

**PERFORMANCE COMPARISON OF THREE TREE -BASED MUTUAL EXCLUSION
ALGORITHMS IN A SIMULATED DISTRIBUTED COMPUTING ENVIRONMENT**

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment of
the Requirements

for the Degree

Master of SCIENCE

by

Yunhong Jiang, B.A.

San Marcos, Texas

May 2004

COPYRIGHT

By

Yunhong Jiang

2004

Acknowledgments

First, I deeply appreciate my thesis advisor Dr. Furman Haddix for devoting so much time and effort to guide me to accomplish my thesis. Without his encouragement and support, this thesis would not be finished. Also, I would like to thank Dr. Jawad Drissi and Dr. Xiao Chen for their kindness and support on my thesis work. Finally, I want to thank my family for their love and support to me.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	ix
ABSTRACT	x
 CHAPTER 1 INTRODUCTION	
1.1 Motivation	1
1.2 Contribution	2
1.3 Organization	3
 CHAPTER 2 DISTRIBUTED SYSTEM	
2.1 Definition of Distributed Systems	4
2.2 Characteristics of Distributed Systems	4
2.3 Advantage of Distributed System	5
 CHAPTER 3 DYNAMIC-TREE READ/WRITE-LOCKS	
3.1 Introduction	7
3.2 Algorithm	10
3.3 Protocol with Application.....	15

CHAPTER 4 DAG-BASED ALGORITHM

4.1 Introduction	17
4.2 Algorithm	20
4.3 Protocol with Application.....	21

CHAPTER 5 TREE-BASED MUTUAL EXCLUSION WITH FAIRNESS

5.1 Introduction	22
5.2 Algorithm	22
5.2.1 Application (Leaf) Algorithm	24
5.2.2 Root Arbiter Algorithm	25
5.2.3 Branch Arbiter Algorithm	26
5.2.4 Algorithm Details.....	28
5.3 Protocol with Application.....	31

CHAPTER 6 TOKEN-BASED ALGORITHM IMPLEMENTATION

6.1 Implementation Overview	32
6.1.1 File View.....	32
6.1.2 Class View	33
6.2 Implementation Details.....	34
6.2.1 Token Server.....	34
6.2.2 Tree Token	35

CHAPTER 7 APPLICATION IMPLEMENTATION

7.1 Application Description	37
7.2 Implementation Overview	39

7.2.1 File View	39
7.2.2 Class View	39
7.3 Implementation Details.....	40
7.3.1 Token Server.....	40
 CHAPTER 8 PERFORMANCE COMPARISON	
8.1 Token Waiting Time Comparison	42
8.1.1 Simulating Three Terminals.....	42
8.1.2 Simulating Four Terminals.....	44
8.2 Total Execution Time Comparison	45
8.2.1 Simulating Three Terminals.....	45
8.2.2 Simulating Four Terminals	47
 Chapter 9 CONCLUSION AND FUTURE WORK	 49
APPENDICES.....	50
REFERENCE.....	109

LIST OF TABLES

Table 5.1 Possible States for Application	25
Table 5.2 Some Example States of the Root Arbiter with Tow Children	26
Table 8.1 Processes Average Waiting Time.....	43
Table 8.2 Processes Average Waiting Time.....	45
Table 8.3 Processes Total Execution Time	46
Table 8.4 Processes Total Execution Time	47

LIST OF FIGURES

Figure 3.1 Simple Example	9
Figure 4.1 Comparison of Raymond and DAG-based Algorithm	17
Figure 4.2 Topology.....	18
Figure 4.3 Example	19
Figure 5.1 Arbiters with Shared Requests	23
Figure 5.2 State Transition Diagram for Grant.....	24
Figure 5.3 State Transition Diagram for Arbiter Request.....	27
Figure 5.4 Variable Definition	28
Figure 5.5 Application Algorithm	28
Figure 5.6 Root Arbiter Algorithm.....	29
Figure 5.7 Branch Arbiter Algorithm	30
Figure 7.1 Examples of Metrics	38
Figure 7.2 Example of the contents in each matrix.....	38
Figure 7.3 Server and Client Hosts	40
Figure 8.1 Token Average Waiting Time Comparisons	43
Figure 8.2 Total Execution Time Comparisons	46

ABSTRACT

PERFORMANCE COMPARISON OF THREE TREE-BASED MUTUAL EXCLUSION ALGORITHMS IN A SIMULATED DISTRIBUTED COMPUTING ENVIRONMENT

By

Yunhong Jiang, B.A.

Texas State University-San Marcos

May 2004

SUPERVISING PROFESSOR: FURMAN HADDIX

In a distributed system, the design of a mutual exclusion algorithm consists of defining the protocols used to coordinate access to a shared object. A distributed algorithm for mutual exclusion is characterized by (1) all processes have an equal amount of information; (2) all processes make a decision based on local information. Many distributed algorithms for mutual exclusion have been proposed, but this thesis is only concerned with the token-based algorithms which involve lower communication traffic overhead than non-token-based algorithms.

This thesis implements three token-based algorithms and compares their performances in terms of the average token delivery time and the total execution time in a distributed computing environment in a simulated application.

CHAPTER 1 INTRODUCTION

1.1 Motivation

The mutual exclusion problem was originally considered in centralized systems for the synchronization of exclusive access to the shared resource. In the problem of mutual exclusion, concurrent access to a shared resource or the critical section (CS) must be synchronized such that at any time only one process can access the CS.

Over the past decade, many algorithms have been proposed to achieve mutual exclusion in distributed systems. These algorithms can be divided into two classes: token-based and non-token-based (or permission-based). In token-based algorithms, only the site holding the token can execute the critical section and make the final decision on the next site to enter the critical section. In non-token-based algorithms, a requesting site can execute the critical section only after it has received permission from each member of a set of sites in the system – each site receiving a critical section request message participates in making the final decision.

In token-based algorithms, sites are woven into a logical configuration. These algorithms can be static or dynamic. In static algorithms, the logical structure remains unchanged although the direction of the edges may change.

For example, in the DAG-based algorithm, sites are usually organized in a special configuration. Requests sequentially propagate through the paths

between the requesting site and the site holding the token, and so does the token. In dynamic algorithms, a dynamic logical tree is maintained such that the root is always the site which will hold the token in the near future (i.e., the root is the last site to get the token among the current requesting sites) when no message is transit. The token is directly sent to the next requesting site to execute the CS, but a request is sequentially forwarded along a virtual path to the root.

As mentioned above, many problems require that a shared object be allocated to a number of requesting processes in mutually exclusive manner. Hence, the mutual exclusion problem plays a vital role in the design of distributed systems.

There is a trade-off between synchronization delay and message complexity of distributed mutual exclusion algorithms. No single mutual exclusion algorithm can optimize both synchronization delay and the message complexity. The purpose of the thesis is to compare three different token-based algorithms, namely, dynamic-tree read/write locks protocol [WaMu00], DAG-based protocol [NeMi91] and fairness tree-based protocol [Hadd04]. Some researches have analyzed the first two protocols, but there is no comparison work among these three protocols.

1.2 Contribution

Contributions of this work include the following:

- Simulation of multiple processors in a local distributed computing environment
- Design of a method to create a logical tree of processes

- Development of a distributed server application as bench mark
- Implementation of the three token-based algorithms for mutual exclusion between reading and writing processes
- Evaluation of the performance of the three token-based algorithms based on waiting time and execution time

1.3 Organization

Chapter 1 of this thesis is an overall introduction to the thesis work and the purpose of creating this thesis. Chapter 2 talks about distributed systems generally. Chapters 3 through 5 describe the three token-based algorithms. Chapter 6 describes how the three token-based algorithms which introduced in the previous 3 chapters are implemented. Chapter 7 describes the application implementation that is used to assist the simulation of three algorithms. Chapter 8 describes the measurement of performance of the three algorithms and analyzes the results. Chapter 9 discusses the conclusions and possible future directions of this thesis work.

CHAPTER 2 DISTRIBUTED SYSTEMS

2.1 Definition of Distributed Systems

In general, distributed system is a collection of (possibly heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to the network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent. Distributed systems often use a client-server organization

2.2 Characteristics of Distributed Systems

Some of the characteristics of distributed systems and their rationales are the following:

1. Resource sharing allows sharing of hardware devices for convenience and to reduce costs, and Sharing of data objects in cooperative working environments.
2. Openness facilitates hardware extensibility, including the addition of processors, peripherals, memory; and software extensibility, including the addition of features and protocols.
3. Concurrency allows several processes to be executed of the same time in a distributed system.

4. Scalability facilitates addition of system components without modification of system and application software and the avoidance of potential bottlenecks as a system grows.
5. Fault Tolerance enables hardware redundancy, such as standby machines which keep checking the system and are always ready to replace the system when it fails and software recovery, such as check points and rollback.
6. Transparency, including access transparency, such as local and remote user using identical operations; local transparency, such as information objects being accessed without knowledge of their location; concurrency transparency, such as several processes operating at the same time and using the same information object but without interference between processes; failure transparency, such as tasks being completed in the presence of failure of hardware or software; migration transparency, such as moving objects within the system without affecting users and application programs; performance transparency, such as system reconfiguring for improved performance as load varies and scaling transparency, such as system expansion without change in system structure.

2.3 Advantages of Distributed Systems

Some of the advantages of distributed systems are the following:

1. Remote access, such as user retrieval of their data when not at their

usual location.

2. The equipment cost reduction due to distributed systems having fewer redundant resources than standalone installation.
3. Flexibility and configurability, for example, distributed systems include features to improve performance and reliability. (e.g. redundant data)
4. Availability of new applications for use in distributed system environments.

CHAPTER 3 DYNAMIC-TREE READ/WRITE-LOCKS

3.1 Introduction

The Dynamic-tree Read/Write-locks protocol was developed by Claus Wagner and Frank Mueller [WaMu00]. This protocol uses a token-based decentralized approach, which allows either multiple concurrent readers or a single writer to enter their critical sections. It utilizes a dynamic structure incorporating path compression to keep the message overhead low resulting in an average complexity of $O(\log n)$ messages per request.

It utilizes a directed tree-like structure. The edges form a chain leading new requests from node R to the last requester L (or the token holder if no requests were pending). While a request is in transit, intermediate nodes set their edges to point to the new requester R, thereby providing path compression, i.e., future requests will propagate directly to R from any of the intermediate nodes.

An example is depicted in Figure 1 where a (read) request from A is sent via B to T. The intermediate nodes B and T have edges directed at A afterwards. If T is still engaged in the critical section with a write lock, the read request cannot be served yet. Instead, T creates a next pointer (dashed edge) to A indicating the next recipient of the token. Once T exits its critical section, it sends the token to A and removes the next edge. A proceeds with its critical section under read protection. At this time, C issues a read request that is sent to A via B resulting in new edges from A and B to C. A responds by sending the token to C

even before exiting its critical section because both A and C may execute their critical sections concurrently. In addition, A registers C as the next reader (double-line edge) and C notes the fact that it is the last reader. At the same time before C exits its critical section, Node B send a write request. C registers B as the next requester. Since C hasn't got the token yet, B must wait until C exits its critical section. Then node D send a read request through T and A to B. Even though B is still waiting for the token, it registers D as the next requester. Once the token has been passed to the D, it will hold on to the token until (1) a request from another reader arrives next or (2) all the requesters exit their critical sections. The former case allows more concurrent readers to be served if they arrive ahead of writers. The latter case ensures that a writer will only be served after all the concurrent reads have completed. Once one node exits its critical section, it sends an acknowledgement to the next it points to, which must be received by the next node before the token may be forwarded to a writer.

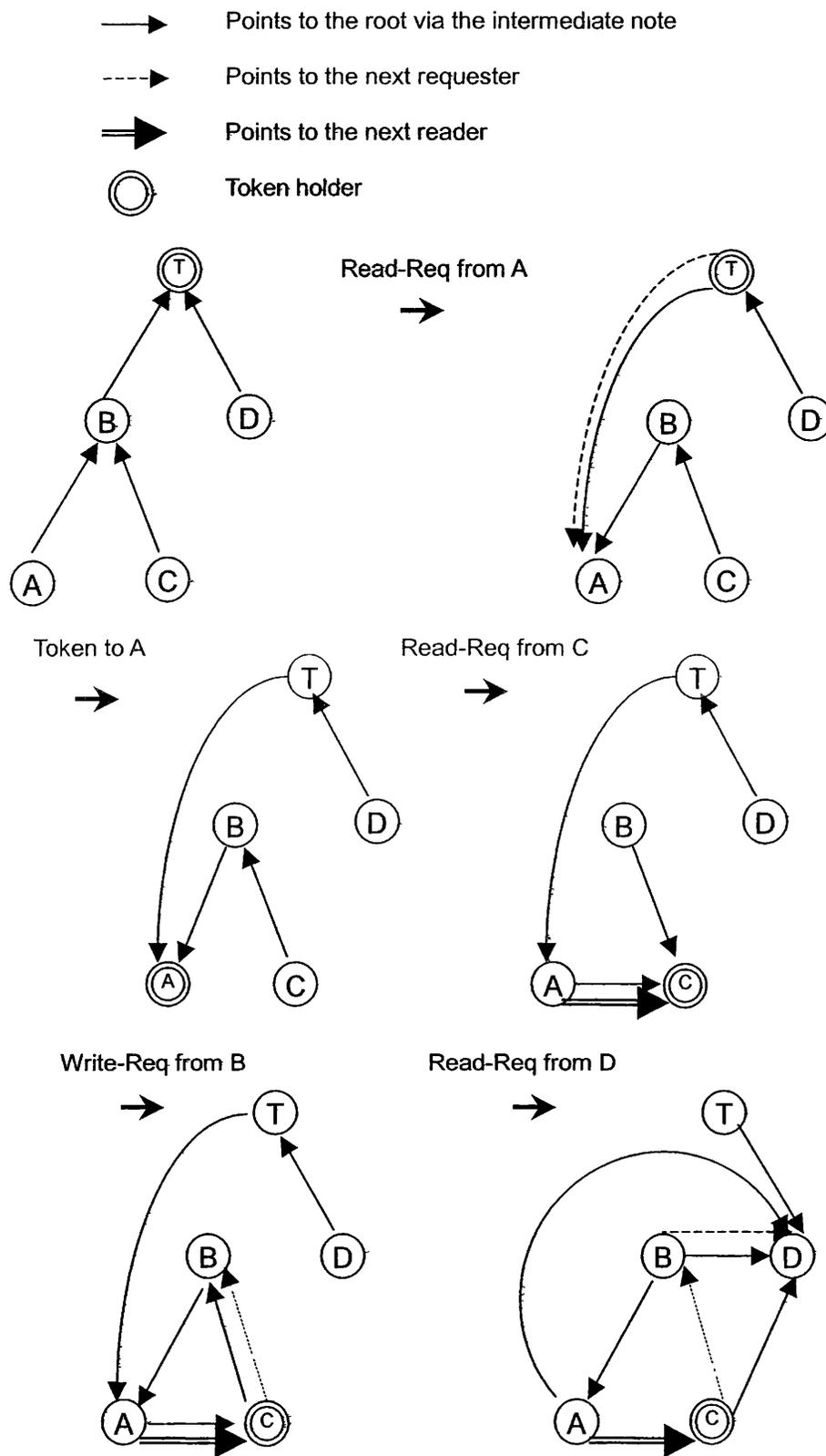


Figure 3.1: Simple example

3.2 Algorithm

The protocol requires directed edges that requests travel along. It includes a distributed next queue of pending requests whose source is the token holder while the sink is the last requester. A distributed read queue links concurrent readers starting at the earliest requester still engaged in a critical section via consecutive requesters to the sink of the reader chain. Consecutive members of the read chain are either in the critical section or have not received an acknowledgement from their predecessor in the read chain yet. When a node issues multiple read requests in a row, the read chain may be circular imposing the necessity to log pending recipients of acknowledgements in a local FIFO queue. Requests are always handled in the order that they arrive at the tail of the request queue, i.e., a write request always terminates a chain of readers and subsequent read requests are served after the write. This ensures fairness and avoids starvation.

