PERFORMANCE COMPARISON OF FOUR TOKEN RINGS IN 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 of

Master of SCIENCE

By

RONG WANG, B.A.

San Marcos, Texas

December 2003

ACKNOWLEDGMENTS

First of all, I am deeply indebted to my thesis supervisor Dr. Furman Haddix, who has devoted so much time and effort in teaching me and guiding me to finish this research work. Without his great encouragement and support, this thesis would have never been finished.

I would also like to thank Dr. Jawad Drissi and Dr. Wuxu Peng. I am very grateful to them for their great kindness and support on my thesis work.

Finally, I want to express my deepest thanks to my family for their great love and support to me.

TABLE OF CONTENTS

List of Tables

List of Figures

ABSTRACT

The concept of self-stabilization was first proposed by Dijkstra. A self-stabilization token ring can make guarantee for a system to recover to a legal state in finite time, regardless of what illegal state the system is in. This property makes it a very preferable way to tolerate arbitrary transient faults.

The thesis implements 4 self-stabilizing token ring algorithms, and compares their performance in terms of stabilization time and token delivery time in a distributed computing environment using a simulated application as the test bed.

CHAPTER 1 INTRODUCTION

1.1 Background

A distributed system is composed of a collection of processes as well as a collection of communications between two processes. There exists the mutual exclusion problem in such a system, and there are several ways to solve it.

One way of solving this problem is to collect the states of all the other processes. But there is the problem of heavy network traffic in this solution because each process must ask for the state of each other process and send its own state to all other processes. The huge amount of messages to be handled will make this method a non-practical solution.

Another way of solving this problem is to ask a central monitor for permission to access the critical section. But this solution also has shortcomings because using an external monitor can delay the recovery beginning time when an error occurs, and the monitor itself can also crash.

A preferable method to solve the problem is to let each process in the system determine its next activity by observing the current states of its neighbors before entering its critical section. Using this approach, traffic is minimal and exponential increases inflating with system expansion is avoided.

Since various faults inherent to modern distributed system may occur, the quality of a distributed system also depends on its tolerance to the potential faults. Tolerance to arbitrary transient faults is one of the most important requirements in such systems.

Various tolerance methods to such transient faults have been proposed and implemented. The concept of self-stabilization proposed by Dijkstra in 1974[Dijk74] is the most general technique.

Self-stabilization means that starting from an arbitrary initial state, it is guaranteed that the system will reach a desirable state in finite time, and if it is in a desirable state, it will remain in desirable states thereafter. That means if a transient fault leads a stabilizing system to an undesirable state, further execution of the system will be guaranteed to return to a desirable state, and will stay there in the absence of additional transient faults.

Token rings are a straightforward way of providing mutual exclusion in a physical or logical ring configuration of processes. Since Dijkstra proposed the first self-stabilizing token ring [Dijk74], Many variations have been proposed. In [Dijk74], no application of self-stabilization to fault tolerance was mentioned, but considerable effort has been done to use self-stabilization as a technique for fault tolerance in the area of distributed computing, this effort can be seen in [BYC88], [BYZ89], [YBL91], and [BY93].

1.2 Goals

As mentioned above, there exist quite a number of proposals regarding token rings to be used in distributed systems. In this thesis work, four token ring algorithms, namely, enabler token rings, alternator token rings, shepherd token rings and companion token rings, are considered. The four token ring algorithms are all self-stabilizing algorithms, but their performance is not well known.

The goal of this thesis is to compare the performance of these four token rings in a simulated distributed computing environment. The comparison of the stabilization time and token delivery time of the four algorithms will be especially focused on.

1.3 Contribution

- Simulated multiple processors in a distributed environment
- Designed an algorithm to create a static logical ring configuration of processes
- Developed an application system to use as bench mark
- Designed a method to give each token ring algorithm exactly same test environment
- Implemented the four token ring algorithms for mutual exclusion between writing processes

- Evaluated the performance of the four token ring algorithms on the stabilization time and privilege delivery time.
- Evaluated the performance of the four token ring algorithms under different levels of contention for the privilege of executing a critical section.

1.4 Thesis Organization

Chapter 1 of the thesis is an overview and introduction to this thesis work as a whole. Chapter 2 gives basic introduction to distributed systems and it also introduces the importance of transient fault tolerance in distributed systems. This chapter should be of interest to readers who are new to the field of distributed systems. Chapters 3 through 6 briefly introduce the four token ring algorithms. Chapter 7 describes the implementation for each token ring algorithm introduced in the previous 4 chapters. Chapter 8 describes the application that is used as the application to test the token ring algorithms, and the implementation for it is also covered in this chapter. Chapter 9 explains how the performance of each token ring algorithm is measured. It also explains the methods used to make sure that the four algorithms are given identical test environments. In chapter 10, the performance results of each token ring algorithm are observed, compared and analyzed. Chapter 11 discusses the conclusions and potential future work.

CHAPTER 2 FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

2.1 The Definition of Distributed Systems

There are many definitions of distributed system, and they do not completely agree with each other, but in general, it is safe to say that a distributed system is composed of independent processors that have no shared memory, that these independent processors cooperate with each other and are linked by a high speed network, and that these facts are transparent to the user of the system. The user should be able to use such a system as if using a single system.

2.2 The Advantages of Distributed Systems

There are many advantages in using a distributed system, here listed are some of them.

- It is more powerful in computing than a single processor system
- It can share data easily
- It allows more processors to be added to meet increasing demands.
- It should be more reliable than a single system

Nowadays, computers are used almost everywhere, and almost every

large computer-based system is a distributed one. Distributed systems are becoming more and more important in both research and commercial fields.

2.3 Transient Faults in Distributed Systems

As described in section 2.2, distributed systems are powerful and important nowadays and there are advantages in using them, but that does not mean there are no problems to be solved. Transient faults are one of the important issues concerning such systems.

From the definition of distributed system, we know that a distributed system is composed of a collection of processors, and the processors of the participating nodes have no shared memory. Hence the only way for them to exchange information is to pass messages between each other through a network. This limitation introduces system faults caused by the occurring of errors during message passing, e.g. message loss, and this kind of faults are called transient faults. A practical distributed system must be able to recover from such transient faults quickly, and the quality of a distributed system depends on how fast it can recover from such faults.

2.4 Self-Stabilization as the Proposal for Transient Faults

An external monitor of system states can be used to help a distributed system recover from a transient failure, but it is preferable if a distributed

system begins its recovering process as soon as an error occurs, and such a method does not provide this capability.

The concept of self-stabilization is the most general tolerance method for handling arbitrary transient faults. There is no universally agreed upon formal definition for self-stabilization, however, it is generally acknowledged initial definition was from [Dijk74], which was already described in chapter 1. For more definitions about self-stabilization, [Dijk82a], [Dijk82b], [LL90], and [BGM93] can be referred. The ring topology is commonly used in distributed computing and network protocols [Herman98], and the subject of token rings is a straightforward way of providing mutual exclusion in a physical or logical ring configuration of processes.

A self-stabilizing token ring is useful for a distributed system where transient failures may occur and the system may enter an arbitrary illegal state after such transient failures [PV2000]. It is able to start recovering automatically as soon as it detects a fault in the system and is able to provide a high degree of fault tolerance.

