactions
in a different way from write transactions. Schlageter’s idea is to let read transactions always proceed and terminate without any consideration of concurrency control. Therefore, read transactions do not have validation phase while write transactions have to consider not only concurrent write transactions but also concurrent read transactions. Conflict between a write transaction and another write transaction results in a backup of the write transaction. Conflict between a write transaction and a read transaction results in deferring the write transaction. The validation phase is shown as follows:

- **Schlageter’s Algorithm**

$T_{\text{CURRENT}}$ is the current write transaction.

\[ L: \text{waitset} := \emptyset \]

\begin{verbatim}
<start CS>
for all $Ti \in \{ \text{active read transaction} \}$ do
  if (write set of $T_{\text{CURRENT}}$ intersects with read set of $Ti$)
    then waitset := waitset $\cup \{ t \}$;

if waitset $\neq \emptyset$
  then <end CS> wait(waitset);
  goto L;

validation as to update transactions;
write phase <end CS>
\end{verbatim}

The critical section is indicated by “<start CS>” and “<end CS>.” The “wait(waitset)” means that current transaction $T_{\text{CURRENT}}$ should wait until all active transactions in the waitset are terminated. If $T_{\text{CURRENT}}$ detects a read transaction which accessed the data objects that $T_{\text{CURRENT}}$ will update, $T_{\text{CURRENT}}$ has to wait until the read
transaction is terminated. The illustration for this method is given in Figure 3.3.

![Schlageter's Optimistic Concurrency Control](image)

**FIGURE 3.3: Schlageter’s Optimistic Concurrency Control**

In Figure 3.3, the current transaction $T_{CURRENT}$ considers every transaction $Ti$ which are in the read phase at the beginning of its validation phase. Having the read phase in other concurrent transactions during $T_{CURRENT}$'s validation phase indicates that $T_{CURRENT}$ may write on some data objects while $Ti$ is reading those data objects. Therefore, we have to consider the data intersections between $T_{CURRENT}$'s write set and $Ti$'s read set. In this example, $T_1$ and $T_2$ are having the read phase at that point. The validation of $T_{CURRENT}$ compares its write set with the read set of $T_1$ and $T_2$. Consider Figure 3.4 for comparison of Kung & Robinson's method and Schlageter's method.
A write transaction $T_{WRITE}$ runs concurrently with several read transactions $T_{R1}$, $T_{R2}$, and $T_{R3}$ in Figure 3.4. The dotted line shows that $T_{WRITE}$ reaches its end while the read transactions are active. In this case, we can distinguish two different results from Kung & Robinson's method and Schlageter's method.

- Kung & Robinson's method:

  $T_{WRITE}$ will get positive validation since it does not consider the read transactions.

  Every read transaction $T_{R1}$, $T_{R2}$, and $T_{R3}$ has to be validated with respect to the previously committed $T_{WRITE}$.

- Schlageter's method:

  $T_{WRITE}$ has to check if its write set is intersected with the read sets of $T_{R1}$, $T_{R2}$, and $T_{R3}$. If a conflict is detected with any $T_R$, $T_{WRITE}$ is delayed until the end of $T_R$. No backup can occur.
Schlageter’s method suffers from the risk of indefinite delay of write transactions. Schlageter proposed a solution for the risk in which the process for a write transaction counts the number of cycles in the validation phase. After this number exceeds some value, some global lockout mechanisms are applied. This solution should be sufficient since the probability of conflict is assumed to be low. However, we cannot know how many number of cycles in the validation phase will be appropriate. The number of cycles should depend on the probability of conflicts.

3.3. An Improved Approach

In the existing optimistic concurrency control, even though a transaction $T_i$ starts later than another transaction $T_j$ and there exist some data intersections between $T_i$ and $T_j$, it is possible for $T_i$ to commit earlier than $T_j$. That is there is a disorder of concurrent transactions. The disorder of the transactions is the main problem with the existing optimistic concurrency control mechanisms. These methods just ignore when a certain transaction really started. What they are really concern with is when the read phase ends and when the validation phase starts. However, transaction processing in correct order according to the transaction’s starting time is essential in some systems, such as distributed banking systems and process control systems. In these systems, the disorder of transactions means failure of entire systems.
3.3.1. The Algorithm

The database objects are not replicated but partitioned in different servers in our distributed prototype. In the improved approach for optimistic concurrency control, the time when a client really starts requesting is the important key. The accurate starting time in a distributed system can be obtained by clock synchronization. Consider Figure 3.5 for this method.