This thesis work makes some enhancements based on the original Token-Based Read/Write-Locks protocol. In the original protocol, a queue structure is used to store the next requester and next reader. But in the enhanced version of this thesis work, only pointers point to the next requester and next reader instead of the queue. It is not necessary for every requester to keep the information on all the later requesters. To make the algorithm more efficient, every requester only has the next requester and next reader pointers to identify the next member to pass the token to. In this case, only the token is sent to the next requester or next

reader instead of sending both the next requester queue and the token. The following variables are used in the protocol:

```

Boolean token = (self == start); // TRUE for current holders of the token if no process is
                                engage within critical section
Identifier dir = start;          // pointer with changing destination for requests
                                propagation. It points to the last requester or, in the
                                absences of requests, to the token holder, creating a
                                distributed tree
Identifier next = NULL,         // points to the next requester of the token in the queue of
                                requests. The next chain represents a distributed queue
                                of pending requests. next receiver of token (=> dist. Q
                                of unserved requesters)
Identifier next_readers = NULL, // points to the next reader in a sequence of concurrent
                                readers. The next readers over all nodes form a
                                chain of concurrent readers that will be reduced
                                successively by acknowledgements
Boolean pending_acks = TRUE;   // true indicates that the current node is the next
                                reader for another node, false means that all
                                subordinate concurrent readers are finished or
                                were existed.
Enumeration token_mode = UNDEF; // protection of critical section
                                (undef/read/multiread/write). WRITE means that a
                                writer is engaged in a critical section. READ if the
                                last reader (of the reader chain) is in a critical
                                section or MULTIREAD if the token has already
                                been sent to a concurrent readers node. UNDEF in
                                any other case.
Enumeration next_mode = UNDEF; // protection of next receiver (undef/read/write). It is
                                READ if next points to the next read requester or, it
                                is WRITE if next points to the next write requester.

```

Pseudo code:

```

PROC lock(mode) IS
  IF ¬ token THEN
    SEND request (self, mode) to dir;
    dir = self;
    AWAIT (token (&expect_ack));
    IF expect_ack THEN
      pending_acks = TRUE;
    ENDIF;
    concurrent_read = (mode == READ AND next_mode == READ);
    IF next ≠ NULL AND concurrent_read THEN
      SEND token (TRUE) to next;
      next_readers = next;
      next = NULL;
      token_mode = MULTIREAD;
    ELSE
      token_mode = mode;
    ENDIF;
  ELSE
    token_mode = mode;
  ENDIF;
  token = FALSE;
END lock;

```

At initialization, all nodes point via *dir* to the start node, which is the token holder. Notice that the edge of the token holder to himself is omitted in the example since it represents a special case beyond the tree structure. A lock request for a locally unused token can be served right away. All other requests propagate along the *dir* edges while the requester clears his *dir* pointer since he is the last requester waiting for the token to arrive. Once the token arrives, the

flag to indicate if the node should “expect an acknowledgement” is set according to the value piggybacked with the token message and the status of pending acknowledgements may be changed accordingly. In case of concurrent reads, the token is forwarded to the next reader and the mode is set to *MULTIREAD*. Otherwise, the request mode is stored before entering the critical section. The token flag is also reset during the critical section indicating that it is in use.

```

PROC unlock IS
  IF token_mode == multiread AND pending_acks == FALSE THEN
    SEND ack to next_readers;
    next_readers = NULL;
  ENDIF,
  IF next ≠ NULL AND pending_acks == FALSE THEN
    SEND token(FALSE) to next,
    next = NULL;
  ELSE IF token_mode ≠ MULTIREAD THEN
    token = TRUE;
  ENDIF;
  token_mode = UNDEF;
END unlock;

```

Upon the end of the critical section (unlock), several cases are distinguished. The first reader sends an acknowledgement to the next reader, thereby changing the value of acknowledgement if they have already received an acknowledgement from their predecessor. If a next requester exists and all read requests have completed (no pending acknowledgements), the token is sent to the next requester. Otherwise, the token is marked as locally available unless it

was already sent to a concurrent reader at an earlier time.

```

PROC receive_request (sender, mode) IS
  IF dir ≠ self THEN
    SEND request (sender, mode) to dir;
  ELSE
    Concurrent_read = (token_mode == READ AND mode == READ);
    IF (token AND pending_acks == FALSE) OR concurrent_read THEN
      SEND token(¬ token) to sender;
      token = FALSE;
      IF concurrent_read THEN
        token_mode = MULTIREAD;
        next_readers = sender;
      ENDIF;
    ELSE
      next = sender;
      token = FALSE;
      next_mode = mode;
    ENDIF;
  dir = sender;
END receive_request;

```

A receiver of a request from some sender also has to distinguish certain cases before changing its *dir* edge to the sender. If the receiver is not the last requester, then he simply forwards the request with the sender's id along the directed edges. Otherwise, one of two cases may apply. If the token is available and all readers have completed (no acknowledges pending) or if the request is for a concurrent read, the receiver sends the token to the sender, clears the token status and records the next concurrent reader, if necessary. The

piggybacked flag is true if the new request represent a concurrent read. If neither the token was available nor was the request for a concurrent read, the sender is logged as the next requester.

```

PROC receive_ack IS
    pending_ack == FALSE;
    IF token_mode == UNDEF THEN
        SEND ack to next_readers;
        next_readers = NULL;
    ELSE IF token AND next ≠ NULL THEN
        SEND token (FALSE) to next;
        next = NULL;
        token = FALSE;
    ENDIF;
END receive_ack,

```

Upon receipt of an acknowledgement, the status of pending acknowledgements is changed and an acknowledgement is sent to the next reader if this had not already been done in the unlock operation. Otherwise, the token is forwarded to the next requester if all acknowledgements have been sent and the next reader points to *NULL*. Notice that the next requester was either a writer or a reader at the first read requester so that the piggybacked value *FALSE* requires no checks for acknowledgements by this requester (similar to unlock).

3.3 Protocol with Application

In the application, three 16 by 16 matrices, A, B and C are used. Once the process gets the token, it will communicate with the server. The reader, reads

matrices A, B and C from the server, and print out what they read. According to the algorithm, all the readers will concurrently read the source from the server, then save it to their local matrices. The writers read matrices A and B, multiply A and B, output the result to matrix C, and write it back to the server.

CHAPTER 4 DAG-BASED ALGORITHM

4.1 Introduction

This protocol is based on Neilsen and Mizuno [NeMi91]. Instead of passing the token step by step through intermediate sites in the logic structure to the token requestor as in the Raymond [Raym89] algorithm, Neilsen and Mizuno proposed an algorithm where the token holder can send the token directly to the requesting site with one message. This is made possible by attaching the requestor's ID in the request message so that the token holder knows, on receiving the message, who is the requestor.

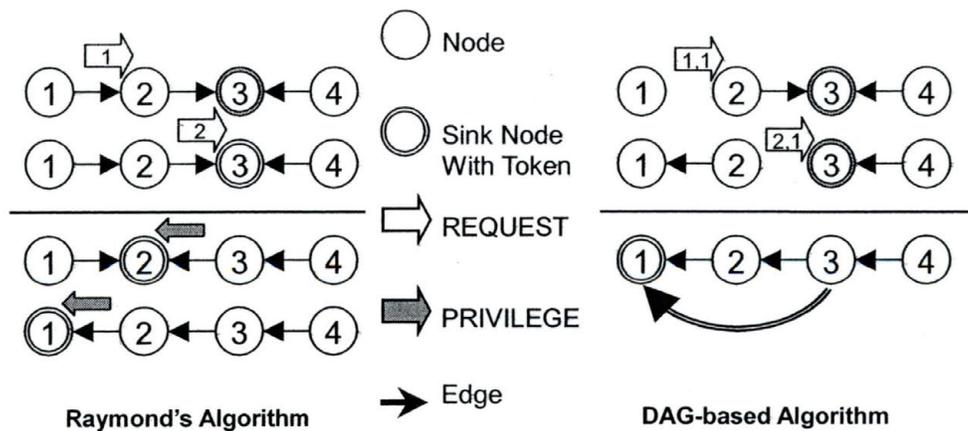


Figure 4.1: Comparison of Raymond and DAG-based Algorithm

One special case of this algorithm is that the logical structure can be a fixed star topology (called the Star algorithm), which means it is a fully connected,

reliable physical network.

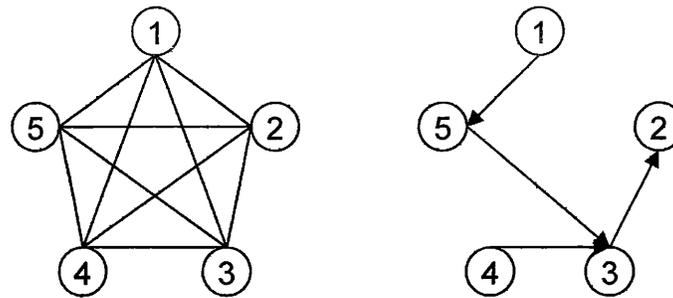


Figure 4.2: Topology

Under such situation, any site ready to enter the critical section always sends a request message attached with its own ID directly to the root node. The root node make it possible to establish a distributed waiting queue (of all requesting sites) by recording the site which has most recently requested the token (and is the tail site in the distributed waiting queue). When receiving a request message, the root forwards the message to the tail site (of the queue) and updates its record, unless the root itself holds the token. On receiving a request message, the token holder, if not in need of the token, forwards the privilege to the requestor directly using a token message. A very attractive property of the Start algorithm is that it always takes three exchange messages for a requestor to get the token, if the root does not own the token and only two messages if the root holds the token.

An example is depicted in Figure 4 illustrating how the root keeps checking the requesters. Before anyone sends request, the tree is rooted at Node 1. Then the first requester is Node 3. So Node 1 registers Node 3 as the next requester and the last requester since there are no more requesters for the token. Node 3 points to its parent which is the root (Node 1). Once Node 3 has

the token, it accesses its critical section. At the same time, Node 2 sends a requester, and both of Node 1 and Node 3 point to the Node 2. Node 1 registers Node 2 as the last requester instead of Node 3. Node 3 set its next pointer and last pointer to the Node 2. And Node 2 sets its parent as node 3. Before Node 2 get the token, Node 4 sends a request, all the node 1, 2 and 3 register Node 4 as the last requester. Node 2 sets its next requester pointer to Node 4 to indicate that Node 4 is its next requester. Once there is a new requester coming, the requested node must set its last pointer to the requesting node indicating that it is the last requester.

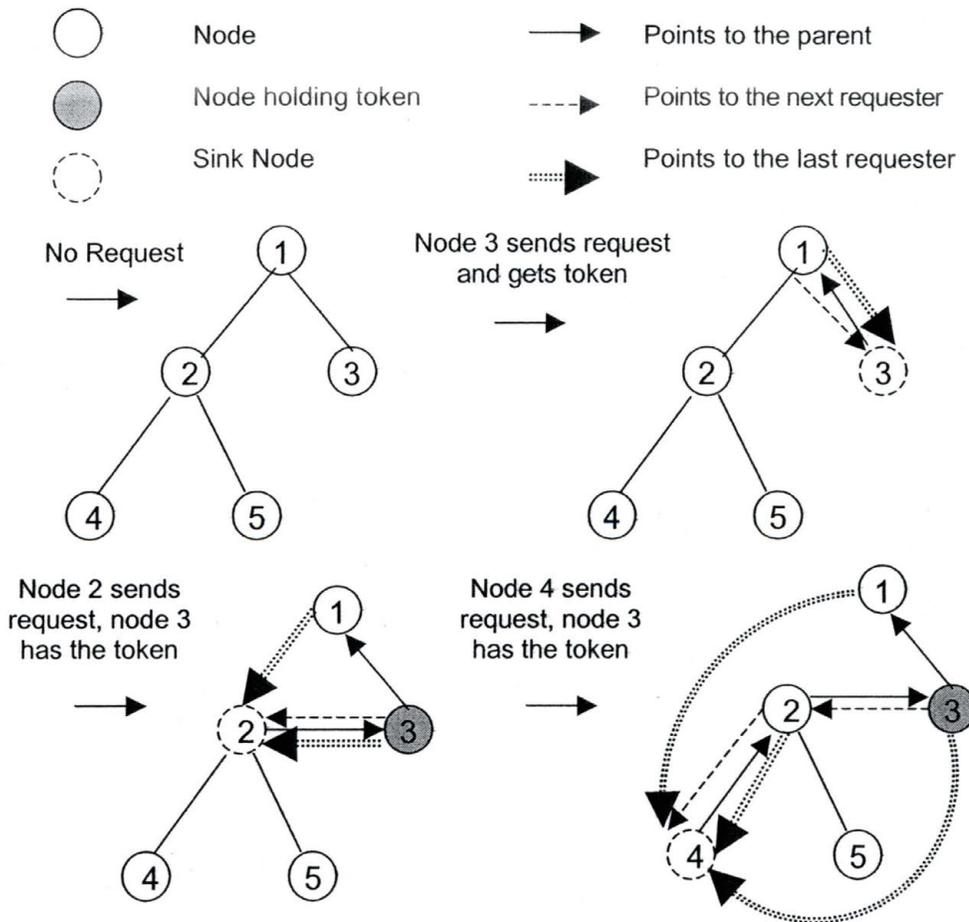


Figure 4.3: Example

4.2 Algorithm

This protocol uses a directed acyclic graph (DAG) structured logical network with one sink. The following variables are used in describing the protocol:

REQUEST: The message a node initiates when it needs to enter the critical section.

PRIVILEGE: The message which grants the node receiving it the privilege to enter the critical section, that is, the token.

LAST: Points toward the tail of the queue.

NEXT: Indicating the next node which will be granted mutual exclusion after this node, and enabling deduction of the implicit waiting queue of the distributed system

HOLDING: Indicating whether this node has the available token or not. It is false if this node doesn't have the token or this node is in its critical section.

Algorithm:

1. When a node needs to enter critical section, it initiates a new REQUEST message

- o Sending REQUEST(I,I) message to the node indicated by LAST where the first I is the id of the node sending the message and the second I is the id of the node initiating the request
- o Being a new sink node, that is, setting LAST = 0 since it is the tail of the queue of requesters

2. When a non-sink node N receives a REQUEST(B,I) message from node B

- o Passing REQUEST message to the node indicated by LAST

- ⚡ When a sink node S receives a REQUEST(N, I) message from node N
 - If HOLDING is false, setting NEXT = the id of the node initiating the request, that is, I
 - If HOLDING is true, forwarding the PRIVILEGE message to the node initiating the request, that is, I

4.3 Protocol with Application

When combined with the application, the protocol provides exclusive access to the critical section. Once a process is accessing its critical section, all the other requester will wait for it to finish the reading or writing. In the application, three matrices (A, B and C) are used. Each process will read all three matrices one by one, and print out what it reads for the *read* process. The writer then multiples matrices A and B, and outputs the results back to the server to rewrite the matrix C.

CHAPTER 5 TREE-BASED MUTUAL EXCLUSION WITH FAIRNESS

5.1 Introduction

This protocol is due to Haddix [Hadd04]. It utilizes a dynamic grant tree, which is a directed acyclic graph where grants flow down the tree (leafward, meaning from the root to the leaves) and requests flow up the tree (rootward, meaning from the leaves to the root).

In this protocol, each leaf represents an application using the grant, each branch and the root node are arbiters granting the privilege in a distributed fashion. An arbiter with an unassigned grant passes it to each child where child may be an arbiter or an application. The root always has a grant in. There are no grants active during arbitration and when there are no outstanding requests.

5.2 Algorithm

The Figure 5.1 shows the arbiters' status when there are shared requests. Once the root arbiter has the shared grant, all the branch arbiters with shared requests can have the shared grant, which allows more concurrency.

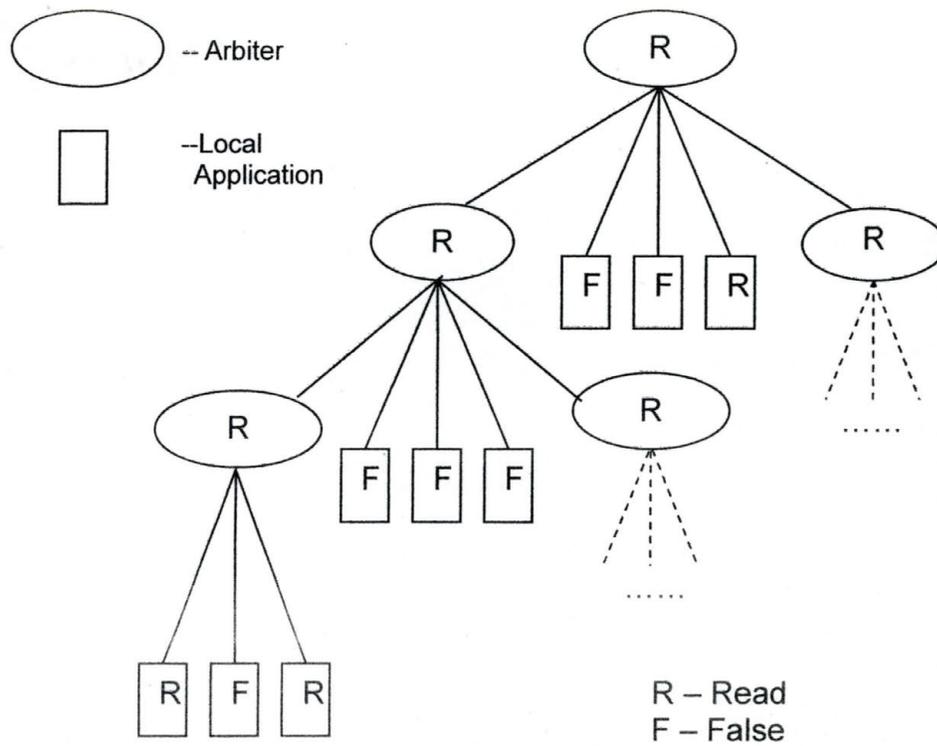


Figure 5.1: Arbiters with Shared Requests

This protocol is implemented by three algorithms described in sections 5.2.1, 5.2.2, and 5.2.3. The relationships between the three values for $\text{grant}[i]$ are illustrated in Figure 5.2. The three cases are the following:

1. The request is false if no arbiter has a READ, WRITE, or BOTH request, i.e., $\forall i, i \in \text{children}, \text{request}[i] = F$ ($0 < i < M + N$).
2. If there exists a child with a READ or BOTH request, then the grant of that arbiter is set to READ, if there is a READ grant in.
3. If there exists a child with WRITE or BOTH request, then the grant of that

arbiter is set to WRITE, if there is a WRITE grant in.