For a distributed system, at any particular time, it will be either at a legal state or at an illegal state. For a self-stabilizing token ring algorithm, given an initial state (no matter if it is legal or illegal), the system can reach a legal state after a finite number of state transitions---if the initial state is an illegal state, the system is guaranteed to reach a legal state in finite time; if the initial state is a legal state, the system will remain in legal states thereafter. If an error

occurs, again, the system will return to a legal state after a finite number of state transitions.

Since Dijkstra proposed the first self-stabilization ring [Dijk74], many variations have been suggested. In this thesis, four of them are considered in the following chapter. These four token ring algorithms are detailed in chapter 3, chapter 4, chapter 5 and chapter 6, respectively.

CHAPTER 3 THE ENABLER TOKEN RING

3.1 Introduction

This is the first stabilizing, unidirectional, deterministic token ring where each process has a constant number of states.

Since Dijkstra proposed the first self-stabilizing, unidirectional token ring in 1974 [Dijk74], many variations were suggested. But all those introduced before the enabler token ring are not deterministic or have a much larger state space.

In the enabler token ring, the states are legal when there is exactly one token circulating around the ring, and the states are illegal when there is more than one circulating token. The construction of token ring precludes a state in which no token exists. Each process in this token ring has three Boolean variables and three actions. Each action in a process is of the form:

<guard> → *<multiple assignment statement>*

Where the guard is a Boolean expression over the variables of a process and its left neighbor. The conjunction of the guards of any two actions

in the same process is false, in other words, at most one enabled action from each process can be executed at a time. Hence the processes in this token ring are deterministic.

3.2 Algorithm for Enabler Token Ring

Suppose a ring system is composed of n processes, and each process P[i] (0 <= i < n) has three Boolean variables named e.i, tkn.i and ready. The variable e.i and tkn.i can be read by the right neighbor of process i, while ready is a local variable used by process i only.

The algorithm for enabler token ring is as the following:

Case 1: if 0 < i < n

$$X.i: \quad e.i \neq e.(i-1) \wedge tkn.i = tkn.(i-1) \rightarrow e.i, ready := \neg e.i, false$$

$$Y.i: \quad e.i \neq e.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge \neg tkn.i \wedge \neg ready \rightarrow e.i, ready := \neg e.i, true$$

$$Z.i: \quad e.i \neq e.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (tkn.i \vee ready) \rightarrow e.i, tkn.i, ready := \neg e.i, \neg tkn.i, false$$

Case 2: if i = 0

$$X.0: \quad e.0 = e.(n-1) \wedge tkn.0 \neq tkn.(n-1) \rightarrow e.0, ready := \neg e.0, false$$

$$Y.0: \quad e.0 = e.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge \neg tkn.0 \wedge \neg ready \rightarrow e.0, ready := \neg e.0, true$$

$$Z.0: \quad e.0 = e.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (tkn.0 \vee ready) \rightarrow e.0, tkn.0, ready := \neg e.0, \neg tkn.0, false$$

In the algorithm, each process has an X action, a Y action and a Z action. The following is the explanation for these actions:

- An X action allows an enabler to be passed from process p[i] to the next process p[i+1 mod n] when p[i] has no token;
- A Y action allows an enabler to be passed from process p[i] to the next process p[i+1 mod n] when p[i] has an F token and its variable ready is false. Its variable ready becomes true in this action;
- A Z action allows both an enabler and a token to be passed from process p[i] to the next process p[i+1 mod n] when p[i] has an F token and its variable ready is true, or when it has a true token. Its variable ready becomes false in this action.

A state of the ring is legal iff in that state exactly one process has a token and at least one process has an enabler.

3.3 Algorithm When Combined with Application

When combined with application, two additional bits are needed, one of which is set by the token ring and read by the application, and the other is set by the application and read by the token ring:

Case 1: if 0 < i < n

$$X.i: \quad e.i \neq e.(i-1) \wedge tkn.i = tkn.(i-1) \rightarrow e.i, ready := \neg e.i, false$$

$$Y.i: \quad e.i \neq e.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge \neg tkn.i \wedge \neg ready \rightarrow e.i, ready := \neg e.i, true$$

$$Z1.i: \quad e.i \neq e.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (tkn.i \vee ready) \wedge \neg req.i \rightarrow e.i, tkn.i, ready, grt.i := \neg e.i, \neg tkn.i, false, false$$

$$Z2.i: \quad e.i \neq e.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (tkn.i \vee ready) \wedge req.i \rightarrow grt.i := true$$

Case 2: if i = 0

$$X.0: \quad e.0 = e.(n-1) \wedge tkn.0 \neq tkn.(n-1) \rightarrow e.0, ready := \neg e.0, false$$

$$Y.0: \quad e.0 = e.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge \neg tkn.0 \wedge \neg ready \rightarrow e.0, ready := \neg e.0, true$$

$$Z1.0: \quad e.0 = e.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (tkn.0 \vee ready) \wedge \neg req.0 \rightarrow e.0, tkn.0, ready, grt.0 := \neg e.0, \neg tkn.0, false, false$$

$$Z2.0: \quad e.0 = e.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (tkn.0 \vee ready) \wedge req.0 \rightarrow grt.0 := true$$

In above algorithm, variable grt is set by the token ring and read by the application, while variable req is set by the application and read by the token ring.

Action Z1 is executed when the corresponding process has the privilege but variable req is false, and action Z2 is executed when the corresponding process has the privilege and variable req is true. [GH1996]

CHAPTER 4 THE RING ALTERNATOR TOKEN RING

4.1 Introduction

An alternator is an array of interacting processes that can be used in transforming a stabilizing system executed serially into a stabilizing system executed concurrently.

4.1.1 The Lower Level Protocol

A ring alternator is an array of n+1 processes p [i: 0...n], where n >= 2. Process p[0] and p[n] are actually one same process---the first process in the ring. When this first process communicates with its right neighbor p[1], it is p[0], while when it communicates with its left neighbor p[n-1], it is p[n]. Each other process has a left neighbor p[i-1] and a right neighbor[i+1].

The lower level algorithm:

$$X.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \rightarrow b.0, b.n, T := \neg b.0, \neg b.n, false$$

$$Y.0: \quad b.0 = b.1 \wedge b.n = b.(n-1) \wedge SysClock - Start > TimeOut \wedge T \rightarrow b.0, T := \neg b.0, false$$

$$Z.0: \quad b.0 \neq b.1 \wedge b.n \neq b.(n-1) \wedge SysClock - Start > Timeout \wedge T \rightarrow b.n, T := \neg b.n, false$$

$$W.0: \quad ((b.0 = b.1 \wedge b.n = b.(n-1)) \vee (b.0 \neq b.1 \wedge b.n \neq b.(n-1)) \wedge \neg T \rightarrow Start, T := SysClock, true$$

$$U.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \rightarrow b.i := \neg b.i$$

Where i refers to the regular processes, whose IDs are ranging between 0 and n, not included.

For the first process p[0] in the ring alternator, there are 3 Boolean variables and a timeout mechanism. The 3 Boolean variables are b.0, b.n and T, where T is the local variable indicating if timeout in progress, while b0 and b.n can be read by process p[1] and process p[n-1], respectively.

For other processes, each of them has one Boolean variable b.i, which can be read by both of its neighbors.