![Diagram of the algorithm](image)

**FIGURE 3.5: An Improved Approach for Optimistic Concurrency Control**

In Figure 3.5, concurrent transactions can be divided into two different cases. Both cases are viewed at the beginning of $T_{CURRENT}$'s validation phase. Since we are concerned with the time when the $T_{CURRENT}$'s validation phase starts, the dotted boxes
show the phases that have not yet committed. The dotted lines show the start and end of $T_{CURRENT}$'s read phase. What we have to be concerned with at the beginning of validation phase is the events that happened during $T_{CURRENT}$'s read phase. The two cases are the possible events which are to be validated. Next, two different cases are explained and analyzed.

- **case (a):**
  
  This is the case when $T_{CURRENT}$ finds a transaction $T_i$ that started earlier than $T_{CURRENT}$ and committed write phase during the read phase of $T_{CURRENT}$. In this case, we have to check if the write set of $T_i$ intersects with the read set of $T_{CURRENT}$.

- **case (b):**
  
  This is the case when $T_{CURRENT}$ finds a transaction $T_i$ that started earlier than $T_{CURRENT}$ and did not finish its transaction. For the correct transaction order, $T_{CURRENT}$ should read the data objects after $T_i$ updates them if the data objects are intersected between $T_i$ and $T_{CURRENT}$. In this case, we have to check both if the read set of $T_i$ intersects with the write set of $T_{CURRENT}$ and if the write set of $T_i$ intersects with the read set of $T_{CURRENT}$.

With these two possible cases, we can build an improved optimistic approach algorithm as follows:
• The Algorithm

$T_{CURRENT}$ is the current transaction.

Call Procedure Clock Rate Synchronization

1: \( \text{valid} := \text{true}; \)
2: for all transactions $T_i \in \{ \text{active transactions during } T_{CURRENT}'s \text{ read phase} \}$
3: \( \text{if ( start of } T_i \text{'s read phase } < \text{ start of } T_{CURRENT}'s \text{ read phase } )} \)
4: \( // \text{ case (a) in Figure 3.5} \)
5: \( \text{if ( } T_i \text{'s write phase intersects with } T_{CURRENT}'s \text{ read phase } )} \)
6: \( \text{if ( } T_i \text{'s write set } \cap T_{CURRENT}'s \text{ read set } \neq \emptyset )} \)
7: \( \text{valid} := \text{false and restart;} \)
8: \( // \text{ case (b) in Figure 3.5} \)
9: else
10: \( \text{if ( } T_i \text{'s read set } \cap T_{CURRENT}'s \text{ write set } \neq \emptyset )} \)
11: \( \text{valid} := \text{false and restart;} \)
12: \( \text{else if ( } T_i \text{'s write set } \cap T_{CURRENT}'s \text{ read set } \neq \emptyset )} \)
13: \( \text{valid} := \text{false and restart;} \)

In the improved algorithm, any transaction $T_i$ which was active during the read phase of $T_{CURRENT}$ and started earlier than $T_{CURRENT}$ is considered to be validated. The case (a) and (b) are distinguished in the algorithm. In case (a), if the write phase of $T_i$ is intersected with the read phase of $T_{CURRENT}$, the write sets of $T_i$ are compared with the read sets of $T_{CURRENT}$. When there are data intersections, $T_{CURRENT}$ should fail and be restarted. $T_{CURRENT}$ should have read the data objects after $T_i$ finished its transaction. The case (b) implies that $T_i$ started earlier than $T_{CURRENT}$ and did not finish its transaction. Notice that the transaction $T_i$ is being processed at the validation of $T_{CURRENT}$. At this point, we don't know which of $T_i$ or $T_{CURRENT}$ will have write phase first. Therefore, we have to consider not only between the read set of $T_i$ and the write set of $T_{CURRENT}$ but also
between the write set of \( Ti \) and the read set of \( T_{CURRENT} \). By looking at both sides of the sets, we can guarantee that \( T_{CURRENT} \) will not write until \( Ti \) read the data, and \( T_{CURRENT} \) will not read the false data which are supposed to be updated by \( Ti \) first.