The difference between the types of grant is that READ grants are issued concurrently while WRITE grants are issued sequentially. The cases are slightly different for the root and branch arbiters.

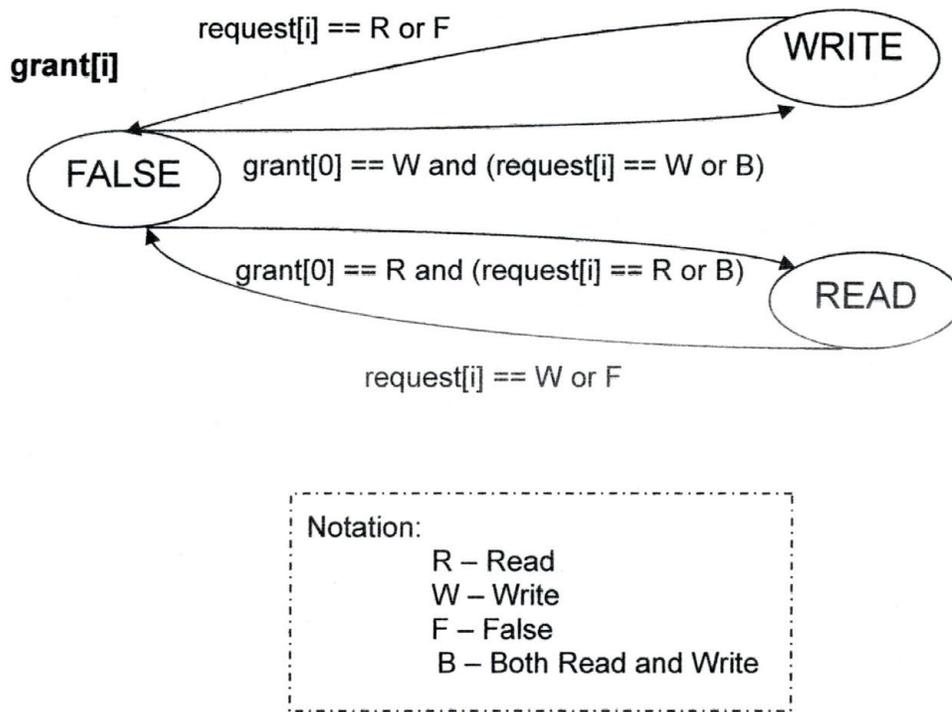


Figure 5.2: State Transition Diagram for Grant

5.2.1 Application (Leaf) Algorithm

For the application, this algorithm can be viewed as an upward-looking grant and request algorithm. The application can read the grant variable and can write the request variable. Table 5.1 shows the possible states for the application.

Request	Grant	INTERPRETATION
FALSE	FALSE	NO INTEREST
FALSE	READ	RELEASES READ LOCK
FALSE	WRITE	RELEASES WRITE LOCK
READ	FALSE	WAITING FOR READ
READ	READ	HOLDING READ LOCK
READ	WRITE	ILLEGAL
WRITE	FALSE	WAITING FOR WRITE
WRITE	READ	ILLEGAL
WRITE	WRITE	HOLDING WRITE LOCK

Table 5.1: Possible States for Application

Since the application control the request variable, it controls the timing of request and release actions. By controlling the grant variable the application's parent arbiter controls the grant and accepts release actions.

5.2.2 Root Arbiter Algorithm

The root arbiter acts as a central privilege arbiter. It is agnostic as to whatever a child is an application (leaf) or arbiter (branch). In effect, the root always has an exclusive grant. It can pass that grant through as a shared or exclusive grant to a child or children.

The root operates in two consecutive modes, exclusive and shared. In the exclusive mode, it successively polls each child for WRITE requests (WRITE or BOTH), issuing grants serially. In the shared mode, it polls each child for READ

requests (READ or BOTH), issuing concurrent grants. When all concurrent grants are released, it returns to the exclusive mode.

Some states of the root arbiter are shown in Table 5.2.

Request(A)	Request(B)	Grant(A)	Grant(B)	INTERPRETATION
FALSE	FALSE	FALSE	FALSE	No Interest
FALSE	READ	FALSE	READ	B holds READ
FALSE	WRITE	FALSE	WRITE	B holds WRITE
FALSE	BOTH	FALSE	READ	B holds READ and needs write
FALSE	BOTH	FALSE	WRITE	B holds WRITE and needs READ
READ	FALSE	READ	FALSE	A holds Read
READ	READ	READ	READ	Both A and B hold READ
READ	WRITE	FALSE	WRITE	B holds WRITE and A needs READ
READ	WRITE	READ	FALSE	A holds READ and B needs WRITE
READ	BOTH	FALSE	WRITE	B holds WRITE and both need READ
READ	BOTH	READ	READ	Both hold READ and B needs WRITE
BOTH	BOTH	FALSE	WRITE	B holds WRITE, both need READ and A needs WRITE
BOTH	BOTH	READ	READ	Both hold READ and both need WRITE
BOTH	BOTH	WRITE	FALSE	A holds WRITE, B want WRITE and both need READ

Table 5.2: Some Example States of the Root Arbiter with Two Children

5.2.3 Branch Arbiter Algorithm

In terms of issuing grants, branch arbiter differs from the root arbiter only in that its mode is determined by its grant[0] variable (indicating grant issued by parent) rather than cyclically as with the root.

The determination of the arbiter request [0] variable is according to the union of the requests of all its children as depicted in Figure 5.3. Thus, request[0] = FALSE if a node's children make no requests; request [0] = READ if some of its children have READ requests; request [0] = WRITE if some of its children have WRITE requests; and request [0] = BOTH, if its children have both READ and WRITE requests or if one or more have BOTH requests.

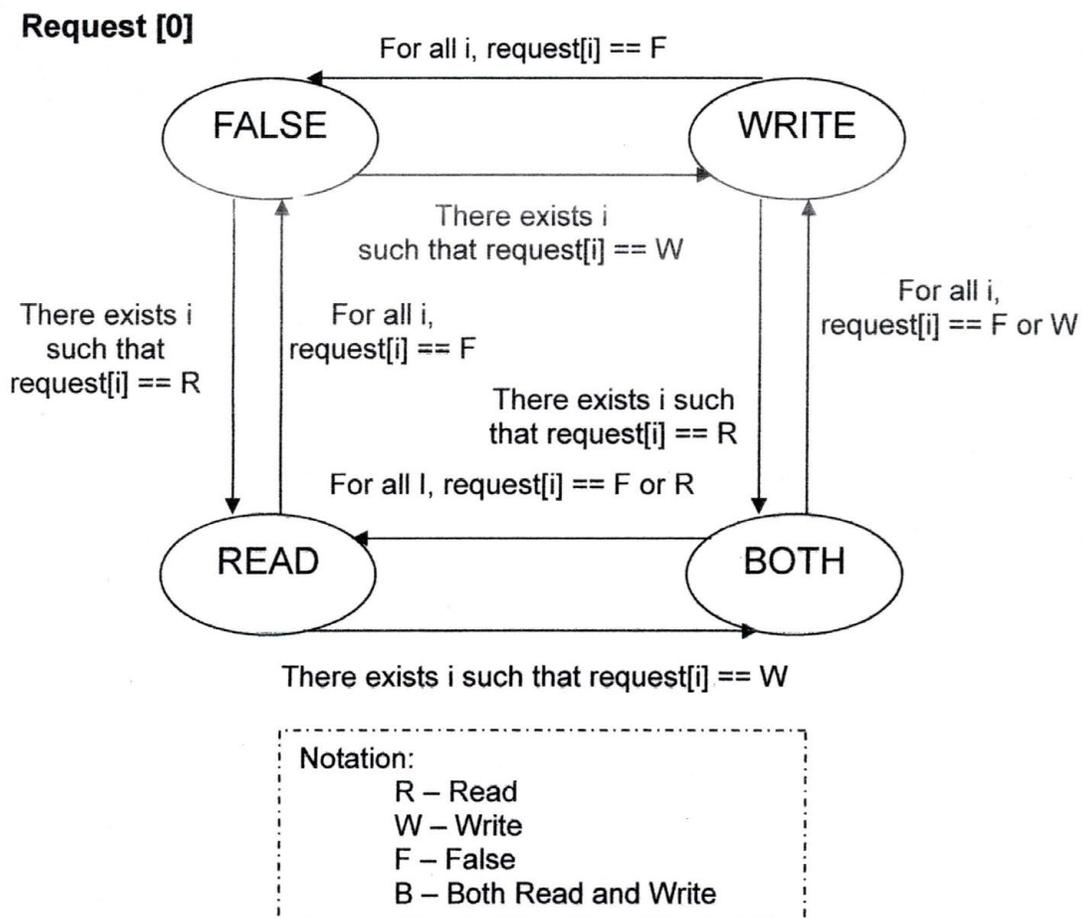


Figure 5.3: State Transition Diagram for Arbiter Request

5.2.4 Algorithm Details

The algorithm is shown in Figure 5.4, 5.5, 5.6, and 5. 7.

```

Identifier neighbor []           // neighbor [0] = par, null if root
Identifier local_app[]          // set of local applications
N = cardinality of neighbor     // the number of child arbiters + 1
M = cardinality of local application
Variables: Enum grant [N + M]   // {R, W, F}
        Enum request [N + M]    // {R, W, B, F}
        Identifier curr {0, 1, ..., N + M + 1}

```

Figure 5.4: Variable Definition

```

Exclusive access to critical section
    request[i] = W;
    while(grant[i] ≠ W) {}
    //critical section
    request[i] = F;
Shared access to critical section:
    request[i] = R;
    while(grant[i] ≠ R) {}
    //critical section
    request[i] = F;

```

Figure 5.5: Application Algorithm

```

while(true)
if(curr < N + M) {
    if((request[curr] == W ∨ request[curr] == B) ∧ grant[curr] ≠ W)
        grant[curr] = W,
    if(request[curr] ≠ W ∧ request[curr] ≠ B ∧ grant[curr] ≠ F)
        grant[curr] == F;
    if(request[curr] ≠ W ∧ request[curr] ≠ B ∧ grant[curr] == F)
        curr ++;
}
else{
    for(i = 1, i < N + M, i ++ )
        if((request[i] == R ∨ request[i] == B) ∧ grant[i] ≠ R) { }
        grant[i] == R,
    for(I = 1; i < N + M, I ++ ) {
        if((request[i] == R ∨ request[i] == B) ∧ grant[i] == R) { }
        while((request[i] == R ∨ request[i] == B ∧ grant[i] == R) { }
        grant[i] == F;
    }
    curr = 1;
}
}

```

Figure 5.6: Root Arbiter Algorithm

```

while(true) {
    if(grant[0] == F)
        for(i=1, i < N + M, i++) {
            if(request[i] == R ∧ request[0] == W)
                request[0] = B;
            if(request[i] == R ∧ request[0] == F)
                request[0] = R;
            if(request[i] == W ∧ request[0] == W)
                request[0] = B;
            if(request[i] == W ∧ request[0] == F)
                request[0] = W;
        }
    if(grant[0] == W ∧ (request[0] == W ∨ request[0] == B)) {
        curr = 0;
        while (curr < M + N) {
            X) if((request[curr] == X ∨ request[curr] == B) ∧ grant[curr] ≠
                grant[curr] = W,
                if(request[curr] ≠ W ∧ request[curr] ≠ B ∧ grant[curr] ≠ F)
                    grant[curr] = F;
                if(request[curr] ≠ W ∧ request[curr] ≠ B ∧ grant[curr] == F)
                    curr ++,
            }
            if(request[0] == B)
                request[0] = R;
            if(request[0] == W)
            }
        if(grant[0] == R ∧ (request[0] == R ∨ request[0] == B)) {
            request[0] = F;
            for(i = 1; i < N+M; i++)
                if((request[i] == R ∨ request[i] == B) ∧ grant[i] ≠ R)
                    grant[i] = R;
            for(i=1; i < N + M; i++) {
                if((request[i] == R ∨ request[i] == B) ∧ grant[i] ==
                    R) {}
                while((request[i] == R ∨ request[i] == B) ∧ grant[i] == R)
            {}
                grant[i] = F;
            }
            if(request[0] == B)
                request[0] = W;
            if(request[0] == R)
                request[0] = F;
        }
    }
}

```

Figure 5.7: Branch Arbiter Algorithm

5.3 Protocol with Application

When it is combined with the application, three matrices are used. Only the process with the token can access the critical section which means can communicate with the server.

In this algorithm, if an arbiter (a computer) gets the grant, it passes the grant to its children with outstanding requests. There are concurrent reads if the token mode is shared, which means all the read processes can access the critical section at the same time. If the token mode is exclusive, then only one process can access to the critical section at a time and all other requesters must wait.

CHAPTER 6 TOKEN-BASED ALGORITHMS IMPLEMENTATION

Java language is used to implement the algorithms. High-level TCP/IP sockets are used for communications.

6.1 Implementation Overview

Three token-based algorithms are implemented in a similar way, and the implementation code for them also has the similar basic structure. The Dynamic-tree algorithm and DAG-based algorithm has the same structure because the queue structure is used for the implementation. Then in the Fairness tree-based algorithm, array is used instead of the queue for the structure. The structure is described in two views, one is from the view of the java files, and the other one is from the view of the classes.

6.1.1 File View

For each algorithm, there are several classes are used to implement the algorithm:

Algorithm.java:

It is implemented with the algorithm. All the algorithms use the same file them in the implementation.

Processes.java

This file holds the process information.

TreeClient.java:

This is a big file. It includes how the processes are used to build the tree; how each process recognize their order to access to the critical section; how the processes communicate with the server to read or overwrite the source.

6.1.2 Class View

For each algorithm, there are several classes to implement them:

Algorithm.class:

This class has the detailed implementation base on the algorithm. For each algorithm, the code in this class is totally different from other algorithms. But they all use the same name.

Processes.class:

This class stores the processes' information. For example, *pAddress* holds the process address information; *portNumber* keeps the port number that a specific process is using.

TreeClient.class:

This class is used to produce the processes which need to access critical sections and puts them into a logical structure based on the protocol.

TokenServer.class:

This class is a server class, which has two different purposes: one is

used to help the simulated processes build the tree structure; the other is to hold the source file (including the code for the critical section) to communicate with the requester processes.

6.2 Implementation Details:

6.2.1 Token Server

TokenServer is the name of the server used to establish a tree in the implementation.

1. This has a method to build a tree in any distributed system. The method described below:
 - The tree will be constructed based on the order in which the processes connect to the server.
 - The first process becomes the root.
 - All the other processes connect as children of processes already connected to the tree. This is related to the *TreeToken* file.
2. This class has the implementation of the critical section and the source (three matrices) the processes want to read or write.
 - Then once the process get the token, it will send the message to the server and ask for getting into the critical section
 - The server will be waiting for the processes to access the CS in the defined order. If no one has the request for the source, the server will be idle.

In this thesis work, multiple processes are simulated using three computers. When 9 processes are used, each computer simulates three processes; when 15 processes are used, each computer simulates five processes.

6.2.2 Tree Token

This class is an important class, which is connected with the processes object and the algorithm. It controls the processes which communicate with the server and access the critical section. It has the following functions:

1. It generates an equal number of processes in each computer, for example, three of each.
2. It assigns the order of the processes in building the structure.
 - The port number is attached to the process when the process is generated, as well as the local address.
 - Each process obtains its own id by receiving a number from the *TokenServer*.
 - Each process generates the mode using a Pseudo Random function (A sequence of numbers generated by some algorithm so as to have an even distribution over some range of values and minimal correlation between successive values).
 - All the requesters will be kept in the queue, and the queue is being updated continually. Once a process finishes execution and leaves the critical section, it will be removed from the queue.