X.0, Y.0, Z.0 and W.0 are the 4 possible actions that can be taken by process p[0], and U.i is the possible action that can be taken by all the other processes.

4.1.2 The Higher Level Protocol

In the token level, each process has 2 Boolean variables, which are tkn.i and ready. The variable ready is a local variable for each process, and the variable tkn.i can be read by the right neighbor of each process. Each action in a process is of the form:

<guard> → *<multiple assignment statement>*

Where the guard is a Boolean expression over the variables of a process and its neighbors. The conjunction of the guards of any two actions in

the same process is false, in other words, at most one enabled action from each process can be executed at a time. Hence the processes in this token ring are also deterministic.

In the ring, the states are legal when there is exactly one token circulating around the ring, and the states are illegal when there is more than one circulating token. The construction of the ring precludes a state in which no token exists.

4.2 Algorithm for Ring Alternator Token Ring

Suppose a ring system is composed of n processes, where n is greater or equal to 2, the algorithm for ring alternator token ring is as the following:

Case 1: if 0 < i < n, we have the following three possible actions:

$$X.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (ready \vee tkn.i) \rightarrow b.i, tkn.i, ready := \neg b.i, \neg tkn.i, false$$

$$Y.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge \neg ready \wedge \neg tkn.i \rightarrow b.i, ready := \neg b.i, true$$

$$Z.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \rightarrow b.i := \neg b.i$$

- An X.i action allows alternator to be passed from process p[i] to its both neighbors, and allows token to be passed to the next process p[i+1 mod n] when p[i] has the alternator, an F token and its variable ready is true, or when it has a true token. Its variable ready becomes false in this action;

- A Y.i action allows alternator to be passed from process p[i] to its both neighbors when p[i] has the alternator, an F token and its variable ready is false. Its variable ready becomes true in this action.
- A Z.i action allows alternator to be passed from process p[i] to its two neighbors;

Case 2: if i = 0

Process p[0] is a special case in the token ring, and it is much more complicated compared with other processes. The following figure listed all the possible actions for process p [0].

$$X.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (ready \vee tkn.0) \rightarrow b.0, b.n, T, tkn.0, ready := \neg b.0, \neg b.n, false, \neg tkn.0, false$$

$$Y.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge \neg ready \wedge \neg tkn.0 \rightarrow b.0, b.n, T, ready := \neg b.0, \neg b.n, false, true$$

$$Z.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \rightarrow b.0, b.n, T := \neg b.0, \neg b.n, false$$

$$U.0: \quad b.0 = b.1 \wedge b.n = b.(n-1) \wedge SysClock - Start > TimeOut \wedge T \rightarrow b.0, T := \neg b.0, false$$

$$V.0: \quad b.0 \neq b.1 \wedge b.n \neq b.(n-1) \wedge SysClock - satrt > Timeout \wedge T \rightarrow b.n, T := \neg b.n, false$$

$$W.0: \quad ((b.0 = b.1 \wedge b.n = b.(n-1)) \vee (b.0 \neq b.1 \wedge b.n \neq b.(n-1)) \wedge \neg T \rightarrow Start, T := SysClock, true$$

- Action X.0 allows alternator b.0 and token to be passed to process p[1], and allows alternator b.n to be passed to p[n-1]. Its local variables T and ready become false in this action;
- Action Y.0 allows alternator b.0 to be passed to process p[1], and alternator b.n to be passed to p[n-1]. Its local variable T becomes false

in this action, while its local variable ready becomes true in this action;

- Action Z.0 allows alternator b.0 to be passed to process p[1], and alternator b.n to be passed to p[n-1]. Its local variable T becomes false in this action;
- Action U.0 allows alternator b.0 to be passed to process p[1], and its local variables T becomes false in this action;
- Action V.0 allows alternator b.n to be passed to process p[n-1], and its local variable T becomes false in this action;
- Action W.0 sets local variable T to 1, and it also sets the start of timeout.

4.3 Algorithm When Combined with Application

When combined with application, two additional bits are needed, one of which is grt and it is set by the token ring and read by the application, and the other variable is req and it is set by the application and read by the token ring.

Case 1: if 0 < i < n, we have the following 4 possible actions:

$$X1.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (ready \vee tkn.i) \wedge \neg req \rightarrow b.i, tkn.i, grt, ready := \neg b.i, \neg tkn.i, false, false$$

$$X2.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge (ready \vee tkn.i) \wedge req \rightarrow grt := true$$

$$Y.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \wedge tkn.i \neq tkn.(i-1) \wedge \neg ready \wedge \neg tkn.i \rightarrow b.i, ready := \neg b.i, true$$

$$Z.i: \quad b.(i+1) = b.i \wedge b.i \neq b.(i-1) \rightarrow b.i := \neg b.i$$

Case 2: if i = 0

When combined with req, we can have 7 possible actions as shown below:

$$X1.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (ready \vee tkn.0) \wedge \neg req \rightarrow b.0, b.n, T, tkn.0, grt, ready := \neg b.0, \neg b.n, false, \neg tkn.0, false, false$$

$$X2.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge (ready \vee tkn.0) \wedge req \rightarrow grt := true$$

$$Y.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \wedge tkn.0 = tkn.(n-1) \wedge \neg ready \wedge \neg tkn.0 \rightarrow b.0, b.n, T, ready := \neg b.0, \neg b.n, false, true$$

$$Z.0: \quad b.0 = b.1 \wedge b.n \neq b.(n-1) \rightarrow b.0, b.n, T := \neg b.0, \neg b.n, false$$

$$U.0: \quad b.0 = b.1 \wedge b.n = b.(n-1) \wedge SysClock - Start > TimeOut \wedge T \rightarrow b.0, T := \neg b.0, false$$

$$V.0: \quad b.0 \neq b.1 \wedge b.n \neq b.(n-1) \wedge SysClock - satrt > Timeout \wedge T \rightarrow b.n, T := \neg b.n, false$$

$$W.0: \quad ((b.0 = b.1 \wedge b.n = b.(n-1)) \vee (b.0 \neq b.1 \wedge b.n \neq b.(n-1)) \wedge \neg T \rightarrow Start, T := SysClock, true$$

In above algorithm, variable grt is set by the token ring and read by the application, while variable req is set by the application and read by the token ring.

Action X1.0 and X1.i are executed when the corresponding process has the privilege but variable req is false, and action X2.0 and X2.i are executed when the corresponding process has the privilege and variable req is true. [GH1997]

CHAPTER 5 SHEPHERD TOKEN RINGS

5.1 Introduction

This is of interest because of some unique characteristics it possesses. Each process in the ring system has three variables---token, shepherd and ready. The right neighbor of each process can read the variables token and shepherd, but variable ready is the local variable used by each process itself only. Not only the presence of a token can grant a process to enter critical section, but also can the existence of a high shepherd.

In the ring, the states are legal when there is exactly one token circulating around the ring, and the states are illegal when there is more than one circulating token. The construction of the ring precludes a state in which no token exists.

5.2 Algorithm for Shepherd Token Ring

Consider a ring system that is composed of n processes, and each process i (0 <= i < n) has three Boolean variables named tkn.i, sh.i and ready. The variable tkn.i and sh.i can be read by the right neighbor of process i, while ready is the local variable used by process i only.