### 3.3.2. Analysis

Theorem 3.1: The algorithm processes the conflicted transactions in the correct order of their timestamps.

Proof: At the validation phase of current transaction \( T_{CURRENT} \), we have to distinguish three different cases for earlier started transactions \( T_1, T_2, \) and \( T_3 \) as shown blow.

![Figure 3.6: Three Different Cases for Earlier Transactions](image)

Let \( \alpha \) be the time when \( T_{CURRENT} \) starts its read phase, \( \beta \) be the time when \( T_{CURRENT} \) finishes its read phase, and \( INT(\alpha, \beta) \) be the interval between \( \alpha \) and \( \beta \). Assume that there are data intersections between \( T_{CURRENT} \) and \( T_1, T_2, \) and \( T_3 \).
(1) \(T_i \notin INT(\alpha, \beta)\):

\(T_i\) is not considered in validation since it is not intersected with \(T_{CURRENT}\).

(2) write phase of \(T_2 \in INT(\alpha, \beta)\):

\(T_{CURRENT}\) is restarted by lines 5 ~ 7 in the algorithm.

(3) write phase of \(T_3 \notin INT(\alpha, \beta)\), and

read phase or validation phase of \(T_3 \in INT(\alpha, \beta)\):

\(T_{CURRENT}\) is restarted by lines 9 ~ 13 in the algorithm.

The other transactions that started later than \(T_{CURRENT}\) are not considered in validation for the correct order of transaction processing. Since the algorithm covers all possible three cases for conflicting transactions, it can guarantee that the concurrent transactions are processed in the correct order of their timestamps.

At this point, we face another obstacle with the improved algorithm. Consider the situation where it takes a very long time for a client's request to arrive at a server due to communication delay in the network. Of course, the improved algorithm does not work in this case. However, notice that we assumed that our study is concerned with only LAN. In the assumption earlier, the transmission time among machines within LAN are of equal amount. Since the improved algorithm does not require any critical section, the time spent from when a query is produced in a client to when the query has arrived at a server can be regarded as the same among the machines. The main point of communication delay
problem is under the transmission time and the time spent for read phase. As long as the maximum transmission time is much less than the time spent for any transaction's read phase, then the improved algorithm can work properly. The time spent for read phase was tested by reading 10 data objects which are discussed in section 4.1 for 100,000 times in blaze, indigo, and aviion. The test results are shown in below.

<table>
<thead>
<tr>
<th>System</th>
<th>Time for Reading 10 Data Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>blaze</td>
<td>0.224 tick</td>
</tr>
<tr>
<td>indigo</td>
<td>0.042 tick</td>
</tr>
<tr>
<td>aviion</td>
<td>0.071 tick</td>
</tr>
</tbody>
</table>

The time spent for read phase ranges from 0.042 to 0.224 and transmission time ranges from 0.001140 to 0.001690 as shown in section 2.4.2. Therefore, the improved algorithm works properly in this LAN. However, for systems that require outside LAN communications, additional mechanisms should be studied to solve the communication delay problem. In this study, both clock synchronization and optimistic concurrency control are applicable only in a LAN.