- The processes iterate the algorithm many times between accesses of the critical section, so that when the information is updated, they take turns communicating with the server.
3. It facilitates reading from or writing to the source from the server.
- A process with a read request reads the matrices A, B and C from the server, and stores them in its local matrices.
 - A process with a write request reads the matrices A and B, multiplies matrices A and B, and stores the result in a local matrix. After this, it reads the last matrix C from the server, and then adds the multiplication result with the original data it read together, and sends the updated data back to the server. At the same time, the server puts the new data back to the matrix C.
 - Based on different algorithm, the readers and writers have different priority to be handled. For example, in the Dynamic-tree protocol, only the consecutive readers can read the source at the same time; but in the Fairness tree-bases protocol, if the arbiter has the grant with read, all the readers can read the source at the same time even they might not be consecutive inside the local application.

CHAPTER 7 APPLICATION IMPLEMENTATION

7.1 Application Description

A simple application is used in the implementation of those three algorithms.

Three 16 by 16 matrices are used for the application. These three matrices are stored in a server. Once a process has the token, it will communicate with the server by reading three matrices from them. If that process has the READ mode, it will read the matrices and print them out; if that process has the WRITE mode, it will read three matrices from the server and multiply the matrix A and B then write the result back to the server. And the algorithm will control the mutual exclusion depends on different protocols.

There are four computers are used for the simulation, which are connected locally to keep a distributed computing environment during the test. Three computers are used for the protocols, and the fourth computer is used to set up as a server which has the critical section part in it.

The following is an example of how the multiplication is done. This example uses two 4 by 4 metrics A and B shown in Figure 7.1.

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 10 \end{pmatrix} \quad \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 10 \end{pmatrix}$$

Matrix A Matrix B

Figure 7.1 Examples of Metrics

The multiplication result is saved in the matrix C, and it is calculated by multiplying every row of A with every column of B. The resulting numbers are arranged in a new matrix C: the x-th row in A times the y-th column in B gives the number at position (x, y) in C as showing below:

$$\begin{aligned} C[0,0] &= A[0,0] * B[0,0] + A[0,1] * B[1,0] + A[0,2] * B[2,0] + A[0,3] * B[3,0] \\ &= 1 * 1 + 3 * 2 + 5 * 3 + 7 * 4 = 50 \end{aligned}$$

$$\begin{aligned} C[0,1] &= A[0,0] * B[0,1] + A[0,1] * B[1,1] + A[0,2] * B[2,1] + A[0,3] * B[3,1] \\ &= 1 * 3 + 3 * 4 + 5 * 5 + 7 * 8 = 96 \end{aligned}$$

The following is an example figure which is used in the implementation:

1	3	5	7	...	29	31
2	4	6	8	...	30	32
3	5	7	9	...	31	33
4	6	8	10	...	32	34
...	...					
15	17	19	21	...	43	45
16	18	20	22	...	44	46

Figure 7.2: Example of the contents in each matrix

7.2 Implementation Overview

The application implementation also can be described in two views. One is from the view of the files, and the other one is from the view of the classes.

7.2.1 File View

Matrix.java

It is a utility file to create and initial three matrices which are hold by the server and are updated by the clients.

TokenServer.java

In the description of the algorithms' implementation in the previous chapters, this file has been mentioned already. Since it includes the application part besides the algorithm, the detailed description is in the next section.

7.2.2 Class View

Matrix.java:

This file generates the 16 by 16 matrix. The row numbers start with 1, 2, 3 and so on until it has 16 columns. And the column numbers start with 1, then 3, 5, 7 until it has 16 rows.

TokenServer.class:

It connects with all the clients, handles sending the matrices to the clients respect to the requests and taking updated matrices back from the clients to overwrite the original source.

7.3 Implementation Details

7.3.1 Token Server

This file is the same one used in the implementation of the algorithms, so besides the algorithm part, it gets the requests from the client processes, sends the information to them, and then reads the new information back and update them in server. In the real implementation, the server is waiting for the client processes to access to the critical section.

Once it's communicating with the process (processes), it sends all three matrices A, B and C to the request process (processes). If the process is with *write* request, the server sends the three matrices first, and then waits until the process send the new data back. And after receiving the new data from the client process (processes), it overwrites the original source (matrix C) with the updated data. If no process is accessing the server, the server will be idle.

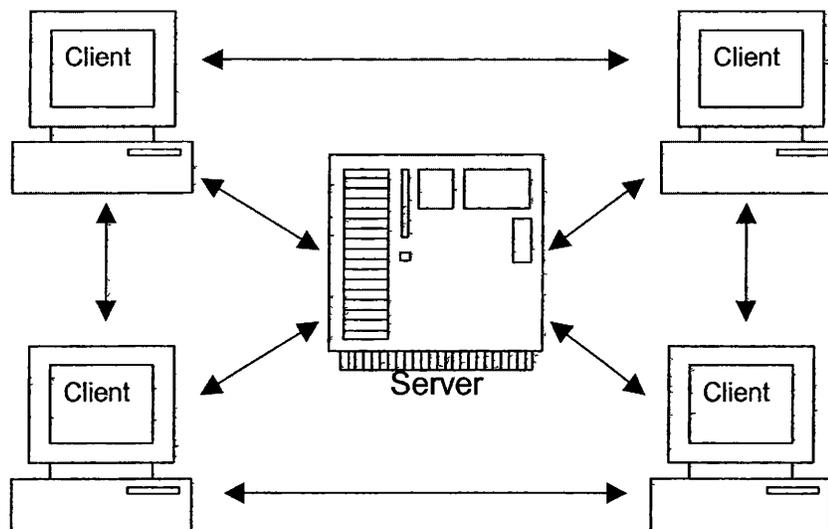


Figure 7.3: Server and Client Hosts

In the figure 10, it shows the relationships between the clients and the server. Clients can access the server as well as communicate with each other.

CHAPTER 8 PERFORMANCE COMPARISON

In Chapter 6 and 7, the implementation for the three tree-based algorithms combined with the test application has been described in details. Then here we compare the performance of each algorithm interns of the waiting time and the execution time.

8.1 Token Average Waiting Time Comparison

By collecting the result from the experimental execution, we list the token delivery time in the following tables. Detailed explanations are given in 8.1.1 and 8.1.2.

8.1.1 Simulating Three Terminals

The following table is the data of average token waiting time in milliseconds from the simulation. According to the more accurate data, the average value is calculated by testing each algorithm for 10 times.

Three computers are used, each simulating three or more terminals. If the process size is 9, then only 3 processes per computer are used; if the process size is 12, then 4 processes per computer are used; if the process size is 15, then 5 processes per computer are used. And as mentioned before, Pseudo random function is used to generate the mode for each process. Then the test

result is based on the multiple simulated terminals on each actual computer are made consistent.

Algorithm	9 Processes (milliseconds)	12 Processes (Milliseconds)	15 Processes (milliseconds)
Dynamic-tree	11580	14579	17589
DAG-based	15500	19842	23706
Fairness tree	6799	7159	7330

Table 8.1: Processes Average Waiting Time

Here is a chart based on the data in the table to enable clearer comparisons between algorithms.

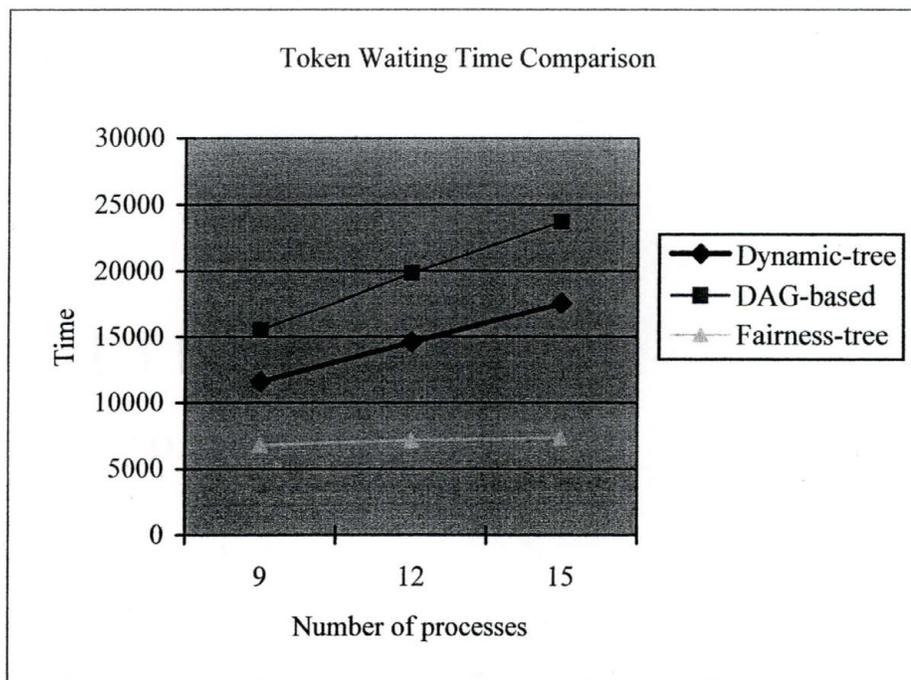


Figure 8.1: Token Average Waiting Time Comparisons

1. The blue (middle) line is for Dynamic-tree protocol. It shows that the waiting time for the token is increased linearly with the increase of the process size.
2. The red (top) line is for DAG-based protocol. It shows that the waiting time for the token is increased linearly with the increase of the process size. But it has the worse performance among these three protocols since it only allows exclusive access to the critical section.
3. The yellow (bottom) line is for Fairness tree-based protocol. The performance is the best among these three protocols. And also the average waiting time is only changing a little bit when the process number is getting bigger, not change as dramatically as the other two protocols.

8.1.2 Simulating Four Terminals

The following table is the data of token waiting time in milliseconds from the simulation. According to the more accurate data, the average value is calculated by testing each algorithm for 10 times.

There are four computers are simulated as four terminals. If the process size is 12, then only 3 processes per computer are used; if the process size is 16, then 4 processes per computer are used; if the process size is 20, then 5 processes per computer are used. Then the test result is based on the multiple simulated terminals on each actual computer are made consistent.

Algorithm	12 Processes (milliseconds)	16 Processes (milliseconds)	20 Processes (milliseconds)
Dynamic-tree	13872	17798	24309
DAG-based	18451	27701	32198
Fairness tree	8230	8512	8753

Table 8.2: Processes Average Waiting Time

The data in the table 8.2 shows the same performance result as we simulated three terminals. The fairness tree protocol has the best performance on the waiting time.

8.2 Total Execution Time Comparison

By collecting the result from the experimental execution, we list the processes execution time in the following tables. Detailed explanations are given in section 8.2.1 and 8.2.2.

8.2.1 Simulating Three Terminals

The following table is the data of processes execution time in milliseconds from the simulation. According to the more accurate data, the average value is calculated by testing each algorithm for 10 times.

Three computers are simulated as three terminals. If the process size is 9, then only 3 processes per computer are used; if the process size is 12, then 4

processes per computer are used; if the process size is 15, then 5 processes per computer are used. Then the test result is based on the multiple simulated terminals on each actual computers are made consistent.

Algorithm	9 Processes (milliseconds)	12 Processes (milliseconds)	15 Processes (milliseconds)
Dynamic-tree	131874	213791	285139
DAG-based	174321	293420	401531
Fairness tree	92195	151070	193538

Table 8.3: Processes Total Execution Time

Here is a chart based on the data in the table to have more clear explanations on each algorithm.

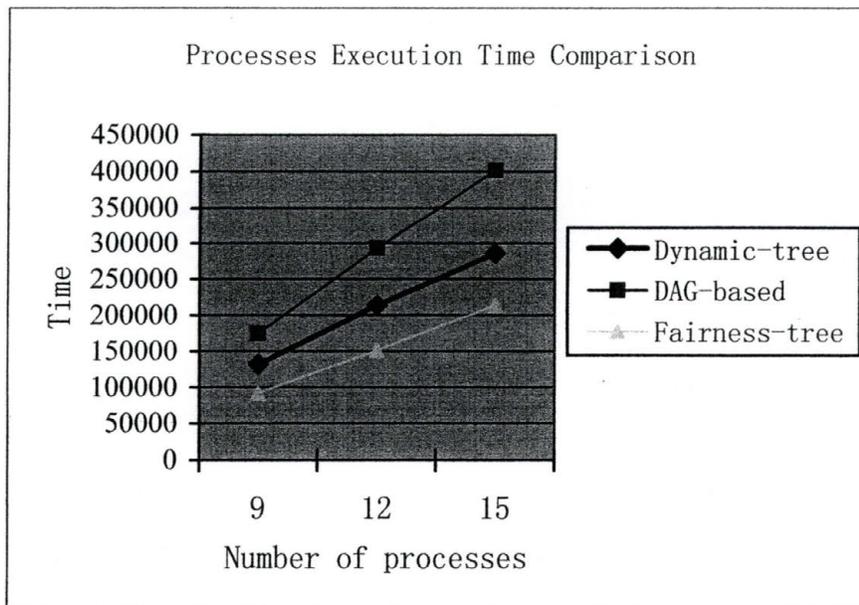


Figure 8.2: Total Execution Time Comparisons

1. The blue line shows the processes execution time of Dynamic-tree algorithm. From the chart.
2. The red line shows the processes execution time of DAG-based algorithm. It has the longest execution time than the other two algorithms.
3. The yellow line shows the processes execution time of Fairness tree-based algorithm, which has the best execution compare to Dynamic-tree and DAG-based tree algorithms.

8.2.2 Simulating Four Terminals

The following table is the data of processes execution time in milliseconds from the simulation. According to the more accurate data, the average value is calculated by testing each algorithm for 10 times.

Three computers are simulated as four terminals. If the process size is 12, then only 3 processes per computer are used; if the process size is 16, then 4 processes per computer are used; if the process size is 20, then 5 processes per computer are used. Then the test result is based on the multiple simulated terminals on each actual computers are made consistent.

Algorithm	12 Processes (milliseconds)	16 Processes (milliseconds)	20 Processes (milliseconds)
Dynamic-tree	224572	303917	432749
DAG-based	308730	487499	740837
Fairness tree	130324	176520	297829

Table 8.4: Processes Total Execution Time

The data in the table 8.4 shows the same performance result as we simulated three terminals. The fairness tree protocol has the best performance on the execution time since all the shared requester can access their critical sections at the same time no matter how many requesters they are.

CHAPTER 9 CONCLUSION AND FUTURE WORK

From the simulated study on three algorithms, it is obviously that Fairness-tree algorithm has the best performance among all the algorithms. Compared with DAG-based algorithm and Dynamic-tree algorithm, Fairness-tree algorithm has the shortest average waiting time and total execution time. Fairness-tree algorithm has an advantage by allowing the maximum concurrent requesters. Furthermore, the DAG-based algorithm has the worst performance on the waiting time and total execution because concurrent requests are not supported.

Both the DAG-based and Dynamic-tree algorithms are passive, since the root does not keep checking the value of itself to control the access of each process. Instead, the root only passes the value to the branches. Conversely, the Fairness-tree algorithm has an active root which keeps checking the status of its grant and request to control the access of each process, and passive branches which execute only when requests are outstanding.

There are several areas for future work. One promising area is comparison of other algorithms. Another area with potential is the utilization of more sophisticated benchmarks. Performance metrics in this work were average waiting time and total execution time. Future work can look at other metrics, such as message passing and critical section request interval comparisons.