The algorithm for shepherd token ring is as the following:

$$X: \quad T.i \wedge \neg S.i \wedge (H.i \vee i \neq 0) \rightarrow tkn.i, ready := \neg tkn.i, false$$

$$Y: \quad T.i \wedge \neg S.i \wedge \neg H.i \wedge \neg R.i \rightarrow tkn.i, ready := \neg tkn.i, true$$

$$Z: \quad T.i \wedge \neg S.i \wedge ((H.i \wedge i \neq 0) \vee (\neg H.i \wedge R.i)) \rightarrow tkn.i, sh.i, ready := \neg tkn.i, \neg sh.i, false$$

$$W: \quad (\neg T.i \vee i = 0) \wedge S.i \wedge H.i \rightarrow sh.i, ready := \neg sh.i, false$$

Where:

$$T.i \equiv (i = 0 \wedge tkn.0 = tkn.(n-1)) \vee (0 < i < n \wedge tkn.i = \neg tkn.(i-1));$$

{token is present }

$$S.i \equiv (i = 0 \wedge sh.0 = sh.(n-1)) \vee (0 < i < n \wedge sh.i = \neg sh.(i-1));$$

{shepherd is present }

$$H.i \equiv sh.i$$

{shepherd state is high }

$$R.i \equiv ready.i$$

{shepherd is ready to be passed }

All the possible four actions grant privilege to the corresponding process.

- Action X allows the token to be passed on to the right neighbor of the process when it has the token, but not the shepherd. Its variable ready is set false in this action;
- Action Y allows the token to be passed on when the process has the token and the shepherd and ready is false and shepherd is low. Its variable ready is set true in this action;
- Action Z allows the token and shepherd to be passed on when the process has the token and the shepherd, and the shepherd is high or ready is true. Its variable ready is set false in this action;

- Action W allows the shepherd to be passed on to the right neighbor of the process when it has the shepherd, but not the token, and the shepherd is high.

5.3 Algorithm When Combined with Application

When combined with application, two additional bits are needed, one of which is set by the token ring and read by the application, and the other is set by the application and read by the token ring. Since every possible action can grant privilege to the corresponding process, we get 4 additional actions when we combine the ring with application, as shown in the following:

$$
\begin{aligned}
A1.1: \quad & T.i \wedge \neg S.i \wedge (H.i \vee i \neq 0) \wedge \neg req \\
& \rightarrow tkn.i, ready, grt := \neg tkn.i, false, false \\
A1.2: \quad & T.i \wedge \neg S.i \wedge (H.i \vee i \neq 0) \wedge req \\
& \rightarrow grt := true \\
A2.1: \quad & T.i \wedge \neg S.i \wedge \neg H.i \wedge \neg R.i \wedge \neg req \\
& \rightarrow tkn.i, ready, grt := \neg tkn.i, true, false \\
A2.2: \quad & T.i \wedge \neg S.i \wedge \neg H.i \wedge \neg R.i \wedge req \\
& \rightarrow grt := true \\
A3.1: \quad & T.i \wedge \neg S.i \wedge ((H.i \wedge i \neq 0) \vee (\neg H.i \wedge R.i)) \wedge \neg req \\
& \rightarrow tkn.i, sh.i, ready, grt := \neg tkn.i, \neg sh.i, false, false \\
A3.2: \quad & T.i \wedge \neg S.i \wedge ((H.i \wedge i \neq 0) \vee (\neg H.i \wedge R.i)) \wedge req \\
& \rightarrow grt := true \\
A4.1: \quad & (\neg T.i \vee i = 0) \wedge S.i \wedge H.i \wedge \neg req \\
& \rightarrow sh.i, ready, grt := \neg sh.i, false, false \\
A4.2: \quad & (\neg T.i \vee i = 0) \wedge S.i \wedge H.i \wedge req \\
& \rightarrow grt := true
\end{aligned}
$$

Where:

$T.i \equiv (i = 0 \wedge tkn.0 = tkn.(n-1)) \vee (0 < i < n \wedge tkn.\ i = \neg tkn.(i-1));$

{ *token is present* }

$S.i \equiv (i = 0 \wedge sh.0 = sh.(n-1)) \vee (0 < i < n \wedge sh.\ i = \neg sh.(i-1));$

{ *shepherd is present* }

$H.i \equiv sh.\ i$

{ *shepherd state is high* }

$R.i \equiv ready.\ i$

{ *shepherd is ready to be passed* }

In above algorithm, variable grt is set by the token ring and read by the application, while variable req is set by the application and read by the token ring.

Action A1.1, A2.1, A3.1 and A4.1 will be executed when the corresponding process has the privilege but variable req is false, and action A1.2, A2.2, A3.2 and A4.2 will be executed when the corresponding process has the privilege and variable req is true. [Hadd91]

CHAPTER 6 COMPANION TOKEN RING

6.1 Introduction

The companion token ring is also a self-stabilizing token ring. But it only uses 2 bits. And logically, there are three artifacts of interest, a high token, a low token, and a companion.

A high token can always execute, companion can always execute unless a low token is present, and a low token can only execute if a companion is present.

It is always true that at least one process can always execute. In the ring, the states are legal when there is exactly one token circulating around the ring, and the states are illegal when there is more than one circulating token. The construction of the ring precludes a state in which no token exists.

6.2 Algorithm for Companion Token Ring

Again, consider a ring system that is composed of n processes, and each process i (0 <= i < n) has 2 Boolean variables named *tkn.i* and *cmp.i*. The right neighbor of process i can read these two variables.

The algorithm for companion token ring is as the following:

Case 1: if 0 < i < n

$$A1.i: \quad tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge \neg tkn.i \rightarrow tkn.i := \neg tkn.i$$

$$A2.i: \quad tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge tkn.i \rightarrow tkn.i, cmp.i := \neg tkn.i, \neg cmp.i$$

$$A3.i: \quad tkn.i = tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \rightarrow cmp.i := \neg cmp.i$$

Case 2: if i = 0

$$A1.0: \quad tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge \neg tkn.0 \rightarrow tkn.0 := \neg tkn.0$$

$$A2.0: \quad tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge tkn.0 \rightarrow tkn.0, cmp.0 := \neg tkn.0, \neg cmp.0$$

$$A3.0: \quad tkn.0 \neq tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \rightarrow cmp.0 := \neg cmp.0$$

- Action A1 allows the token to be passed on when the process has a low token and a companion.
- Action A2 allows both the token and the companion to be passed when the process has the companion and a high token.
- Action A3 allows the companion to be passed on when the process has a companion and has no token.