By applying clock synchronization scheme to the optimistic concurrency control, the time when a client really starts requesting a job could be more accurate than when transaction number is used. Kung & Robinson's method can be regarded as centralized database system since it didn't consider the global clock in a distributed system. The improved optimistic concurrency control mechanism employs clock rate synchronization.
so that every machine has the global clock in a distributed database system. Furthermore, Kung & Robinson’s method and Schlageter’s method suffer from reducing the potential for concurrency due to keeping critical sections for the validation and write phases. Even though improved approach requires more comparisons with concurrent transactions and has more possibility to fail in validation phase, it is expected that the improved algorithm may be comparable with the existing algorithms since the critical section is eliminated. But the main advantage of the improved approach for concurrency control is that this method can guarantee that earlier transactions will be processed first than other transactions with later timestamps when there exist data intersections among these transactions.
CHAPTER 4. COMPARATIVE PERFORMANCE RESULTS

In this chapter, a distributed database prototype is presented in terms of the structure of data objects and the kinds of queries. Then, a simulation of load on the system will be discussed for the arrival of queries. Lastly, the performance results will be compared between Kung & Robinson's [6] method and the improved approach based on clock synchronization. Since the idea of Schlageter's [11] method was mostly based on Kung & Robinson's method, we just focus on comparing Kung & Robinson's method with the improved algorithm.

4.1. Design and Implementation of the Prototype

For testing both algorithms, Kung & Robinson's and the improved algorithm, a distributed database prototype was developed. The prototype is concerned with a distributed banking system. Consider Figure 4.1 for this prototype.

![Diagram](image)

**FIGURE 4.1: A Distributed Database Prototype**
In Figure 4.1, we have two servers which have database objects and three clients which request queries. The database objects are not replicated but partitioned in servers. The maximum number of concurrent transactions is three. The database objects in the servers consist of the following items: account number, saving/checking, balance, name, address, and phone number. The data are divided into twenty objects which are written by "page" in Figure 4.2. Page #1 through page #10 are stored in server #1, and page #11 through page #20 are stored in server #2.

![Database Objects in Servers](image)

**FIGURE 4.2: Database Objects in Servers**

With these pages, we can define both read sets and write sets of a transaction. The different kinds of query jobs in each client are described as follows:

- **Query #1**: Reading the data items for saving/checking and balance.
- **Query #2**: Reading the data items for name, address, and phone number.
- **Query #3**: Modifying the data item balance.
• Query #4: Adding a new tuple.

• Query #5: Deleting a tuple.

Query #1 and #2 are for read transactions and Query #3, #4, and #5 are for write transactions. Appendix A shows the pseudo codes for the server and client systems and sample queries and data table. Appendix B contains the locations of source and executable files for the distributed database prototype.

4.2. Simulation of Load on the System

In order to produce queries in each client with appropriate delay, we need Poisson probability distributed arrival time. The simulation of transaction arrival time can be obtained by the following relation.

\[ x = -\frac{1}{\lambda} \ln(1 - y) \]  

(4.1)

where \( x \) = random number which has a Poisson probability distribution

\( \lambda \) = average arrival of transactions

\( y \) = uniformly distributed random numbers, \( 0 < y < 1 \)

By generating the value of \( y \) between 0 and 1, we can control the average delay time during the entire simulation. The graph in Figure 4.3 shows the average delay time
according to $\lambda$ value. These average delay time were obtained by running (4.1) 1,000,000 times and averaging them.

![Graph showing average delay time vs $\lambda$]

**FIGURE 4.3: Test for Average Arrival of Transactions**

As shown in Figure 4.3, the average delay time decreases as $\lambda$ increases. These average delay time are used in the comparative performance analysis. The value of $\lambda$ can be classified into three types of loads: 0.1 - 0.3 for lightly loaded system, 0.4 - 0.7 for moderately loaded system, and 0.8 - 0.9 for heavily loaded system.

4.3. Performance Results

In the prototype, we process one thousand transactions and compare the performance of Kung & Robinson’s method with the improved algorithm. The improved
algorithm requires more comparisons for concurrent transactions during the validation phase to prevent concurrent transactions from being disordered. Even though more comparisons can affect the concurrency, our main concern is to solve the problem of processing disordered transactions in Kung & Robinson's algorithm. The performance of the improved algorithm is comparable to Kung & Robinson's because we don't need to keep critical sections in validation and write phases. Figure 4.4 shows the performance results using Kung & Robinson's algorithm.