APPENDICES

Appendix I Processes.java

```
/**
 *File name: "Processes.java"
 *This file is used for the Dynamic-tree algorithm. It holds the process information
 */

import java.io.*;
import java.net.*;
import java.lang.*;

/**
 * Processes Class
 */
public class Processes implements Serializable
{
    public int portNumber;
    public int connectPort;
    public InetAddress pAddress;
    public InetAddress serverAddress;
    public int pID;
    protected String myMode;
    public Processes next;
    protected boolean myToken;

    /**
     *Constructor without parameters
     */
    Processes()
    {
        this.portNumber = -1;
        this.connectPort = 0;
        this.pAddress = null;
        this.serverAddress = null;
        this.pID = -1;
        this.myMode = "UNDEF";
        this.next = null;
        this.myToken = false;
    }

    /**
     * Constructor with 4 parameters
     */
    Processes(int pNum, int sP, InetAddress pAdd, InetAddress sAdd )
    {
        this.portNumber = pNum;
        this.connectPort = sP;
        this.pAddress = pAdd;
        this.serverAddress = sAdd;
        this.pID = -1;
        this.myMode = "UNDEF";
    }
}
```

```
        this.next = null;
        this.myToken = false;
    }

    /**
     *Defines object input
     */
    public void readObject(java.io.ObjectInputStream stream)throws IOException,
        ClassNotFoundException
    {
        stream.defaultReadObject();
    }

    /**
     *Defines object output
     */
    public void writeObject(java.io.ObjectOutputStream stream)throws IOException,
        ClassNotFoundException
    {
        stream.defaultWriteObject();
        stream.flush(),
    }

    /**
     *Set the address
     */
    public void setAddress(InetAddress addr)
    {
        this.pAddress = addr;
    }

    /**
     *Set the port number
     */
    public void setPort(int p)
    {
        this.portNumber = p;
    }

    /**
     * Returns the mode of the object
     */
    public String getMode()
    {
        return myMode;
    }

    /**
     * Returns the next node
     */
    public Processes getNext()
```

```
{
    return next;
}

/**
 * Returns the token of the node
 */
public boolean getToken ()
{
    return myToken;
}

/**
 * Set the mode for the object
 */
public void setMode(String newMode)
{
    this.myMode= newMode;
}

/**
 * Sets the next
 */

public void setNext(Processes next)
{
    this.next = next;
}

/**
 *Sets the token.
 */
public void setToken(boolean value)
{
    this.myToken = value;
}

/**
 *Sets the ID
 */
public void setID (int newID)
{
    this.pID = newID;
}

}/*end of Processes class*/
```

Appendix II Algorithm.java

```

/**
 *File name: "Algorithm.java"
 *It is implemented with the Dynamic-tree algorithm
 */

import java.io.*;
import java.net.*;
import java.lang.*;

/**
 * Algorithm class
 */
public class Algorithm
{
    /**
     * Constructor without any parameters
     */
    public Algorithm()
    {}

    /*Variables used in the class*/
    static String token_mode="UNDEF";
    static String next_mode = "UNDEF";
    static int self = 0;
    static boolean concurrent_read = false;
    static boolean cToken = false;
    protected boolean pending_acks = false;
    static Processes dir =null;
    static Processes nextNode =null;
    static Processes next_readers = null;

    /**
     * Lock function which controls the readers and writers' order to get into the CS
     */
    public void lock(Processes temp) throws IOException
    {
        if(!temp.myToken)
        {
            request(temp, temp.myMode, dir);
            dir = temp;
            await(token(pending_acks));
            if(temp.myMode=="READ" &&( token_mode == "READ"||
                token_mode == "MULTIREAD"))
                concurrent_read = true;
            else
                concurrent_read = false;
            if(temp.next != null && concurrent_read == true)
            {

```

```

        temp.next.myToken= true;
        next_readers = temp.next;
        nextNode = null;
        token_mode = "MULTIREAD";
    }
}

/**
 * UnLock function which release the lock for readers and writers
 */
public void unLock(Processes temp)
{
    if(token_mode=="MULTIREAD" && pending_acks == false)
    {
        sendAck(next_readers);
        next_readers =null;
    }
    if(temp.next !=null && pending_acks == false)
    {
        temp.next.myToken = false;
        nextNode = null,
    }
    else if(token_mode != "MULTIREAD")
        cToken = true;
    if(temp.next!=null)
        temp.next.myToken = true;
    token_mode = "UNDEF";
}

/**
 * Receive_request function which handles request propagation in the tree
 */
public void receive_request(Processes sender)
{
    if(dir !=sender )
        request(sender, sender.myMode, dir);
    else
    {
        if(token_mode == "READ" && sender.myMode == "READ")
            concurrent_read = true;
        if((cToken == true && pending_acks == false) ||
            concurrent_read == true)
        {
            sendToken(!cToken, sender);
            sender.myToken = false;
            if(concurrent_read == true)
            {
                token_mode = "MULTIREAD";
                next_readers=sender;
            }
        }
    }
}

```

```

        }
        else
        {
            nextNode= sender;
            cToken=false;
            next_mode = sender.myMode;
        }
    }
    dir = sender;
}

/**
 * Receiver_ack function which count the acknowledgments
 */
public void receive_ack(Processes temp)
{
    pending_acks =false;
    if( token_mode == "UNDEF")
    {
        sendAck(next_readers);
        next_readers = null;
    }
    else if(cToken && nextNode != null)
    {
        sendToken(false, temp.next);
        nextNode = null;
        cToken = false;
    }
}

/**
 * Private function, which prints out the message when sending the request
 */
private String request(Processes sender , String mode, Processes rec)
{
    return ("sender "+sender.portNumber + " send request to " +
            rec.portNumber);
}

/**
 * Private function, which returns token status base on the value of expect
 *acknowlegements
 */
private boolean token(boolean expect_ack)
{
    if(expect_ack == true)
        cToken = false;
    else
        cToken = true;
    return cToken;
}

```

```
/**
 * Private function, which returns the wait status
 */
private boolean await(boolean t)
{
    if( t == true)
        return false;
    else
        return true;
}

/**
 * Private function, which prints out the messages when sending out the token
 */
private void sendToken(boolean value, Processes next)
{
    next.myToken = value;
}

/**
 * Private function, which sends out the acknowledgement to the first node
 * in the queue
 */
private void sendAck(Processes theOne)
{
    pending_acks =false;
}

}/*end of Algorithm class*/
```

Appendix III TreeToken.java

```

/**
 *File name: TreeToken.java
 *This class is used for Dynamic-tree algorithm, which includes how the
 *processes are used to build the tree;how each process recognizes its privilege
 *to access to the critical section;how the processes communicate with the server
 *to read or overwrite the source
 */

import java.io.*;
import java.net.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;

/**
 * TreeToken Class
 */
public class TreeToken implements Runnable, WindowListener, ActionListener
{
    /**
     * Variables used in this class
     */
    protected String host;
    protected int port;
    protected Frame frame;
    protected TextArea output;
    protected TextField input;
    protected PrintWriter out ;
    protected DataInputStream oIn;
    protected DataOutputStream oOut;
    protected FileOutputStream fileOut;
    protected Thread listener;
    protected Socket socket;
    protected Processes p = new Processes();
    private InetAddress localAddress;
    static Algorithm protocolObj = new Algorithm();
    public Processes[] pArray = new Processes[4];
    static int tempCounter=0;
    protected static int clientPort[] = {3000, 3200, 3400, 3600};
    static int clientArray = 0;
    protected int localPort = 0;
    static int index = 0;
    static int theOrder = 0;

    /*Static variables used for the matrix*/
    static int array1[][] = new int [16][16];
    static int array2[][] = new int [16][16];
    static int array3[][] = new int [16][16];

```

```

static Matrix arrayA = new Matrix (16,16, array1);
static Matrix arrayB = new Matrix ( 16,16, array2);
static Matrix arrayC = new Matrix ( 16,16, array3);

/*Variables used to record the waiting time*/
protected long startWaiting = 0;
protected long endWaiting = 0;
protected long totalWaiting = 0;

/**
 * Constructor with two paras
 */
public TreeToken (String host, int port) throws FileNotFoundException
{
    this.host = host;
    this.port = port;
    frame = new Frame ("TokenClient [" + host + ":' + port + "']");
    frame.addWindowListener (this);
    output = new TextArea ();
    output.setEditable (false);
    input = new TextField ();
    input.addActionListener (this);
    frame.add ("Center", output);
    frame.add ("South", input);
    frame.pack ();
    out = new PrintWriter(new FileOutputStream("output" + tempCounter
        + ".txt"));
    tempCounter++;
}

/**
 *Function will be called in main
 */
public synchronized void start () throws IOException
{
    if (listener == null)
    {
        Socket socket = new Socket (host, port,
            InetAddress.getLocalHost(), clientPort[clientArray]);
        clientArray ++;
        localAddress = socket.getLocalAddress();
        p = new Processes(socket.getLocalPort(), 1000, localAddress,
            socket.getInetAddress());
        try
        {
            oIn = new DataInputStream
                (new BufferedInputStream (socket.getInputStream ()));
            oOut = new DataOutputStream
                (new BufferedOutputStream (socket.getOutputStream ()));
        } catch (IOException ex) {
            socket.close ();
        }
    }
}

```

```

        throw ex;
    }
    listener = new Thread (this);
    listener.start ();
    frame.setVisible (true);
}
}

/**
 * function stop
 */
protected synchronized void stop () throws IOException
{
    frame.setVisible (false);
    if (listener != null)
    {
        listener.interrupt ();
        listener = null;
        oOut.close ();
    }
}

/**
 * run the threads
 */
public void run ()
{
    try
    {
        while (!Thread.interrupted ())
        {
            getIDFromServer();
            handleCS(p);
            out.close();
        }
    } catch (IOException ex) {
        handleIOException (ex);
    } catch (InterruptedException e) {}
    catch (ClassNotFoundException c) {}
}

/**
 * Each client is assigned the ids by the server, and gets different port numbers for
 * and the information about their neighbors
 */
protected void getIDFromServer() throws IOException, ClassNotFoundException
{
    int id = oIn.readInt();
    p.setID(id);
    generateRandomMode(p);
    output.append("\nMy id is : " + id);
}

```

```

        output.append("\nMy port number is : " + p.portNumber);
        out.println("My id is : " +id);
        out.println("My port number is : " + p.portNumber);
        pArray[id] = p;
        index++;
        assignedNext(p);
    }

    /**
     *The next pointer is set
     */
    protected synchronized void assignedNext(Processes theP) throws IOException,
        ClassNotFoundException
    {
        theP.next = new Processes();
        if((theP.pID+1)< pArray.length)
            theP.next.setID (oIn.readInt());
    }

    /**
     * Generates the random mode for each process
     */
    protected synchronized void generateRandomMode(Processes tempN) throws
        IOException
    {
        String mode[] = {"WRITE", "READ"};
        int mode_id = 2;
        int index = (int)(Math.random()*mode_id);
        tempN.setMode(mode[index]);
        output.append("\nMy mode is : " + mode[index] );
        out.println("My mode is : " + mode[index] );
    }

    /**
     *This function is used to set the root
     */
    protected void initialValue(Processes pro)throws IOException
    {
        if(pro.pID== 0)
        {
            pro.myToken = true;
            protocolObj.token_mode = pro.myMode;
            protocolObj.dir =pro;
            if(pro.next !=null)
            {
                protocolObj.next_mode = pro.next.myMode;
                protocolObj.nextNode = pro.next;
            }
        }
    }
}

```

```

/**
 *This function is used to set the value of the algorithm variables
 */
protected void setValue(Processes pro)throws IOException
{
    if(pro.next!= null)
    {
        pro.next.myToken = true;
        protocolObj.token_mode = pro.next.myMode;
        protocolObj.dir =pro.next;
        if(pro.next.next !=null)
        {
            protocolObj.next_mode = pro.next.next.myMode;
            protocolObj.nextNode = pro.next.next;
        }
    }
}

/**
 *This function is used to set the readers after they exit the critical section
 */
protected void dealReader(Processes rNode) throws IOException
{
    protocolObj.receive_ack(rNode);
    if(protocolObj.token_mode equals("READ")|| protocolObj pending_acks
        == false)
    {
        protocolObj.concurrent_read =false;
        protocolObj.unlock(rNode);
        setValue(rNode);
    }
}

/**
 *It handles reading and writing for the processes
 */
protected void controlRW(Processes tNode) throws IOException,
    InterruptedException
{
    oOut.writeUTF(tNode.myMode);
    oOut.flush();
    if(tNode.myMode.equals("READ"))
    {
        readFromFile();
        tNode.myToken= false;
        protocolObj.pending_acks=true;
        dealReader(tNode);
    }
    else if( tNode.myMode.equals("WRITE"))
    {

```

```

        writeToFile();
        tNode.myToken = false;
        protocolObj.unlock(tNode);
        setValue(tNode);
    }
    else
        System.out.println("***WRONG***");
}

/**
 *It calls the algorithm to control the access of entering the critical section
 */
protected synchronized void handleCS(Processes tempNode) throws
    IOException, InterruptedException
{
    startWaiting = System.currentTimeMillis() ;
    if(tempNode.pID== 0)
    {
        tempNode.myToken = true;
        protocolObj.token_mode = tempNode.myMode;
        protocolObj.dir =tempNode;
        if(tempNode.next !=null)
        {
            protocolObj.next_mode = tempNode.next.myMode;
            protocolObj.nextNode = tempNode.next,
        }
    }
    protocolObj.lock(tempNode);
    if(protocolObj.next_readers !=null && protocolObj.next_readers.pID
        == tempNode.pID)
        tempNode.myToken = true;
    while(tempNode.myToken == false)
    {
        Thread.sleep(1);
        protocolObj.lock(tempNode);
        if(protocolObj.next_readers !=null &&
            protocolObj.next_readers.pID == tempNode.pID)
            tempNode.myToken = true;
        theOrder = oIn.readInt();
        if(tempNode.pID == theOrder)
            tempNode.myToken = true;
        if(tempNode.myToken ==true)
            break;
    }
    if(tempNode.myToken==true)
    {
        endWaiting = System.currentTimeMillis() ;
        totalWaiting = endWaiting - startWaiting;
        output.append("\nTotal Waiting Time is " + totalWaiting );
        controlRW(tempNode);
        oOut.writeInt(tempNode.pID);
    }
}

```

```

        oOut.flush();
    }
    oOut.close();
    oIn.close();
}

/**
 * Read the files from server if the client has the token and has read mode
 */
public synchronized void readFromFile() throws IOException, InterruptedException
{
    output.append("\nI'm going to read now... ");
    out.println("\nI'm going to read now... ");
    output.append("\nFirst Matrix \n");
    out.println("First Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)
        {
            arrayA.M[i][j]= oIn.readInt();
            output.append(" " + arrayA.M[i][j] + " ");
            out.print(" " + arrayA.M[i][j] + " ");
        }
        output.append("\n");
        out.print("\n ");
    }
    output.append("\nSecond Matrix \n");
    out.println("Second Matrix \n");
    for(int i =0;i<arrayB.rows; i++)
    {
        for(int j = 0;j<arrayB.cols; j++)
        {
            arrayB.M[i][j]= oIn.readInt();
            output.append(" " + arrayB.M[i][j] + " ");
            out.print(" " + arrayB.M[i][j] + " ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\nThird Matrix \n");
    out.println("\nThird Matrix \n ");
    for(int i =0;i<16; i++)
    {
        for(int j = 0;j<16; j++)
        {
            int temp = oIn.readInt();
            output.append(" " + temp + " | ");
            out.print(" " + temp + " | ");
        }
        output.append("\n");
        out.println(" ");
    }
}

```

```

    }
    output.append("\n I finished reading.....\n" );
    out.println(" I finished reading..... \n" );
}

/**
 * Write to file in the server if the client has the token and has the write mode
 */
protected synchronized void writeToFile() throws IOException, InterruptedException
{
    output.append("\nI'm going to write now...");
    out.println("\nI'm going to write now...");
    output.append("\nFirst Matrix \n");
    out.println("First Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)
        {
            arrayA.M[i][j]= oIn.readInt();
            output.append(" " + arrayA.M[i][j] + " ");
            out.print(" " + arrayA.M[i][j] + " ");
        }
        output.append("\n");
        out.print("\n ");
    }
    output.append("\nSecond Matrix \n");
    out.println("Second Matrix \n");
    for(int i =0;i<arrayB.rows; i++)
    {
        for(int j = 0;j<arrayB.cols; j++)
        {
            arrayB.M[i][j]= oIn.readInt();
            output.append(" " + arrayB.M[i][j] + " | ");
            out.print(" " + arrayB.M[i][j] + " | ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\nThird Matrix \n");
    out.println("Third Matrix \n");
    for(int i =0;i<arrayC.rows; i++)
    {
        for(int j = 0;j<arrayC.cols; j++)
        {
            arrayC.M[i][j] = oIn.readInt();
            output.append(" " + arrayC.M[i][j] + " | ");
            out.print(" " + arrayC.M[i][j] + " | ");
        }
        output.append("\n");
        out.println(" ");
    }
}

```

```

int sum;
for(int l=0;l<16;l++)
{
    for(int j=0;j<16;j++)
    {
        sum=0;
        for(int k=0;k<16;k++)
            sum = (int)sum + arrayA.M[l][k] * arrayB.M[k][j];
        arrayC.M[l][j] = sum + arrayC.M[l][j];
    }
}
output.append("\nThe new source is: \n" );
out.println("The new source is: \n" );
for(int i =0;i<arrayC.rows; i++)
{
    for(int j = 0;j<arrayC.cols; j++)
    {
        oOut.writeInt( arrayC.M[i][j]);
        oOut.flush();
        output.append(" " + arrayC.M[i][j] + " | " );
        out.print(" " + arrayC.M[i][j] + " | " );
    }
    output.append("\n");
    out.println(" ");
}
output.append("\nI'm finished writing now ... \n");
out.println("I'm finished writing now ... \n");
}

/**
 * Function handles IO exceptions
 */
protected synchronized void handleIOException (IOException ex)
{
    if (listener != null)
    {
        output.append (ex + "\n");
        input.setVisible (false);
        frame.validate ();
        if (listener != Thread.currentThread ())
            listener.interrupt ();
        listener = null;
        try {
            oOut.close ();
        } catch (IOException ignored) {}
    }
}

public void windowOpened (WindowEvent event)
{
    input.requestFocus ();
}