6.3 Algorithm When Combined with Application

When combined with application, two additional bits are needed, one of which is set by the token ring and read by the application, and the other is set by the application and read by the token ring. Since only one action can grant

privilege to the corresponding process, we get 1 additional actions when we combine the ring with application, as shown in the following:

Case 1: if 0 < i < n

$$A1.1.i: tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge \neg tkn.i \wedge \neg req \rightarrow tkn.i, grt := \neg tkn.i, false$$

$$A1.2.i: tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge \neg tkn.i \wedge req \rightarrow grt := true$$

$$A2.1.i: tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge tkn.i \wedge \neg req \rightarrow tkn.i, cmp.i, grt := \neg tkn.i, \neg cmp.i, false$$

$$A2.2.i: tkn.i \neq tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \wedge tkn.i \wedge req \rightarrow grt := true$$

$$A3.i: \quad tkn.i = tkn.(i-1) \wedge cmp.i \neq cmp.(i-1) \rightarrow cmp.i := \neg cmp.i$$

Case 2: if i = 0

$$A1.1.0: tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge \neg tkn.0 \wedge \neg req \rightarrow tkn.0, grt := \neg tkn.0, false$$

$$A1.2.0: tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge \neg tkn.0 \wedge req \rightarrow grt := true$$

$$A2.1.0: tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge tkn.0 \wedge \neg req \rightarrow tkn.0, cmp.0, grt := \neg tkn.0, \neg cmp.0, true$$

$$A2.2.0: tkn.0 = tkn.(n-1) \wedge cmp.0 = cmp.(n-1) \wedge tkn.0 \wedge req \rightarrow grt := true$$

$$A3.0: \quad tkn.0 \neq tk.(n-1) \wedge cmp.0 = cmp.(n-1) \rightarrow cmp.0 := \neg cmp.0$$

In above algorithm, variable grt is set by the token ring algorithm and read by the application, while variable req is set by the application and read by the token ring.

Action A1.1 and A2.1 will be executed when the corresponding process has the privilege but variable req is false, and action A1.2 and A2.2 will be executed when the corresponding process has the privilege and variable req is true.

CHAPTER 7 TOKEN RING ALGORITHM IMPLEMENTATION

Java language is used in the coding part of the thesis. Low-level UDP sockets are used.

7.1 Implementation Overview

The four token ring algorithms are implemented in a similar way, and the implementation code for them all has the same basic structure. The structure is described in two views, one is from the view of the classes, and the other is from the view of the files.

7.1.1 Class View

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

FinalVariables.class:

This class defines some constant variables used in the implementation. They are variables such as port numbers, package size and so on

ProcessInfo.class:

This class is used to store the information of every process that wants

to join the token ring, and it implements the interface Serializable so that its object can be transported over the network. This class has the following fields:

- Address: the address of the machine on which a process is running
- Lport: the port number used by a process to listen from its neighbor(s)
- NotifyNeighborPort: the port number used a process to listen from the ring maker to know who is its neighbor.
- ServerID: the id of a process

TransportTool.class:

This class is used to send and receive an object of a class through the network. In this thesis work, it is used to send and receive the objects of class ProcessInfo through the network.

RingMaker.class:

This class is used to create a logical token ring so that every process that requests to join the ring can know which process(es) it should communicate with.

RingInitialization.class:

A process that wants to join the token ring uses this class to find out which process(es) is/are its neighbor(s). This class communicates with the ring maker directly.

ListenToken.class:

This class is where each algorithm is actually implemented. Unlike other classes that are very similar, and even the same, code for this class is quite different from each other for each algorithm.

The name of this class is also slightly different from each other according to the different name of each algorithm in order to distinguish them. For example, the listen token class for companion token ring algorithm is CompanionToken.class, while the listen token class for enabler token ring algorithm is EnablerToken.class, and so on.

7.1.2 File View

For each token ring algorithm, it contains several files as described in the following:

TransportTool.java:

This is a utility file written to transport a class object over the network

FinalVariables.java:

This is also a utility file used to define constant variables.

RingMaker.java and RingInitialization.java:

These two files are used to set up the ring. In this thesis work, only static token ring is considered.

ListenToken.java:

This file implements each token ring algorithm.

7.2 Implementation Details:

7.2.1 Ring Maker

Ring maker is the name of the server used to establish a logical ring in this implementation.

There is algorithm on how to form a logical ring in any connected distributed system. The algorithm described below starts with forming a spanning tree first:

1. Elect a leader as the root of the tree
2. The root multicasts to each neighbor and each neighbor becomes a child of the root.
3. Each neighbor N multicasts to its neighbors Mi
4. Each neighbor Mi replies to the first message it receives, identified by the root. Mi is then a child of the sender of the first message.
5. Repeat steps 3 and 4 until every node has received at least one message.

After the spanning tree is formed, a depth-first traversal of the spanning tree will be performed to form the logical ring.

1. Each time an actual node is visited, add a logical node to the logical ring
2. The root will be virtual node 0

3. When all the branches of the root have been traversed, the logical ring is completed.

In this research, multiple processors are simulated using four computers, and only static token rings are considered. When we simulate 8 processors, it is natural to let each computer simulate 2 processors; while when we simulate 16 processors, it is natural to let each computer simulate 4 processors, so on and so forth. The Figure 1 shows an example of simulating 16 processors on 4 computers.

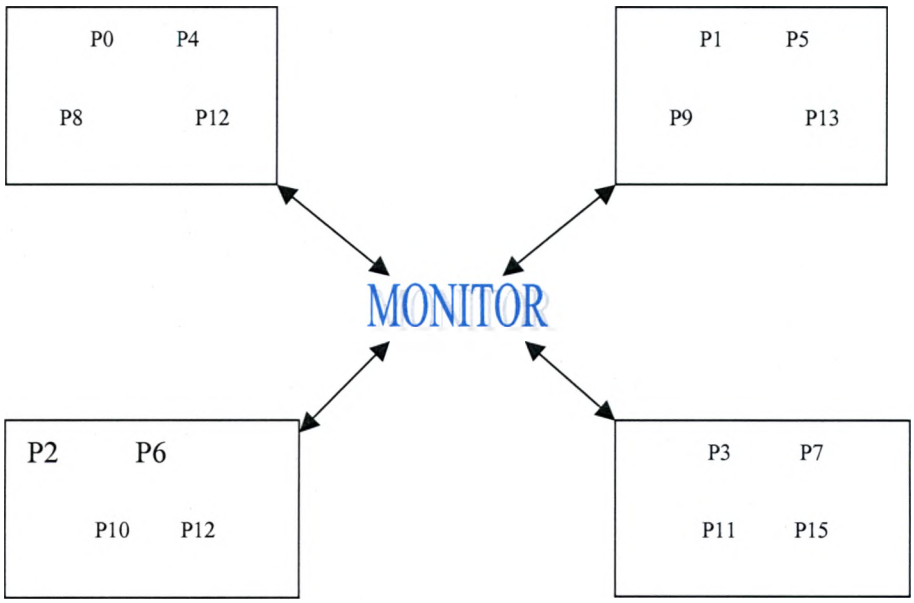

Figure 1: Simulating 16 processors on 4 computers

To make the simulation look more like a distributed system, two neighbors will not be allowed to run on one same machine. For the sake of simplicity, instead of using the procedure described above to form a logical token ring, I used a simple algorithm to create a logical ring.

4. The four computers used to simulate multiple processors are given an id of 0, 1, 2, 3, respectively. Each simulated processor is also given an ID according to the ID of the computer that it is running on. The formula to calculate ID for each simulated processor is:

   Computer ID + *c* * *i* (where 0 < i < number of processors simulated on that computer, and c is the number of actual computers used for simulation)

   According to this formula, if we simulate16 processes on 4 computers, the IDs of the 4 processors simulated on computer with ID 0 will be 0, 4, 8 and 12, the IDs of those on computer with ID 1 will be 1, 5, 9 and 13, the IDs of those on computer with ID 2 will be 2, 6, 10 and 14, and finally, the IDs of those on computer with ID 3 will be 3, 7, 11 and 15, as indicated in Figure 1.