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>Ticks / 1000 Jobs</th>
<th>Ticks per Job</th>
<th>Rollbacks / 1000 Jobs</th>
<th>Disorders / 1000 Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1084</td>
<td>1.084</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0.2</td>
<td>628</td>
<td>0.628</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0.3</td>
<td>445</td>
<td>0.445</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>0.4</td>
<td>357</td>
<td>0.357</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>0.5</td>
<td>319</td>
<td>0.319</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>0.6</td>
<td>287</td>
<td>0.287</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>0.7</td>
<td>257</td>
<td>0.257</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>0.8</td>
<td>247</td>
<td>0.247</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>0.9</td>
<td>233</td>
<td>0.233</td>
<td>5</td>
<td>43</td>
</tr>
</tbody>
</table>

**FIGURE 4.4: Performance Results for Kung & Robinson's Method**

In the first column of Figure 4.4, \( \lambda \) values are shown from 0.1 though 0.9. The number of ticks spent for processing 1,000 transactions and for processing one transaction are shown in the second and third column respectively. The number of ticks spent are obtained by averaging the time spent in the two servers. The fourth column shows the number of rollbacks per 1,000 transactions. The number of rollbacks indicates the number of failures in the validation phase. The last column of Figure 4.4 implies that Kung &
Robinson's method has the crucial drawback compared with the improved algorithm. This column shows the number of transaction disordering per 1,000 transactions. Figure 4.5 shows two kinds of cases arising in transaction disordering.

\begin{figure}
\centering
\includegraphics[width=1\textwidth]{figure4.5.png}
\caption{Disorder of Transactions}
\end{figure}

In Figure 4.5, transactions $T_1$ and $T_2$ are regarded as the disordered transactions if there exist some data intersections with the current transaction $T_{\text{CURRENT}}$. For $T_1$, we have to check either if the write sets of $T_i$ intersect with the read sets of $T_{\text{CURRENT}}$ or if the read sets of $T_i$ intersect with the write sets of $T_{\text{CURRENT}}$. If there are data intersections between them, transaction $T_1$ should have updated the data objects after $T_{\text{CURRENT}}$ has been done since $T_{\text{CURRENT}}$ started earlier than transaction $T_1$. Similarly, transaction $T_2$ should be regarded as disordered with $T_{\text{CURRENT}}$ if their read sets and write sets are intersected. Even though transaction $T_2$ should be processed first in this case, we don't know which of the transactions will be processed first at this point. In some certain systems, such as distributed banking systems and process control systems, the disorder of transactions means failure of the entire systems.

In Figure 4.4, as $\lambda$ value increases, the system is getting heavily loaded and the
time spent for processing the certain number of transactions decreases. The number of rollbacks due to failing validation and the number of disorders transactions increases. In Figure 4.6, we present several graphs that show the changes in ticks, number of rollbacks, and the number of disorders as $\lambda$ value changes. These graphs are corresponded with the results of Figure 4.4.
FIGURE 4.6: Changes in Ticks, Rollbacks, and Disorders

Next, the performance results of the improved algorithm are shown in Figure 4.7.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>Ticks / 1000 Jobs</th>
<th>Ticks per Job</th>
<th>Rollbacks / 1000 Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1109</td>
<td>1.109</td>
<td>48</td>
</tr>
<tr>
<td>0.2</td>
<td>634</td>
<td>0.634</td>
<td>65</td>
</tr>
<tr>
<td>0.3</td>
<td>454</td>
<td>0.454</td>
<td>72</td>
</tr>
<tr>
<td>0.4</td>
<td>372</td>
<td>0.372</td>
<td>74</td>
</tr>
<tr>
<td>0.5</td>
<td>314</td>
<td>0.314</td>
<td>98</td>
</tr>
<tr>
<td>0.6</td>
<td>305</td>
<td>0.305</td>
<td>124</td>
</tr>
<tr>
<td>0.7</td>
<td>269</td>
<td>0.269</td>
<td>130</td>
</tr>
<tr>
<td>0.8</td>
<td>247</td>
<td>0.247</td>
<td>135</td>
</tr>
<tr>
<td>0.9</td>
<td>221</td>
<td>0.221</td>
<td>147</td>
</tr>
</tbody>
</table>

FIGURE 4.7: Performance Results for Improved Algorithm

In the improved algorithm, there is no transaction disorder since every condition that may result in disorder has been considered during the validation phase. There are a
lot of rollbacks of transactions due to failing validation in the last column in Figure 4.7. These rollback operations have been performed to prevent the transactions from being disordered. The time spent and disorders of transactions are compared between Kung & Robinson’s method and our method in Figure 4.8.