```

```

public void windowClosing (WindowEvent event)
{
    try {
        stop ();
    } catch (IOException ex) {
        ex.printStackTrace ();
    }
}
public void windowClosed (WindowEvent event) {}
public void windowIconified (WindowEvent event) {}
public void windowDeiconified (WindowEvent event) {}
public void windowActivated (WindowEvent event) {}
public void windowDeactivated (WindowEvent event) {}
public void actionPerformed (ActionEvent event)
{
    try {
        input.selectAll();
        oOut.writeUTF (event.getActionCommand ());
        oOut.flush ();
    } catch (IOException ex) {
        handleIOException (ex);
    }
}

/**
 * Function main
 */
public static void main (String[] args) throws IOException, InterruptedException,
FileNotFoundException
{
    for (int i = 0;i<4;i++)
    {
        TreeToken theClient = new TreeToken("147.26.101.141", 1000);
        theClient.start ();
    }
}
}/*end of TokenC class*/

```

Appendix IV Processes.java

```

/**
 *File name: "Processes.java"
 *This file is used by DAG-based algorithm, which holds the process information
 */

import java.io.*;
import java.net.*;
import java.lang.*;

/**
 * Processes class
 */
public class Processes
{
    public int portNumber;
    public int connectPort;
    public InetAddress pAddress;
    public InetAddress serverAddress;
    public int pID,
    protected String myMode,
    protected Processes nextNode;
    protected Item iObj;
    protected boolean rec ;
    protected boolean token ;
    public Request reObj;

    /**
     *Subclass Item
     */
    public class Item
    {
        public int last;
        public int next;
        public boolean holding;
        /**
         *Constructor without parameter
         */
        public Item()
        {
            last = 0;
            next =0;
            holding = false;
        }
        /**
         *Constructor with 3 parameter
         */
        public Item(int lastItem, int nextItem, boolean theHolding)
        {
            this.last = lastItem;

```

```

        this.next = nextItem;
        this.holding = theHolding;
    }
}/*end of Item class */

/**
 *Constructor without parameters
 */
Processes( )
{
    this.portNumber = -1;
    this.connectPort = -1;
    this.pAddress = null;
    this.serverAddress = null;
    this.pID = -1;
    this.iObj = new Item();
    this.rec = false;
    this.token = false;
    this.nextNode = null;
    this.myMode = "UNDEF";
    this.reObj = new Request(),
}

/**
 *Constructor with parameters
 */
Processes(int pN, int sP, InetAddress addr, InetAddress sAddr)
{
    this.portNumber = pN;
    this.connectPort = 0;
    this.pAddress = addr;
    this.serverAddress = sAddr;
    this.pID = -1;
    this.myMode = "UNDEF";
    this.nextNode = null;
    this.iObj = new Item();
    this.reObj = new Request();
    this.rec = false;
    this.token = false;
}

/**
 *Set the address
 */
public void setAddress(InetAddress addr)
{
    this.pAddress = addr;
}

/**
 * Set the mode

```

```
    */  
    public void setMode(String newMode)  
    {  
        this.myMode= newMode;  
    }  
  
    /**  
     * Sets the id  
     */  
    public void setObj(int theID)  
    {  
        this.pID = theID;  
    }  
  
    /**  
     * Sets the next.pointer  
     */  
    public void setNext(Processes next)  
    {  
        this.nextNode = next;  
    }  
  
    /**  
     *Sets the token  
     */  
    public void setToken(boolean value)  
    {  
        this.token = value;  
    }  
}/* end of Processes class*/
```

Appendix V Request.java

```

/**
 *File name: "Request.java"
 *This file is used by DAG-based algorithm, which includes the information of
 *the requester and the request receiver
 */
import java.io.*;
import java.net.*;
import java.lang.*;
/**
 *Request class
 */
public class Request
{
    int rid;
    int sid;
    /**
     *Constructor without parameter
     */
    public Request()
    {
        rid = 0;
        sid = 0;
    }
    /**
     *Constructor with 2 parameter
     */
    public Request(int senderID, int initialID)
    {
        this.rid = senderID;
        this.sid = initialID;
    }
    /**
     *Prints out the requester of sender and receiver
     */
    public String sendReq ()
    {
        return "(" + rid + "," + sid + ")";
    }
    /**
     *Sets the information of request sender and receiver
     */
    public String sendReq(int firstID, int secondID)
    {
        this.rid = firstID;
        this.sid = secondID;
        return "(" + rid + "," + sid + ")";
    }
}
/* end of Request class */

```

Appendix VI Queue.java

```
/**
 *File name: "Queue.java"
 *This file is used by DAG-based algorithm, which includes the queue structure
 */

import java.io.*;
import java.net.*;
import java.lang.*;

/**
 * Queue class
 */
public class Queue
{
    public Processes first, last;
    public int length = 0;

    /**
     * Constructor without paramter
     */
    public Queue()
    {
        first = null;
        last = null;
    }

    /**
     * Add a node into the queue from the back
     */
    public void addToQueue(Processes newNode)
    {
        if (first == null)
        {
            last = newNode;
            first = last;
        }
        else
        {
            last.setNext(newNode);
            last = newNode;
        }
        length++;
    }

    /**
     * Remove an node from the queue from the front
     */
    public Processes removeFromQueue()
    {

```

```
        if (first == null)
            return null;
        else
        {
            Processes temp = first;
            first = first.nextNode;
            length--;
            return temp;
        }
    }
}/*end of Queue class */
```

Appendix VII Algorithm.java

```

/**
 *File name: "Algorithm.java"
 *It is implemented with the DAG-based algorithm
 */
import java.io.*;
import java.net.*;
import java.lang.*;

public class Algorithm
{
    static Processes rootNode;
    static Processes nNode;
    /**
     * Constructor without any parameters
     */
    public Algorithm(){}
    /**
     *Show how the request and token are passing with
     *the distributed queue
     */
    public void passToken(Processes temp)
    {
        if(temp.pID==1)
        {
            if(nNode!=null)
            {
                nNode.reObj.rid = temp.pID;
                nNode.reObj.sid = temp.pID;
                nNode.rec = true;
            }
            temp.iObj.last =0;
        }
        else if( temp.reObj.rid !=temp.reObj.sid &&
            temp.rec==true&&temp.token==false)
        {
            nNode.reObj.rid= temp.reObj.rid;
            nNode.reObj.sid = temp.reObj.sid;
            temp.iObj.last = temp.reObj.rid;
        }
        else if(temp.token==true&&temp.rec ==true)
        {
            if(temp.iObj.holding ==false)
                temp.iObj.next = temp.reObj.rid;
            else
                nNode.token = true;
            temp.iObj.next =temp.reObj.rid;
        }
    }
}
/* end of Algorithm class */

```

Appendix VIII TreeToken.java

```

/**
 *File name: TreeToken.java
 *This class is used for DAG-based algorithm, It includes how the processes
 *are used to build the tree;how each process recognizes its privilege to
 *access to the critical section;how the processes communicate with the
 *server to read or overwrite the source
 */

import java.io.*;
import java.net.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;

/**
 * TreeToken Class
 */
public class TreeToken implements Runnable, WindowListener, ActionListener
{
    /**
     * Variables used in this class
     */
    protected String host;
    protected int port;
    protected Frame frame;
    protected TextArea output;
    protected TextField input;
    protected PrintWriter out ;
    static int tempCounter=0;
    protected DataInputStream dataIn;
    protected DataOutputStream dataOut;
    protected FileOutputStream fileOut;
    protected Thread listener;
    protected Socket socket ;
    protected boolean myToken = false;
    protected static int clientPort[] = {3000, 3200, 3400, 3600};
    static int clientArray = 0;
    protected int localPort = 0;
    protected Processes qNode;
    private InetAddress localAddress;
    static Queue waitQueue = new Queue();
    static int computerID = 0;
    static int aIndex =0;
    static String tokenMode = "UNDEF";
    static int j = 0;
    protected String tempMode = "UNDEF";
    static Algorithm algObject= new Algorithm();
    static int tmp= 1;
    static int theOrder = 0;

```

```

/*Static variables used for the matrix*/
static int array1[][] = new int[16][16];
static int array2[][] = new int [16][16];
static int array3[][] = new int [16][16];
static Matrix arrayA = new Matrix (16, 16, array1);
static Matrix arrayB = new Matrix ( 16, 16, array2);
static Matrix arrayC = new Matrix ( 16, 16, array3);

/*Variables used for the waiting time*/
protected long startWaiting = 0;
protected long endWaiting = 0;
protected long totalWaiting = 0;

/**
 * Constructor with two paras
 */
public TreeToken (String host, int port) throws FileNotFoundException
{
    this.host = host;
    this.port = port;
    frame = new Frame ("TokenClient [" + host + "' + port + "]"),
    frame.addWindowListener (this),
    output = new TextArea ();
    output.setEditable (false);
    input = new TextField ();
    input.addActionListener (this);
    frame.add ("Center", output);
    frame.add ("South", input);
    frame.pack ();
    out = new PrintWriter(new FileOutputStream("output" + tempCounter
        + ".txt"));
    tempCounter++;
}

/**
 *Function will be called in main
 */
public synchronized void start () throws IOException
{
    if (listener == null)
    {
        Socket socket=new Socket(host, port,InetAddress.getLocalHost(),
            clientPort[clientArray]);
        clientArray ++;
        localAddress = socket.getLocalAddress();
        qNode = new Processes(socket.getLocalPort(), 1000,
            localAddress, socket.getInetAddress());
        try
        {
            dataIn = new DataInputStream
                (new BufferedInputStream (socket.getInputStream ()));

```

```

        dataOut = new DataOutputStream
            (new BufferedOutputStream (socket.getOutputStream ()));

        } catch (IOException ex) {
            socket.close ();
            throw ex;
        }
        listener = new Thread (this);
        listener.start ();
        frame.setVisible (true);
    }
}

/**
 * function stop
 */
public synchronized void stop () throws IOException
{
    frame.setVisible (false);
    if (listener != null)
    {
        listener.interrupt ();
        listener = null;
        dataOut.close ();
    }
}

/**
 * run the threads
 */
public void run ()
{
    try
    {
        while (!Thread.interrupted ())
        {
            getIDFromServer();
            handleCS(qNode);
            out.close();
        }
    } catch (IOException ex) {
        handleIOException (ex);
    } catch (InterruptedException e) {}
}

/**
 * Each client is assigned the id by the server, gets different port numbers
 * and gets the information about their neighbors
 */
protected synchronized void getIDFromServer() throws IOException
{
    

```

```

int id = dataIn.readInt();
qNode.setID(id);
aIndex ++;
qNode.pID = id;
if(id+1>4)
qNode.iObj.last = 0;
else
qNode.iObj.last = id+1;
qNode.setMode(generateRandomMode());
waitQueue.addToQueue(qNode) ;
output.append("\nMy id is : "+ id + " with port # " + qNode.portNumber);
out.println("My id is : " + id + " with port # " + qNode.portNumber);
output.append("\nMy last: " + qNode.iObj.last + " next: "
+qNode.iObj.next + " holding : " + qNode.iObj.holding);
out.println("My last: " + qNode.iObj.last + " next: "
+qNode.iObj.next + " holding : " + qNode.iObj.holding);
}

/**
 * Generates the random mode for each client
 */
protected synchronized String generateRandomMode() throws IOException
{
    String mode[] = {"WRITE", "READ","WRITE"};
    int mode_id = 3;
    int index = (int)(Math.random()*mode_id);
    output.append("\nMy mode is : " +mode[ index]);
    out.println("My mode is : "+ mode[ index]);
    return mode[index];
}

/**
 *Handles the access of the processes
 */
protected void handleCS(Processes aNode)throws IOException,
    InterruptedException
{
    startWaiting = System.currentTimeMillis() ;
    waitQueue.addToQueue(aNode) ;
    theOrder = dataIn.readInt();
    if(computerID == theOrder)
    initialValue();
    else
        while(computerID !=theOrder )
        {
            Thread.yield();
            theOrder = dataIn.readInt();
            if(computerID == theOrder)
            {
                initialValue();
                break;
            }
        }
}

```

```

        }
    }
    if(aNode.token== false)
    {
        while(aNode.token ==false)
        {
            Thread.yield();
            algObject.passToken(aNode);
            if(aNode.token ==true)
                break;
        }
    }
    if(aNode.token==true)
    {
        endWaiting = System.currentTimeMillis() ;
        totalWaiting = endWaiting - startWaiting;
        output.append("\nTHE WAITING TIME IS : " + totalWaiting);
        tokenMode = aNode.myMode;
        controlRW(aNode);
        dataOut.writeInt(waitQueue.length);
        dataOut.flush();
        if(waitQueue.length==0)
        {
            dataOut.writeInt(theOrder);
            dataOut.flush();
        }
    }
    tmp = dataIn.readInt();
}

/**
 *Initial the value for the root
 */
public void initialValue() throws IOException
{
    algObject.rootNode = waitQueue.first;
    algObject.rootNode.token = true;
    if(waitQueue.first.nextNode!=null)
    {
        waitQueue.first.rec = true;
        algObject.nNode = waitQueue.first.nextNode;
    }
    waitQueue.first.iObj.holding = true;
}

/**
 *Resets the value after the processes exit the critical section
 */
public void resetValue() throws IOException
{
    tokenMode = "UNDEF";
}

```

```

waitQueue.first.token =false;
waitQueue.first.iObj.holding = false;
waitQueue.first.rec = false;
waitQueue.removeFromQueue();
if(waitQueue.length>0&&tokenMode == "UNDEF")
{
    waitQueue.first.setToken(true);
    waitQueue.first.iObj.holding = true;
    if(waitQueue.first.nextNode!=null)
    {
        waitQueue.first.rec = true;
        algObject.nNode = waitQueue.first.nextNode;
    }
}
}

/**
 *Controls the read and write for the processes
 */
public void controlRW(Processes tNode) throws IOException,
    InterruptedException
{
    dataOut.writeUTF(tNode.myMode);
    dataOut.flush();
    if( tNode.myMode.equals("READ") )
    {
        readFromFile();
        resetValue();
    }
    else if(tNode.myMode.equals("WRITE"))
    {
        writeToFile();
        resetValue();
    }
    else
        System.out.println("WRONG!!!");
}

/**
 * Read the files from server if the client has the token and has read mode
 */
public synchronized void readFromFile() throws IOException,
    InterruptedException
{
    output.append("\n\nI'm going to read now..." );
    ut.println("\n\nI'm going to read now..." );
    output.append("\n\nFirst Matrix \n");
    out.println("\n Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)

```

```

        {
            arrayA.M[i][j]= dataIn.readInt();
            output.append(" " + arrayA.M[i][j] + " ");
            out.print(" " + arrayA.M[i][j] + " ");
        }
        output.append("\n");
        out.print("\n ");
    }
    output.append("\nSecond Matrix \n");
    out.println("\nSecond Matrix \n");
    for(int i =0;i<arrayB.rows; i++)
    {
        for(int j = 0;j<arrayB.cols; j++)
        {
            arrayB.M[i][j]= dataIn.readInt();
            output.append(" " + arrayB.M[i][j] + " ");
            out.print(" " + arrayB.M[i][j] + " ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\nThird Matrix \n");
    out.println("\nThird Matrix \n" );
    for(int i =0;i<16; i++)
    {
        for(int j = 0,j<16; j++)
        {
            int temp = dataIn.readInt();
            output.append(" " + temp + " | " );
            out.print(" " + temp + " | ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\n I finished reading....." );
    out.println(" I finished reading....." );
}

/**
 * Write to file in the server if the client has the token and has the write mode
 */
protected synchronized void writeTofile() throws IOException,
    InterruptedException
{
    output.append("\n\nI'm going to write now...");
    out.println("\nI'm going to write now...");
    output.append("\n\nFirst Matrix \n");
    out.println("\nFirst Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)