And if we want to use different number of computers to simulate multiple processors, say, if we use 3 computers to simulate 15 processors, the IDs will be assigned as shown in Figure 2:

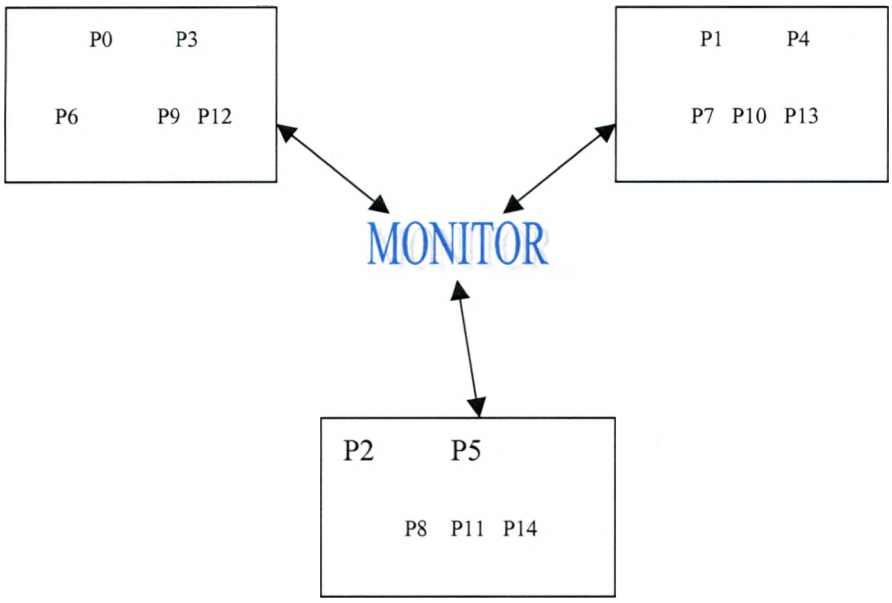

Figure 2: Simulating 15 processors on 3 computers

4. Each simulated processor will send its ID to the ring maker server when it joins the ring.

4. The Ring maker server assigns neighbors according to the ID of each simulated processors. Since step 1 guarantees that no adjacent IDs can be assigned to simulated processors that run on the same machine, we can now assign simulated processors with adjacent IDs as neighbors without worry. Again, we use simulating 16 processors on 4 computers as an example to explain. For the algorithms which require each processor to listen from one neighbor and talk to another neighbor, it will let 0 talk to 1, 1 talk to 2, ... 14 talk to 15, and15 talk to 0, which gives the result of 0 listening from 15, 1 listening from 0, ... 14 listening from 13, and 15 listening from 14, thus, avoiding neighbors running on same computer. The following figure shows the

scenario. In Figure 3, C represents computer, and P represents simulated processors.

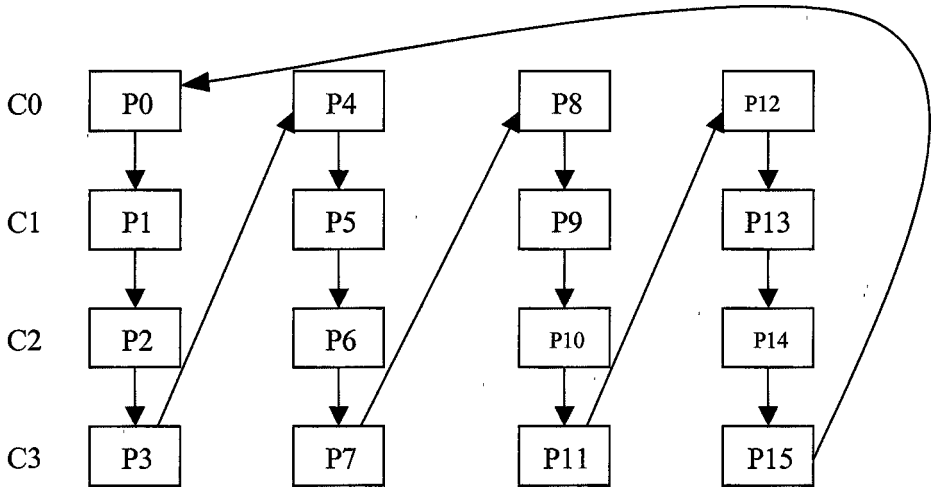

Figure 3: Assigning neighbors in one direction

4. For alternator token ring algorithms, which, unlike the other three algorithms, require every simulated processor to talk to and listen from two neighbors, it is also easy to assign neighbors. We still use 16 simulated processors on 4 computers as an example. In this case, the ring maker server will assign 15 and 1 as 0's neighbors, 1 and 3 as 2's neighbors, ... 13 and 15 as 14's neighbors, and 14 and 0 as 15's neighbors. This scenario is depicted in the following figure, which is similar to the above figure except that two directional arrows are used. Again, in the figure, C represents computer, and P represents simulated processors.

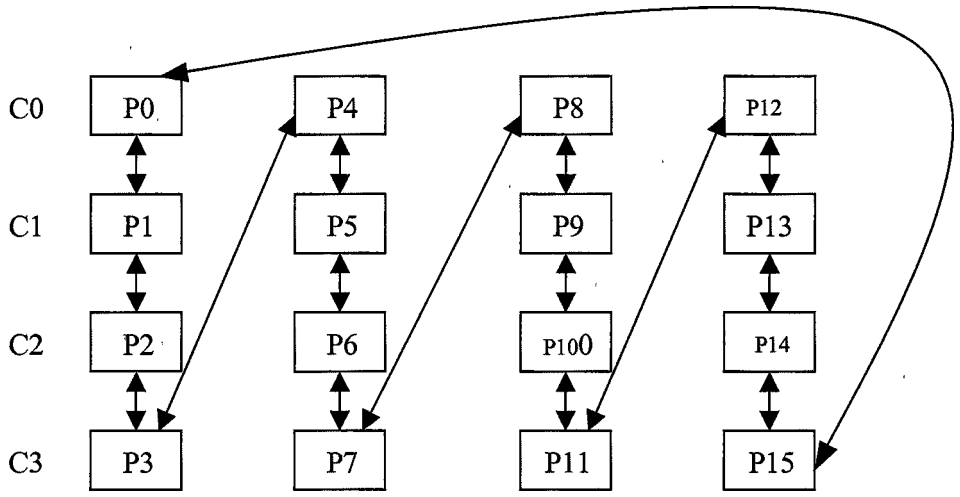

Figure 4: Assign neighbors in two directions

7.2.2 Listen Token

For each token ring algorithm, there are two methods provided in the listen token class. Both of the methods implement the token ring algorithm, and the only difference is that one of them considers the case when application is combined, while the other does not consider it. The one combined with application is called when we want to measure the waiting time for a process to get privilege after its request under the high contention, moderate contention and low contention, and it is also used when we want to measure the number of critical section executed per amount of time, while the other is used to measure the stabilization time and token delivery time for each token ring algorithm.

CHAPTER 8 APPLICATION DESCRIPTION AND IMPLEMENTATION

8.1 Application Description

A simple application used as benchmark is developed in the coding part of the thesis.