![Comparison of Time](image1)

![Comparison of Disorders](image2)

**FIGURE 4.8: Comparison of Time Spent and Disorders**
In Figure 4.8, the first graph shows time spent processing 1,000 transactions in Kung & Robinson’s algorithm and in the improved algorithm. As you can see from the graph, our method is comparable to theirs since critical section is not required any more.

In addition to the comparative performance result, the problem of transaction disorder can be solved with the improved algorithm shown in the second graph of Figure 4.8.
CHAPTER 5. CONCLUSION AND FUTURE DIRECTIONS

We have developed a clock rate synchronization algorithm and used this algorithm to design an improved optimistic concurrency control mechanism. This chapter will summarize the important results of this thesis and present future directions in the last section.

5.1. Improvements to Existing Algorithms

Through this study, several important concepts have been considered to improve concurrency control efficiency in a distributed database system. First, several existing algorithms for clock synchronization were studied. These algorithms were proposed by Lamport [7], Cristian [2], and Gusella & Zatti [3]. Lamport defined the "happen before" relation for event ordering. In the view of Lamport for clock synchronization, what all distributed processes have to know is the order in which events occur. To order any pair of events, the physical true time is not really important. However, for ordering events by Lamport's algorithm, the messages among the distributed machines should always be broadcasted. Cristian used a central time server that has WWV receiver for physical clock synchronization. Cristian also proposed a method to estimate the transmission time among distributed machines. This method suffers from the fact that a single time server has all responsibility for clock synchronization all the time during transaction processing.
In the Berkeley algorithm, the estimation of transmission time was improved. The time server sends the amount by which each individual machine's clock requires adjustment instead of sending the current time value. The failure of the time server is still a serious problem in this method. In the new approach for clock synchronization, clock rates between a clock rate server and a distributed machine are considered rather than their clock time. It requires only a single synchronization performance at initialization to synchronize clocks. A time server which is CRS (Clock Rate Server) in the new approach needs to be maintained only during the clock synchronization period at initialization.

Optimistic concurrency control mechanisms were proposed by Kung & Robinson [6] and Schlageter [11]. The most serious problem of these optimistic concurrency control mechanisms is the processing of disordered transactions. In the improved approach, the clock rate synchronization method was applied to the optimistic concurrency control mechanism to solve the problem of disordered transactions. Kung & Robinson's method can be regarded as centralized database system since it didn't consider the global clock in a distributed system. The improved optimistic concurrency control mechanism employs clock rate synchronization so that every machine has the global clock in a distributed database system. Even though the concurrency efficiency was decreased by 1.76% in comparison with Kung & Robinson's method, this method prevents the processing of concurrent disordered transactions. However, we still have a limitation for both the clock rate synchronization algorithm and the improved approach of optimistic concurrency control. The improved algorithms work only within LAN since the transmission time among the distributed machines can be delayed.
5.2. Proposed Improvements

The improved optimistic concurrency control mechanism is restricted within LAN. This limitation occurs from communication delay in a LAN with bridges or in a metropolitan area network (MAN). Consider Figure 5.1 for suggested optimistic concurrency control mechanism.

![FIGURE 5.1: Four Phases of Optimistic Concurrency Control](image)

In Figure 5.1, a transaction consists of four phases rather than three phases. After the write phase, the additional phase which is called "Delay Phase" is to be performed. During the delay phase, copies of data objects, which are the data before the current transaction had write phase, are maintained for a certain appropriate amount of time. It does not mean that the subsequent transaction should wait until the end of delay phase. The next transaction may be overlapped with the delay phase of the current transaction. Therefore, the delay phase makes it possible to roll back the current transaction even after the write phase for a certain amount of time. In order to design this mechanism, several complex techniques are required to preserve the data consistency, such as determining the...
appropriate amount of time for delay phase and maintaining the local copies of data objects during delay phase.
APPENDIX A. DESIGN OF THE PROTOTYPE

The pseudo codes of two servers and three clients are shown below.