```

```

        {
            arrayA.M[i][j]= dataIn.readInt();
            output.append(" " + arrayA.M[i][j] + " ");
            out.print(" " + arrayA.M[i][j] + " ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\nSecond Matrix \n");
    out.println("Second Matrix \n");
    for(int i =0;i<arrayB.rows; i++)
    {
        for(int j = 0;j<arrayB.cols; j++)
        {
            arrayB.M[i][j]= dataIn.readInt();
            output.append(" " + arrayB.M[i][j] + " ");
            out.print(" " + arrayB.M[i][j] + " ");
        }
        output.append("\n");
        out.println(" ");
    }
    output.append("\nThird Matrix \n"),
    out.println("Third Matrix \n");
    for(int i =0;i<arrayC.rows; i++)
    {
        for(int j = 0;j<arrayC.cols; j++)
        {
            arrayC.M[i][j]= dataIn.readInt();
            output.append(" " + arrayC.M[i][j] + " | " );
            out.print(" " + arrayC.M[i][j] + " | ");
        }
        output.append("\n");
        out.println(" ");
    }
    int sum;
    for( int l=0;l<16;l++)
    {
        for(int j =0;j<16;j++)
        {
            sum =0;
            for (int k = 0;k<16;k++)
                sum = (int) sum + arrayA.M[l][k] * arrayB.M[k][j];
            arrayC.M[l][j] = sum+ arrayC.M[l][j];
        }
    }
    output.append("\nThe new source is: \n" );
    out.println("The new source is: \n" );
    for(int i =0;i<arrayC.rows; i++)

        for(int j = 0;j<arrayC.cols; j++)
        {

```

```

        dataOut.writeInt( arrayC.M[i][j]);
        dataOut.flush();
        output.append(" " + arrayC.M[i][j] + " | " );
        out.print(" " + arrayC.M[i][j] + " | " );
    }
    output.append("\n");
    out.println(" ");
}
output.append("\nI'm finished writing now .....");
out.println("I'm finished writing now .....");
}
/**
 * Function handles IO exceptions
 */
protected synchronized void handleIOException (IOException ex)
{
    if (listener != null)
    {
        output.append (ex + "\n");
        input.setVisible (false);
        frame.validate ();
        if (listener != Thread.currentThread ())
            listener.interrupt ();
        listener = null;
        try {
            dataOut.close ();
        } catch (IOException ignored) {}
    }
}

public void windowOpened (WindowEvent event)
{
    input.requestFocus ();
}
public void windowClosing (WindowEvent event)
{
    try {
        stop ();
    } catch (IOException ex) {
        ex.printStackTrace ();
    }
}
public void windowClosed (WindowEvent event) {}
public void windowIconified (WindowEvent event) {}
public void windowDeiconified (WindowEvent event) {}
public void windowActivated (WindowEvent event) {}
public void windowDeactivated (WindowEvent event) {}
public void actionPerformed (ActionEvent event)
{
    try {
        input.selectAll();
    }
}

```

```
        dataOut.writeUTF (event.getActionCommand ());
        dataOut.flush ();
    } catch (IOException ex) {
        handleIOException (ex);
    }
}

/**
 * Function main
 */
public static void main (String[] args) throws IOException, InterruptedException,
    FileNotFoundException
{
    for (int i = 0;i<4;i++)
    {
        TreeToken theClient = new TreeToken("147.26.101.142", 1000);
        theClient.start ();
    }
}
}/*end of TreeToken class*/
```

Appendix IX Processes.java

```
/**
 *File name: "Processes.java"
 *This file is used for Fairness-tree holds the process information
 */

import java.io.*;
import java.net.*;
import java.lang.*;

/**
 *Processes Class
 */
public class Processes
{
    public int portNumber;
    public int connectPort;
    public InetAddress pAddress;
    public InetAddress serverAddress;
    public int pID;
    public char grant,
    public char request,
    public Processes child;
    public LocalApplication localApp = new LocalApplication();

    /**
     *Constructor without parameters
     */
    public Processes()
    {
        this.portNumber = 0 ;
        this.connectPort = 0;
        this.pAddress = null;
        this.serverAddress = null;
        this.pID = -1;
        this.grant = 'F';
        this.request = 'F';
    }

    /**
     * Constructor with 3 parameters
     */
    public Processes(int pNum, int sP, InetAddress pAdd, InetAddress sAdd )
    {
        this.portNumber = pNum;
        this.connectPort = sP;
        this.pAddress = pAdd;
        this.serverAddress = sAdd;
        this.pID = -1;
        this.grant = 'F';
    }
}
```

```
        this.request ='F';
    }

    /**
     *Defines object input
     */
    public void readObject(java.io.ObjectInputStream stream)throws IOException,
        ClassNotFoundException
    {
        stream.defaultReadObject();
    }

    /**
     *Defines object output
     */
    public void writeObject(java.io.ObjectOutputStream stream)throws IOException,
        ClassNotFoundException
    {
        stream.defaultWriteObject();
        stream.flush();
    }

    /**
     *Sets the ID
     */
    public void setID(int newID)
    {
        this.pID = newID;
    }

    /**
     *Set the address
     */
    public void setAddress ( InetAddress addr)
    {
        this.pAddress = addr;
    }

    /**
     *Set the port number
     */
    public void setPort(int p)
    {
        this.portNumber = p;
    }

    /**
     *Set the grant
     */
    public void setGrant(char value)
    {
```

```
        this.grant = value;
    }

    /**
     *Set the request
     */
    public void setRequest(char rValue)
    {
        this.request = rValue;
    }
}/*end of Processes class*/
```

Appendix X LocalApplication.java

```
/**
 *File name: "LocalApplication.java"
 *This file is used by Fairness-tree algorithm, which holds the local application
 *information of the arbiter
 */
import java.io.*;
import java.net.*;
import java.lang.*;

/**
 *LocalApplication Class
 */
public class LocalApplication
{
    public char mode;
    public char grant ;
    public char request;
    /**
     *Constructor without parameter
     */
    public LocalApplication()
    {
        this.mode = 'F';
        this.grant = 'F';
        this.request = 'F';
    }
    /**
     *Sets the mode
     */
    public void setMode(char newMode)
    {
        this.mode = newMode;
    }
    /**
     *Sets the grant
     */
    public void setGrant (char g)
    {
        this.grant = g;
    }
    /**
     *Sets the request
     */
    public void setRequest (char r)
    {
        this.request = r;
    }
}
}/*end of LocalApplication class*/
```

Append XI Algorithm.java

```

/**
 *File name: "Algorithm.java"
 *This class is based on Fairness-tree algorithm
 */
import java.io.*;
import java.net.*;
import java.lang.*;

/**
 * Algorithm class based on Frank Mueller's algorithm
 */
public class Algorithm
{
    static int NBR;
    protected Processes curr;
    protected Processes root;

    /**
     * Constructor without any parameters
     */
    public Algorithm()
    {
        NBR = 0;
        curr = null;
        root = null;
    }

    /**
     * Constructor with only one parameters
     */
    public Algorithm(int number)
    {
        NBR=number;
        curr = null;
        root = null;
    }

    /**
     *Branch Algorithm
     */
    public void BranchExecution()
    {
        Processes tempRoot = root;
        Processes temp2 = root;
        Processes temp3 = root;
        if(root.grant == 'F' )
        {
            while(tempRoot.child !=null)
            {

```

```

        tempRoot = tempRoot.child;
        if(tempRoot.request == 'R' && root.request == 'W')
            root.request = 'B';
        if(tempRoot.request == 'R' && root.request == 'F')
            root.request = 'R';
        if(tempRoot.request == 'W' && root.request == 'W')
            root.request = 'B';
        if(tempRoot.request == 'W' && root.request == 'F')
            root.request = 'W';
    }
}
if(root.grant == 'F' && ( root.request == 'W' || root.request == 'B'))
{
    curr = root;
    while(curr.pID < NBR)
    {
        if((curr.request == 'W' || curr.request == 'B') && curr.grant != 'W')
            curr.grant = 'W';
        if(curr.request != 'W' && curr.request != 'B' && curr.grant != 'F')
            curr.grant = 'F';
        if(curr.request != 'W' && curr.request != 'B' && curr.grant == 'F')
            curr = curr.child,
    }
    if(root.request == 'B')
        root.request = 'R';
    if(root.request == 'W')
        root.request = 'F';
}
if(root.grant == 'R' && ( root.request == 'R' || root.request == 'B'))
{
    while(temp2.child != null)
    {
        temp2 = temp2.child;
        if((temp2.request == 'R' || temp2.request == 'B') &&
            temp2.grant != 'R')
            temp2.grant = 'R';
    }
    while(temp3.child != null)
    {
        temp3 = temp3.child;
        if((temp3.request == 'R' || temp3.request == 'B') &&
            temp3.grant == 'R')
            {}
        while((temp3.request == 'R' || temp3.request == 'B') &&
            temp3.grant == 'R')
            {}
        temp3.grant = 'F';
    }
    if(root.request == 'B')
        root.request = 'W';
    if(root.request == 'R')

```

```

        root.request = 'F';
    }
}

/*
 *Root execution
 */
public void RootExecution()
{
    root.grant = 'W';
    if(curr.pID < NBR)
    {
        if((curr.request == 'W' || curr.request == 'B') && curr.grant != 'W')
            curr.grant = 'W';
        if(curr.request != 'W' && curr.request != 'B' && curr.grant != 'F')
            curr.grant = 'F';
        if(curr.request != 'W' && curr.request != 'B' && curr.grant == 'F')
            curr = curr.child;
    }
    else
    {
        Processes temp = root;
        Processes temp2 = root;
        while(temp.child != null)
        {
            if((temp.child.request == 'R' || temp.child.request == 'B')
                && temp.child.grant != 'R')
                temp.child.grant = 'R';
            temp = temp.child;
        }
        while(temp2.child != null)
        {
            if((temp2.child.request == 'R' || temp2.child.request
                == 'B') && temp2.child.grant == 'R')
            {}
            while((temp2.child.request == 'R' || temp2.child.request
                == 'B') && temp2.child.grant == 'R')
            {}
            temp2.child.grant = 'F';
            temp2 = temp2.child;
        }
        curr = root.child;
    }
}

/**
 *Local Application execution
 */
public void ApplicationExecution (Processes p)
{
    if(p.request == 'W')

```

```
    {
        while(p.grant!='W')
        {
            if(p.grant == 'W')
                break;
        }
    }
else if(p.request == 'R')
{
    while(p.grant !='R')
    {
        if(p.grant =='R')
            break;
    }
}
}

/* end of class */
```

Appendix XII TreeToken.java

```

/**
 *File name: TreeToken.java
 *This class is used for Fairness-tree algorithm, which includes how the processes
 *are used to build the tree;how each process recognizes its privilege to access to
 *the critical section;how the processes communicate with the server to read or
 *overwrite the source
 */

import java.io.*;
import java.net.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;

/**
 * TreeToken Class
 */
public class TreeToken implements Runnable, WindowListener, ActionListener
{
    /**
     * Variables used in this class
     */
    protected String host;
    protected int port;
    protected Frame frame;
    protected TextArea output;
    protected TextField input;
    protected PrintWriter out ;
    static int tempCounter=0;
    protected DataInputStream dataIn;
    protected DataOutputStream dataOut;
    protected FileOutputStream fileOut;
    protected Thread listener;
    protected Socket socket ;
    private InetAddress localAddress;
    static Algorithm alg= new Algorithm();
    protected Processes p = new Processes();
    protected int clientPort[] = {3000, 3200, 3400, 3600};
    static int pCounter = 0;
    static char tokenMode = 'F';

    /*Static variables used for the matrix*/
    static int array1[][] = new int[16][16];
    static int array2[][] = new int [16][16];
    static int array3[][] = new int [16][16];
    static Matrix arrayA = new Matrix (16, 16, array1);
    static Matrix arrayB = new Matrix ( 16,16, array2);
    static Matrix arrayC = new Matrix ( 16, 16, array3);
}

```

```

/*Variables used for recording the waiting time*/
protected long startWaiting = 0,
protected long endWaiting = 0,
protected long waitingTime = 0;

/**
 * Constructor with two paras
 */
public TreeToken (String host, int port) throws FileNotFoundException
{
    this.host = host;
    this.port = port;
    frame = new Frame ("TokenClient [" + host + ':' + port + "]" );
    frame.addWindowListener (this);
    output = new TextArea ();
    output.setEditable (false);
    input = new TextField ();
    input.addActionListener (this);
    frame.add ("Center", output);
    frame.add ("South", input);
    frame.pack ();
    out=new PrintWriter(new FileOutputStream("output"+tempCounter
        + " txt")),
    tempCounter++;
}

/**
 *Function will be called in main
 */
public synchronized void start () throws IOException
{
    if (listener == null)
    {
        Socket socket = new Socket (host, port,
            InetAddress.getLocalHost(), clientPort[pCounter]);
        pCounter++;
        localAddress = socket.getLocalAddress();
        p = new Processes(socket.getLocalPort(),1000, localAddress,
            socket.getInetAddress());
        try
        {
            dataIn = new DataInputStream
                (new BufferedInputStream (socket.getInputStream ()));
            dataOut = new DataOutputStream
                (new BufferedOutputStream (socket.getOutputStream ()));
        } catch (IOException ex) {
            socket.close ();
            throw ex;
        }
        listener = new Thread (this);
        listener.start ();
    }
}

```

```

        frame.setVisible (true);
    }
}

/**
 * function stop
 */
protected synchronized void stop () throws IOException
{
    frame.setVisible (false);
    if (listener != null)
    {
        listener.interrupt ();
        listener = null;
        dataOut.close ();
    }
}

/**
 * run the threads
 */
public void run ()
{
    try
    {
        while (!Thread.interrupted ())
        {
            getIDFromServer();
            assignNeighbor();
            exeOrder(p);
            out.close();
        }
    } catch (IOException ex) {
        handleIOException (ex);
    } catch (InterruptedException e) {}
    catch (ClassNotFoundException c) {}
}

/**
 * Each client is assigned the id by the server, gets different port numbers
 * and the information about their neighbors
 */
protected void getIDFromServer() throws IOException
{
    int newID = dataIn.readInt();
    p.setID (newID);
    output.append("\nThis is Arbiter # : " + p.pID);
    out.println("This is Arbiter # : " + p.pID);
    p.localApp.setMode(generateRandomMode());
    p.setRequest(p.localApp.mode);
    dataOut.writeChar(p.request);
}

```

```

        dataOut.flush();
    }

    /**
     *The neighbor arbiter is set
     */
    protected void assignNeighbor() throws IOException, ClassNotFoundException
    {
        p.child = new Processes();
        output.append("\nset id -- " + p.child.pID);
        output.append("\nset request -- " + p.child.request);
    }

    /**
     * Generates the random mode for each process
     */
    protected char generateRandomMode() throws IOException
    {
        char mode[] = {'W', 'R'};
        int mode_id = 2;
        int index = (int)(Math.random()*mode_id);
        output.append("\nMy mode is . " +mode[ index]),
        out.println("My mode is : "+ mode[ index]),
        return mode[index];
    }

    /**
     *It calls the algorithm to control the access of entering the critical section
     */
    protected void exeOrder (Processes temp)throws IOException,
        InterruptedException
    {
        startWaiting = System.currentTimeMillis() ;
        if(temp.pID==0)
        {
            alg.root = temp;
            alg.curr = alg.root;
            alg.RootExecution();
            temp.grant = temp.request;
        }
        else
        {
            alg.BranchExecution();
        }
        if(temp.grant != temp.request)
        {
            while(temp.grant != temp.request)
            {
                Thread.yield();
                int readID = dataIn.readInt();
                output.append("readID " + readID);
            }
        }
    }

```

```

        if(readID == temp.pID&& temp.request !='F' )
        {
            alg.root.setGrant(temp.request);
            temp.grant = temp.request;
            break;
        }
    }
}
if(temp.grant == temp.request)
{
    endWaiting = System.currentTimeMillis() ;
    waitingTime = endWaiting - startWaiting;
    output.append("\nMy Waiting Time is : " + waitingTime);
    tokenMode = temp.localApp.mode;
    dataOut.writeChar(temp.request);
    dataOut.flush();
    if( temp.request=='R' )
    {
        readFromFile();
        tokenMode ='F';
    }
    else if(temp request=="W")
    {
        writeToFile();
        tokenMode ='F';
    }
    else
        System.out.println("****WRONG****");
    dataOut.writeInt(temp.pID);
    dataOut.flush();
    temp.request = 'F';
    temp.grant ='F';
    dataOut.close();
    dataIn.close();
}
}

/**
 * Read the files from server if the client has the token and has read mode
 */
public synchronized void readFromFile() throws IOException,
    InterruptedException
{
    output.append("\nI'm going to read now..." );
    out.println("\nI'm going to read now..." );
    output.append("\nFirst Matrix \n");
    out.println("First Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)
        {