The application is a simulation of an application that asks people to select 20 favorite songs from a list of 1000 available songs. The application consists of a server named Result that processes the response of the , and multiple terminals from where people can input their choice. Each terminal will require updating the result (which is the critical section part in this application) after it receives response from a person. It is possible for several terminals to request updating at the same time, and hence, the token ring algorithms can be used to provide mutual exclusion mechanism.

The simulation used 5 computers, which are connected together locally to form a distributed computing environment during the test. All the terminals are simulated on 4 computers, and the 5th computer is used to run the code for the critical section part as well as other code pieces that are not related to the terminal simulation.

If it were a real application terminal and not a simulated one, there

should be at least 3 obvious scenarios---the terminal is idle and is waiting for somebody to use, or somebody is using it to input his choice (note that some people may finish their choice very quickly, while some other people may take a longer time to consider their choice), or the choice was done and the terminal needs to update the data (to execute the critical section). In the simulation, all that we care is about each terminal requesting access to the critical section part, and we do not care to differentiate if the terminal is idle or if somebody is using it. Therefore, we just let each simulated terminal sleep some random time to simulate the terminal idle state as well as user's different response time. As for the responses from people, the simulation is done by generating 20 different numbers range from 1 to 1000 randomly. Here, 20 numbers represent 20 songs selected by a person, and the value of each number represents the ID for each song, ranging from 1 to 1000.

8.2 Implementation Overview

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

8.2.1 Class View

The application part contains three classes as the following:

PollInfo.class:

The application server uses this class to store the information concerned in the result. It contains only two fields----the ID of a

song and the support number for the song with that ID.

ApplicationServer.class:

This class is used to implement a server that stores the result. This server plays the critical section part in the application, hence only the terminal that gets the privilege can have access to it. What it does is to receive the response from the privileged terminal, process the response and then notify the privileged terminal that the update to the result is finished so that the privileged terminal can release its privilege.

Simulated*Terminal.class:*

This class is used to simulate terminals for the application.

8.2.2 File View

There are only two files for the application implementations.

ApplicationServer.java:

This file declares and implements class Info and class Result.

SimulatedTerminal.java:

This file corresponds to SimulatedTerminal.class.

8.3 Implementation Details

8.3.1 Application Server

The application server contains several methods. The main method waits to receive response from privileged terminals, and after it gets the response, it will call

the updateResult method, which processes the response passed by the main method in the format of String----namely, recovers the selected 20 numbers from the string, updates the support numbers for the related 20 songs, and sorts the array. After all these things are done, the main method will send a message to the privileged terminal that the update is done so that the terminal can know that it is safe to release the privilege.

8.3.2 Simulated Terminal

When the simulated terminal starts running, it combines with the token algorithm first. After that, what it does is to wait for people to input their responses (idle time), to accept response from a person (different person has different response time), to request access to the critical section, to enter the critical section after it gets the privilege, and to release the privilege when it finishes accessing the critical section, and then it goes back to wait for another response and the cycle repeats again and again.

Note that the idle time---the time for the terminal to wait for people to use it, and the different response time used by different people are simulated by let the terminals sleep different amount of time because for the purpose of performance evaluation, our simulation does not need to differentiate idle time and response time. Also note that the responses from users are simulated by generate 20 different numbers ranging from 1 to 1000 randomly

The waiting time---the time for a terminal to get privilege after it requests for it is measured by each terminal and the average waiting time can be calculated.

CHAPTER 9 CODE FOR PERFORMANCE MEASUREMENT

For the comparison part of this thesis, the focus is on the comparison of stabilization time and token delivery time of the four token ring algorithms. For the stabilization time measurement, each ring is started from an arbitrary illegal state. And for the token delivery time measurement, the average time for a process to get privilege again since the last time it released privilege is measured.

The time used for each algorithm to finish certain amount of critical section executions is also measured.

9.1 Stabilization Time Monitor

This is a server that is used to record stabilization time of a token ring. The algorithm for this server was designed in the following way:

1. The server will first receive a message from ring maker to know the time when the token ring is formed.
2. The token ring is set to start at an arbitrary illegal state. Each participating process in the token ring will send a message to the stabilization time monitor each time it receives the privilege.
3. When the stabilization time monitor receives the first message from a

privileged process, it records the ID of that process in a variable named firstSender. It also records the time that the first message is received in a variable named startDelivery. It always assumes that the token ring has been stabilized, and therefore, it expects to receive the next message from firstSender's next neighbor N. If the next message it receives is from the process that it expects, then it will expect to receive the next message from N's next neighbor N+1, and again if it does receive the next message from N+1, then it will expect to receive the next message from M's next neighbor N+2, and this repeats until the message is expected to be received from firstSender again and it does receive from firstSender.

4. If the message is received from a process that the server does not expect, that indicates the existence of an illegal state. The server will change the value of firstSender to the ID of the process that just sent a message, and it also records the time the message is received in variable startDelivery. The cycle in step 2 repeats again.
5. The server broadcasts a message to all processes to notify them of the stabilization of the token ring and each privileged process will not send message to the server after receiving the broadcast message.
6. The server calculates the stabilization time by using the following formula:

   Stabilization time = startDelivery – (the time when the ring is formed)

When the token ring finally stabilized, the report of privilege will be something that looks like the following: 1 7 2 3 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8, where the first 8 is the last out-of-order report. When the server calculates the time used for stabilization, snapshot approach is used to make a valid measurement. In the case of 1 7 2 3 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8, the first report by 8 would indicate the achievement of stabilization, even though we are not confident about it until later time when the second 8 is received.

It seems simpler if we let the server broadcasts request for state to all ring processes, and the processes send states back to the server, and then the server inspects the states to see if there is only one privilege exists. If more than one exists, the server will broadcast again until there is only one privilege. But on reflection, we see that this method is not quite feasible. Since we are doing real time package communications, once the server broadcasts a request for state report, we cannot guarantee that all the clients can receive the request at the same time and report the states at the same time. Suppose the token ring has already stabilized when process 1 got the broadcast request, it has the privilege, and will report that to server, meanwhile it releases the privilege to process 2, process 2 may get the state report request till then, and it will also report privilege to server. In this case, the server will see two privileges while actually there is only one. Therefore, this method was not adopted in the implementation.

9.2 Token Traversal Time Monitor

This is a server that can be used to measure the time for the privilege to traverse a cycle under different degrees of contention, namely, the time for a process to get the privilege again after it releases the privilege last time. In theory, the token traversal time will be:

$$T = (n-1)*a + k*cs$$

Where *T* is the token traversal time, *n* is the number of ring processes, *a* is the arbitration time for each algorithm, *k* is the number of critical section executed in the traversal cycle ($0 \leq k \leq n-1$), and *cs* is the time used to execute critical section. Note that when *k* is equal to 0, which occurs when no critical section is executed during the token traversal cycle, *T* is the pure token delivery time with no interference from outside.

This server can measure the exact value of *T*, and how many times the critical section is executed during the time *T*, which is the value of *k* in the above formula.

To measure T and how many times the critical section is executed in a cycle, the algorithm is designed as the following:

1. Each process will send a message "get" to the server when it gets the privilege, send a message "cs" when it starts executing the critical section, and send a message "release" when it releases the token.
2. Since we need to measure the time for a process to get the privilege again after it releases it last time, when the server receives "release", it

will begin to record it as the time that the traversal starts, and when it receives "get" from the same process, it will consider it as the time that the traversal ends During this period, if it receives "cs", it will update the count of critical section executed by adding 1.