At Server Process, $S1$ and $S2$:

Make_Connection(); // connection established with other machines
Clock_Synchronization(); // clock rate synchronization

\[
\text{process id} := 1;
\text{forever [ // generating concurrent processes for each client}
\quad \text{if fork()} \neq 0
\quad \text{break;}
\quad \text{process id} := \text{process id} + 1;
\quad \text{if process id} \geq \text{NUM CLIENT} \quad // \text{NUM_CLIENT: number of clients}
\quad \text{break;}
\]
\[
\text{client id} := \text{process id};
\]
\[
\text{forever [}
\quad \text{Receive Request(client id); // receiving requested transactions}
\quad \text{forever [}
\quad \quad \text{Read Phase(client id); // reading data objects}
\quad \quad \text{if Validation Phase(client id) = true do}
\quad \quad \quad \text{break; // running new optimistic algorithm}
\quad \quad \text{]}
\quad \text{Write Phase(client id); // update data objects and reply}
\quad \text{]}
\]

At Client Process, $C1$, $C2$, and $C3$

Make_Connection();
Clock_Synchronization();

\[
\text{forever [}
\quad \text{server id} = \text{Choose Server(); // choosing a server at random}
\quad \text{Send Request(server id); // requesting a transaction}
\quad \text{Receive Reply(server id); // receiving reply from a server}
\quad \text{Wait(random); // pause for a random time}
\quad \text{]}
\]
Sample Queries:

Q1 11111
Q2 11112
Q3 11113 470
Q4 11114 saving 580 Bob 9353_Highland_#9453 (909)493-2392
Q5 11115

Sample Data Table:

<table>
<thead>
<tr>
<th>ACCT#</th>
<th>S/C</th>
<th>Bal</th>
<th>Name</th>
<th>Address</th>
<th>Phone#</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111</td>
<td>saving</td>
<td>770</td>
<td>Marks</td>
<td>3421_Kendall_#3492</td>
<td>(909)302-4023</td>
</tr>
<tr>
<td>11112</td>
<td>check</td>
<td>490</td>
<td>David</td>
<td>3425_Kendall_#2449</td>
<td>(909)342-4532</td>
</tr>
<tr>
<td>11113</td>
<td>check</td>
<td>924</td>
<td>Paul</td>
<td>4953_Waterman_#4923</td>
<td>(909)342-4923</td>
</tr>
<tr>
<td>11114</td>
<td>saving</td>
<td>1039</td>
<td>Kevin</td>
<td>3404_Beach_#9424</td>
<td>(714)345-4592</td>
</tr>
<tr>
<td>11115</td>
<td>saving</td>
<td>953</td>
<td>John</td>
<td>4953_Western_#4534</td>
<td>(213)449-4592</td>
</tr>
</tbody>
</table>
APPENDIX B. MAINTENANCE

These servers and clients implementations are located in aviion.

Server #1 : aviion.csci.csusb.edu /u/grad/mpark/server1/server1.c
Server #2 : aviion.csci.csusb.edu /u/grad/mpark/server2/server2.c
Client #1 : aviion.csci.csusb.edu /u/grad/mpark/client1/client1.c
Client #2 : aviion.csci.csusb.edu /u/grad/mpark/client2/client2.c
Client #3 : aviion.csci.csusb.edu /u/grad/mpark/client3/client3.c

The files server1.c and server2.c are compiled by cc -o output source. The files client1.c, client2.c, and client3.c are compiled by cc -o output source -lm. At the start of execution, it asks hostnames and port numbers for distributed machines. Then, the user of this program has to input a λ value raging from 0.1 to 0.9 in each client machine. The servers keep processing the requested transactions until the total number of transactions reaches 1,000.
REFERENCES