```

```

        arrayA.M[i][j]= dataIn.readInt();
        output.append(" " + arrayA.M[i][j] + " ");
        out.print(" " + arrayA.M[i][j] + " ");
    }
    output.append("\n");
    out.print("\n ");
}
output.append("\nSecond Matrix \n");
out.println("Second Matrix \n");
for(int i =0;i<arrayB.rows; i++)
{
    for(int j = 0;j<arrayB.cols; j++)
    {
        arrayB.M[i][j]= dataIn.readInt();
        output.append(" " + arrayB.M[i][j] + " ");
        out.print(" " + arrayB.M[i][j] + " ");
    }
    output.append("\n");
    out.println(" ");
}
output.append("\nThird Matrix \n");
out.println("\nThird Matrix \n" ),
for(int i =0;i<16; i++)
{
    for(int j = 0;j<16; j++)
    {
        int temp = dataIn.readInt();
        output.append(" " + temp + " | ");
        out.print(" " + temp + " | ");
    }
    output.append("\n");
    out.println(" ");
}
output.append("\n I finished reading.....\n" );
out.println(" I finished reading.....\n" );
}

/**
 * Write to file in the server if the client has the token and has the write mode
 */
protected synchronized void writeToFile() throws IOException,
    InterruptedException
{
    output.append("\nI'm going to write now...");
    out.println("\nI'm going to write now...");
    output.append("\nFirst Matrix \n");
    out.println("First Matrix \n");
    for(int i =0;i<arrayA.rows; i++)
    {
        for(int j = 0;j<arrayA.cols; j++)
        {

```

```

        arrayA.M[i][j]= dataIn.readInt();
        output.append(" " + arrayA.M[i][j] + " ");
        out.print(" " + arrayA.M[i][j] + " ");
    }
    output.append("\n");
    out.print("\n ");
}
output.append("\nSecond Matrix \n");
out.println("Second Matrix \n");
for(int i =0;i<arrayB.rows; i++)
{
    for(int j = 0;j<arrayB.cols; j++)
    {
        arrayB.M[i][j]= dataIn.readInt();
        output.append(" " + arrayB.M[i][j] + " | ");
        out.print(" " + arrayB.M[i][j] + " | ");
    }
    output.append("\n");
    out.println(" ");
}
output.append("\nThird Matrix \n"),
out.println("Third Matrix \n"),
for(int i =0;i<arrayC.rows; i++)
{
    for(int j = 0;j<arrayC.cols; j++)
    {
        arrayC.M[i][j] = dataIn.readInt();
        output.append(" " + arrayC.M[i][j] + " | ");
        out.print(" " + arrayC.M[i][j] + " | ");
    }
    output.append("\n");
    out.println(" ");
}
int sum;
for(int l=0;l<16;l++)
{
    for(int j=0;j<16;j++)
    {
        sum=0;
        for(int k=0;k<16;k++)
            sum = (int)sum + arrayA.M[l][k] * arrayB.M[k][j];
        arrayC.M[l][j] = sum + arrayC.M[l][j];
    }
}
output.append("\nThe new source is: \n" );
out.println("The new source is: \n" );
for(int i =0;i<arrayC.rows; i++)
{
    for(int j = 0;j<arrayC.cols; j++)
    {
        dataOut.writeInt( arrayC.M[i][j]);
    }
}

```

```

        dataOut.flush();
        output.append(" " + arrayC.M[i][j] + " | " );
        out.print(" " + arrayC.M[i][j] + " | " );
    }
    output.append("\n");
    out.println(" ");
}
output.append("\nI'm finished writing now ...\n");
out.println("I'm finished writing now ...\n");
}

/**
 * Function handles IO exceptions
 */
protected synchronized void handleIOException (IOException ex)
{
    if (listener != null)
    {
        output.append (ex + "\n");
        input.setVisible (false);
        frame.validate ();
        if (listener != Thread.currentThread ())
            listener.interrupt ();
        listener = null,
        try {
            dataOut.close ();
        } catch (IOException ignored) {}
    }
}

public void windowOpened (WindowEvent event)
{
    input.requestFocus ();
}

public void windowClosing (WindowEvent event)
{
    try {
        stop ();
    } catch (IOException ex) {
        ex.printStackTrace ();
    }
}

public void windowClosed (WindowEvent event) {}
public void windowIconified (WindowEvent event) {}
public void windowDeiconified (WindowEvent event) {}
public void windowActivated (WindowEvent event) {}
public void windowDeactivated (WindowEvent event) {}
public void actionPerformed (ActionEvent event)
{
    try {
        input.selectAll();
        dataOut.writeUTF (event.getActionCommand ());
    }
}

```

```
        dataOut.flush ();
    } catch (IOException ex) {
        handleIOException (ex);
    }
}

/**
 * Function main
 */
public static void main (String[] args) throws IOException, InterruptedException,
    FileNotFoundException
{
    for (int i = 0;i<4;i++)
    {
        TreeToken theClient = new TreeToken ("147.26.101.141", 1000);
        theClient.start ();
    }
}
}/*end of TokenC class*/
```

Appendix XIII Matrix.java

```

/**
 *File name: "Matrix.java"
 *It is a utility file to create and initialize three matrices
 *which are hold by the server and are updated by the clients
 */

/**
 * Matrix class
 */
public class Matrix
{
    public int rows, cols;
    public int M[][];
    protected int temp1=0;
    protected int counter =0;

    /**
     * Constructor with 3 parameters
     */
    public Matrix(int tRows, int tCols, int T[][])
    {
        M = new int[tRows][tCols];
        rows = tRows;
        cols = tCols;
        for(int i = 0; i<rows; i++)
            for(int j = 0; j<cols; j++)
                M[i][j] = T[i][j];
    }

    /**
     * fills in the empty matrix with some numbers
     */
    public void makeMatrix(int tempM[][])
    {
        temp1 = 1;
        counter = 1;
        for(int i = 0; i<rows; i++)
        {
            for(int j = 0; j<cols; j++)
            {
                tempM[i][j] = temp1 ;
                M[i][j] = tempM[i][j] ;
                temp1 = temp1 +2;
            }
            counter = counter +1;
            temp1 = counter;
        }
    }
}
/*end of Matrix class*/

```

Appendix XIII TokenServer.java

```

/**
 *File name: TokenServer.java
 *It is used for Dynamic-tree algorithm. Since for different algorithms, it has a slight
 *difference among the algorithms. Here only append one TokenServer class in the
 *thesis. This class connects with all the clients, handles sending the matrices to
 *the clients respect to the requests and taking updated matrices back from
 *the clients to overwrite the original source
 */

import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.*;

/**
 * TokenServer Class
 */
public class TokenServer implements Runnable
{
    protected Socket socket,

    /**
    *Constructor with one para
    */
    public TokenServer(Socket socket)
    {
        this.socket = socket;
    }

    /**Variables used in the class*/
    protected DataInputStream oIn;
    protected DataOutputStream oOut;
    protected Thread listener;
    protected static Vector handlers = new Vector ();
    protected String mode;
    protected int portN;
    static Processes clientArray[] = new Processes[4];
    static int pOrder = -1;
    static long waitingTime = 0;
    static int clientCounter = 0;
    static InetAddress holder=null;
    static InetAddress temp = null;

    /**Static variables used for the matrix*/
    static int array1[][] = new int[16][16];
    static int array2[][] = new int [16][16];
    static int array3[][] = new int[16][16];
    static Matrix arrayA = new Matrix (16, 16, array1);
    static Matrix arrayB = new Matrix ( 16, 16, array2);

```

```

static Matrix arrayC = new Matrix ( 16, 16, array3);

/**
 *Function will be called in main
 */
public synchronized void start ()
{
    if (listener == null)
    {
        try{
            oIn = new DataInputStream
            (new BufferedInputStream (socket.getInputStream ()));
            oOut = new DataOutputStream
            (new BufferedOutputStream (socket.getOutputStream ()));
            listener = new Thread (this);
            listener.start ();
        } catch (IOException ignored) { }
    }
}

/**
 *Function stop
 */
public synchronized void stop ()
{
    if (listener != null)
    {
        try
        {
            if (listener != Thread.currentThread ())
                listener.interrupt ();
            listener = null;
            oOut.close ();
        } catch (IOException ignored) { }
    }
}

/**
 * Function run
 */
public synchronized void run ()
{
    try
    {
        handlers.addElement (this);
        while (!Thread.interrupted ())
        {
            AssignIDToClient();
            while(true)
            if(clientArray[3]!=null)
            handleReadWrite();
        }
    }
}

```

```

    }
    }catch (EOFException ignored) {}
    catch (InterruptedException e) {}
    catch(ClassNotFoundException c) {}
    catch (IOException ex) {
        if (listener == Thread.currentThread ())
            ex.printStackTrace ();
    } finally {
        handlers.removeElement (this);
    }
    stop ();
}

static int j = 0;
static int computerID = 0;
/**
 * Assigns the ids and send the neighbors' port
 * number to each client.
 */
protected void AssignIDToClient()throws IOException,ClassNotFoundException
{
    oOut.writeInt(j);
    oOut.flush();
    clientArray[j]=new Processes(socket.getPort(),1000,
        socket.getInetAddress(), socket.getLocalAddress());
    j++;
    if(j<4)
        assignNeighbor(clientArray[j]);
}

/**
 * Sends the source to the read for reading.
 */
protected synchronized void sendFileToReader() throws IOException,
    InterruptedException
{
    for(int i = 0;i<arrayA.rows;i++)
        for(int j = 0;j <arrayA.cols; j++)
        {
            oOut.writeInt(arrayA.M[i][j]);
            oOut.flush();
        }
    for(int i = 0;i<arrayB.rows;i++)
        for(int j = 0;j <arrayB.cols; j++)
        {
            oOut.writeInt(arrayB.M[i][j]);
            oOut.flush();
        }
    for(int i = 0;i<arrayC.rows;i++)
        for(int j = 0;j <arrayC.cols; j++)

```

```

        {
            oOut.writeInt(arrayC.M[i][j]);
            oOut.flush();
        }
    }

/**
 * Function sendFileToWriter send the source to each client and read the
 * modified data from them
 */
protected synchronized void sendFileToWriter() throws IOException,
    InterruptedException
{
    for(int i = 0;i<arrayA.rows;i++)
        for(int j = 0;j <arrayA.cols; j++)
        {
            oOut.writeInt(arrayA.M[i][j]);
            oOut.flush();
        }
    for(int i = 0;i<arrayB.rows;i++)
        for(int j = 0;j <arrayB.cols; j++)
        {
            oOut.writeInt(arrayB.M[i][j]);
            oOut.flush();
        }
    for(int i = 0;i<arrayC.rows;i++)
        for(int j = 0;j <arrayC.cols; j++)
        {
            oOut.writeInt(arrayC.M[i][j]);
            oOut.flush();
        }
    for(int i =0;i<arrayC.rows; i++)
        for(int j = 0;j<arrayC.cols; j++)
            arrayC.M[i][j]= oIn.readInt();
}

/**
 * Response to the client for the readers and writers for different requests
 */
protected synchronized void handleReadWrite() throws IOException,
    InterruptedException
{
    String getmode = "UNDEF";
    getmode =oIn.readUTF();
    if(getmode.equals("READ"))
        sendFileToReader();
    else if ( getmode .equals("WRITE"))
        sendFileToWriter();
    else
        System.out.println("***WRONG***" );
    pOrder = oIn.readInt();
}

```

```

        System.out.println("VALUE---" + pOrder);
        pOrder = pOrder + 1;
        broadcast(pOrder );
    }

    /**
     * Function broadcast sends the neighbors' port numbers to each client
     */
    protected void broadcast (int k)
    {
        synchronized (handlers)
        {
            Enumeration enum = handlers.elements ();
            while (enum.hasMoreElements ())
            {
                TokenServer handler = (TokenServer) enum.nextElement();
                try
                {
                    handler.oOut.writeInt(k);
                    handler.oOut.flush();
                }catch (IOException ex) {
                    handler.stop ();
                }
            }
        }
    }

    /**
     * Function broadcast sends the neighbors' port numbers to each client
     */
    protected void assignNeighbor (Processes pNext)
    {
        synchronized (handlers)
        {
            Enumeration enum = handlers.elements ();
            while (enum.hasMoreElements ())
            {
                TokenServer handler = (TokenServer) enum.nextElement();
                try {
                    if(clientArray[3]!= null )
                    {
                        handler.oOut.writeInt(pNext.pID);
                        handler.oOut.flush();
                    }
                }catch (IOException e){
                    handler.stop ();
                }
            }
        }
    }
}

```

```
/**
 * Function main
 */
public static void main (String args[]) throws IOException
{
    arrayA.makeMatrix(array1);
    arrayB.makeMatrix(array2);
    holder = temp;
    ServerSocket server = new ServerSocket (1000);
    while(true)
    {
        Socket client = server.accept ();
        temp = client.getInetAddress ();
        if(!temp.equals(holder))
            computerID ++;
        holder =temp;
        System.out.println ("Accepted from " +temp);
        TokenServer theServer = new TokenServer(client);
        theServer.start();
    }
}
}
} //end of TokenServer class
```

REFERENCE

1. [BaCh96] S. Banerjee and P. Chrysanthis. A New Token Passing Distributed Mutual Exclusion Algorithm. *Proceedings of the Intl. Conf. on Distributed Computing Systems (ICKCS)*, 1996.
2. [BWRL02] J. Bishop, B. Worrall, K. Renaud and J. Lo. Java and Distribution of Application Requiring Mutual Exclusion and Deadlock Detection. *Technical Report*, November 2002.
3. [Chan96] Y. I. Chang, "A Simulation Study on Distributed Mutual Exclusion", *Journal of Parallel and Distributed Computing*, vol. 33, no. 2, pp. 107-121, March 1996.
4. [DaHa02] P. K. Dash and R. C. Hansdah. "An Efficient Token-Based Algorithm for Distributed Mutual Exclusion." *International Council for Computer Communication*, pp. 955-971, 2002.
5. [Fidg91] C. Fidge. "Logical Time in Distributed Computing Systems." *Computer*, vol. 24, pp. 28-33, 1991.
6. [FuTL97] S. Fu, N. Tzeng, and Z. Li. "Empirical Evaluation of Distributed Mutual Exclusion Algorithms." *International Parallel Processing Symposium*, pp. 255-259, 1997.
7. [Hadd04] F. Haddix. "Tree-based Mutual Exclusion with Fairness." Unpublished manuscript, 2004.
8. [John95] T. Johnson, "A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms", *Proc. 9th Int. Parallel Processing Symp.*, pp. 258-264,

April 1995.

9. [Lamp78] L. Lamport. "Time, Clocks and Ordering of Events in Distributed Systems." *Communication of the ACM*, vol. 21, no. 7, pp. 558-565, June 1978.
10. [LiHu89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *Communication of the ACM*, vol. 21, no. 7, pp. 321-359. November 1989.
11. [Maek85] M. Maekawa. A N Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145-159, May 1985.
12. [Muel98] F. Mueller. "Prioritized Token-based Mutual Exclusion for Distributed Systems." *International Parallel Processing Symposium*, pp. 791-795, 1998.
13. [Muel97] F. Mueller. Distributed shared-memory threads: DSM-threads. *In Workshop on Run-Time Systems for Parallel Programming*. vol. 39, pp. 31-40, April 1997.
14. [NaTr96] M. Naimi, M. Trehel, and A. Arnold. "A $\log(N)$ Distributed Mutual Exclusion Algorithm Based on Path Reversal." *JPDC: Journal of Parallel and Distributed Computing*, vol. 34, no. 1, pp. 1-13, April 1996.
15. [NeMi91] M. L. Neilsen and M. Mizuno, "A Dag-Based Algorithm for Distributed Mutual Exclusion," *Proc. 11th Int. Conf. Distributed Computer Systems*, pp. 354-360, May 1991.
16. [Raym89] K. Raymond, "A Tree based Algorithm for Distributed Mutual Exclusion." *ACM Trans. On Computer Systems*, vol. 7, no. 1, pp. 61-77, February 1989.

17. [SuKa85] I. Suzuki and T. Kasami, "A distributed Mutual Exclusion Algorithm." *ACM Transactions on Computer Systems*, vol. 3, no. 4, pp. 344-349, November 1985.
18. [TrHo01] M. Trehel and A. Housni. Comparison of Techniques used in Prioritized Mutual Exclusion by Groups. *PDCAT*, 2001.
19. [WaMu00] C. Wagner and F. Mueller. "Token-Based Read/Write-Locks for Distributed Mutual Exclusion." *A. Bodc et al. (Eds.), Euro-Par 2000, LNCS 1900*, pp. 1185-1195, 2000.
20. [XuHw96] A. Xu and K. Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel & Distributed Technology*, Spring 1996.

VITA

Yunhong Jiang was born in Shunchang, Fujian, P.R. China, on August 18, 1976, the daughter of ShaoTai Jiang and Yadian Zheng. After completing her study at Shunchang High School, Shunchang, Fujian, in 1994, she entered Fujian Normal University. She received the degree of Bachelor of Arts from Fujian Normal University in July 1998. After two years as a graduate student in Music at Fujian Normal University, she moved to United States to study Computer Science at Texas State University-San Marcos.

Permanent Address: 1620 W. 6th, Apt. C

Austin, Texas 78703

This thesis was typed by Yunhong Jiang.