3. The server keeps a two-dimension array of class MeasureInfo objects, which is measureArray[r][p], where *r* is the measure size for each process, and *p* is the number of ring processes. The MeasureInfo object stores the information about the time *T* and how many times the critical section has been executed in a token traversal cycle. For example, if it is the first time to measure the token traversal time for process P3, then the information measured will be stored in measureArray [0][3], and if it is the third time to measure the token traversal time for P11, then the information measured will be stored in measureArray [2][11], and so on.
4. The server can control how many times to measure the token traversal time for each process by changing the value of *r* in step 3, and calculates the arbitration time for each algorithm according to the formula described above.
5. The server is also used to calculate how much time is used for each algorithm to finish certain amount of critical execution executions. This is useful when we compare the performance of the 4 algorithms combined with the simulated application.

9.3 Contention Control

To compare the performance of each algorithm fairly, we need to give each algorithm an identical application. We need to ensure that every algorithm has the same test environment. That means, the application combined with different token ring algorithms must do exactly the same task with no variations.

9.3.1 Method to Ensure Identical Test Environments

Consider the actions of the application in the scenario we have created. Simply stated, the application will be in one of the following 4 states:

- Idle (no current uses)
- Executing non-critical section
- Waiting for privilege to execute critical section
- Executing critical section

We can re-characterize these actions in terms of token interest as follows:

- No interest in token (idle or executing non-critical section)
- Waiting for token (performance comparison)
- Holding token (while executing critical section)

The time used to wait for token is what we want to compare between different algorithms. As for the idle time and the non-critical section time, we simulate them by letting the simulated terminal sleep randomly. But to give

each token ring algorithm identical test conditions, we also need to make sure that the sleeping time for each algorithm is identical. To do this, the following method is used:

1. Generate a two-dimension integer array SleepingTime[t][n], where *t* is the maximum rows of the array, and *n* is the number of terminals. Hence, each column in the array corresponds to a terminal so that when there are 16 processes, there will be 16 columns.
2. The array is filled with integer numbers generated randomly---- these numbers in the array indicate the time the processes should sleep before request for critical section-----this sleep is used to simulate the terminal idle time and the different response time used by different users.

   The array will be something that looks like the following after it is filled with randomly generated integers:

21	0	122	3	51	10	421	425	12	153	15	14	19	9	531
12	122	18	564	122	18	654	33	21	15	753	531	16	531	531
14	13	155	15	20	564	8	421	424	531	538	17	541	15	0
111	134	155	4	153	11	351	12	20	19	135	21	531	10	51

Table 1: Sleeping Time

Table 1 is just an example, and the real array used will be filled with numbers within range determined according to the contention degree that

we wish. Ideally, they should be specified in the consideration of the number of requests for critical section during one token traversal cycle. Say, if we consider 1 requests in one cycle as a low contention degree, then it takes totally *(1 * critical section time + token traversal time)* for the token to finish one cycle, and therefore, token traversal time should play a great part in sleeping time specification. But according to what we got from the experimental runs, the token traversal time for the 4 token ring algorithms is quite different, and it is hard to use one token traversal time to determine similar contention degree for all the token ring algorithms.

We know that in low contention states, average waiting time should be relatively constant because the number of critical section executed per token traversal of the ring is small. And on average contention level, the time a process makes a request for the token should be approximately half way around the ring. Mathematically, we can describe this as:

$$w < 1/2 * (t + cs)$$

where w is the average waiting time, t is token traversal time, and cs is critical section execution time. In high contention states, we can assume that an increase in workload will have a direct effect on system performance, as measured by waiting time. One way of viewing this is that if a process usually must wait for other processes to finish the critical section execution to execute its critical section, the contention is high. In

other words, when that point is reached, process idle time is excessive. Mathematically, we can represent this as

$w > 1/2 * t + cs,$

where w is the average waiting time, *t* is token traversal time, and *cs* is critical section execution time. This represents a fairly fine delineation, and indeed, graphically, it could be represented that moderate contention is only an inflection point between low contention, a range of values where waiting time is relatively insensitive to increase in workload, and high contention, a range of values where waiting time is increasing directly with increases in workload.

Based on the proceeding, the method used to specify the sleeping time for each terminal is as the following: first we tried different numbers to specify the sleeping time ranges, then used these ranges and ran the application with different token ring algorithms combined. Through observation, we determined the ranges that give similarly low, moderate and high contention level.

For the low contention level, we determined an average sleeping time range *ST* that gives each terminal some waiting time (the time between its request for critical section and its being granted the token) that will remain fairly constant even when using a larger range than *ST*.

For the high contention level, the shortest time for a terminal to wait before it can execute critical section is 0, and the longest time is *Token*

*traversal time +(n-1) * (critical section time)*, where *n* is the token ring size. The average case should be the average of these two values. Since the token traversal time for the 4 token algorithms ranges from 121 to 1781 milliseconds, and the critical section time ranges from 900 to 1150 milliseconds, the average case for a token ring with size 16 should be close to the range from 7560 milliseconds to 9415. And therefore, we consider the sleeping time range that gives the waiting time close to this range as the range for high contention level.

For the moderate level, we used the range between the low level and the high level.

9.3.2 Usage of the Sleeping Time in Terminals

1. Declare a variable j of type integer, and initialize it to 0.
2. At the beginning, each terminal i, where 0 <= i < n, fetches the value of SleepingTime [j][i]. After that, it increases the value of j by 1. If j is out of the index of the IdleTime array after the increase, set it to 0.
3. After the value is fetched, it will enter sleep. The sleeping time period is indicated by the number it just fetched. This is to simulate the terminal idle time and the different response time from different people. It avoids the problem that all the processes request access to critical section at the same time.

4. After the sleep, it will generate 20 valid numbers to simulate a response from a person, and then will request access to the critical section. When it gets privilege, it will execute critical section and releases the privilege after the critical section execution is finished.
5. Repeat step 2 to 4, until the terminal is terminated.

The good thing to generate idle time in advance this way rather than to generate them randomly in run time is that we can give a fair and precise comparison of different token ring algorithms. In this way, we can make sure that the test environment is identical for each token ring algorithm---the terminals will generate responses according to same procedure because the terminal idle time and the response time of the simulated terminal users are identical.

The simulated terminals were run on 4 computers with each computer simulating same number of terminals, and all the code for measuring performance was run on the 5th computer during the test.

9.4 Relationships between Different Servers

In chapter 7, code for algorithm implementation was introduced, in chapter 8, code for application implementation was introduced, and in this chapter, code for performance measurement was introduced. Now it is time to see the relationships between different code pieces.

We will begin by giving a simple review about the servers introduced.

1. Ring maker: used to form a logical token ring
2. Simulated terminal: used as terminals of an application system, it is an application combined with token ring algorithm.
3. Application server: in charge of updating the data, only the terminal with privilege can talk to it.
4. Stabilization time monitor: used to measure stabilization time for each token ring algorithm.
5. Token traversal time monitor: used to measure the time for a privilege traversal cycle, records how many critical sections are executed during the cycle, and calculates arbitration time for each process.

The relationships between different servers are shown in figure 5. Note that we run the simulated terminals on computers with ID 0, 1, 2 and 3, and all the code for servers and monitors are run on computer with ID 4.

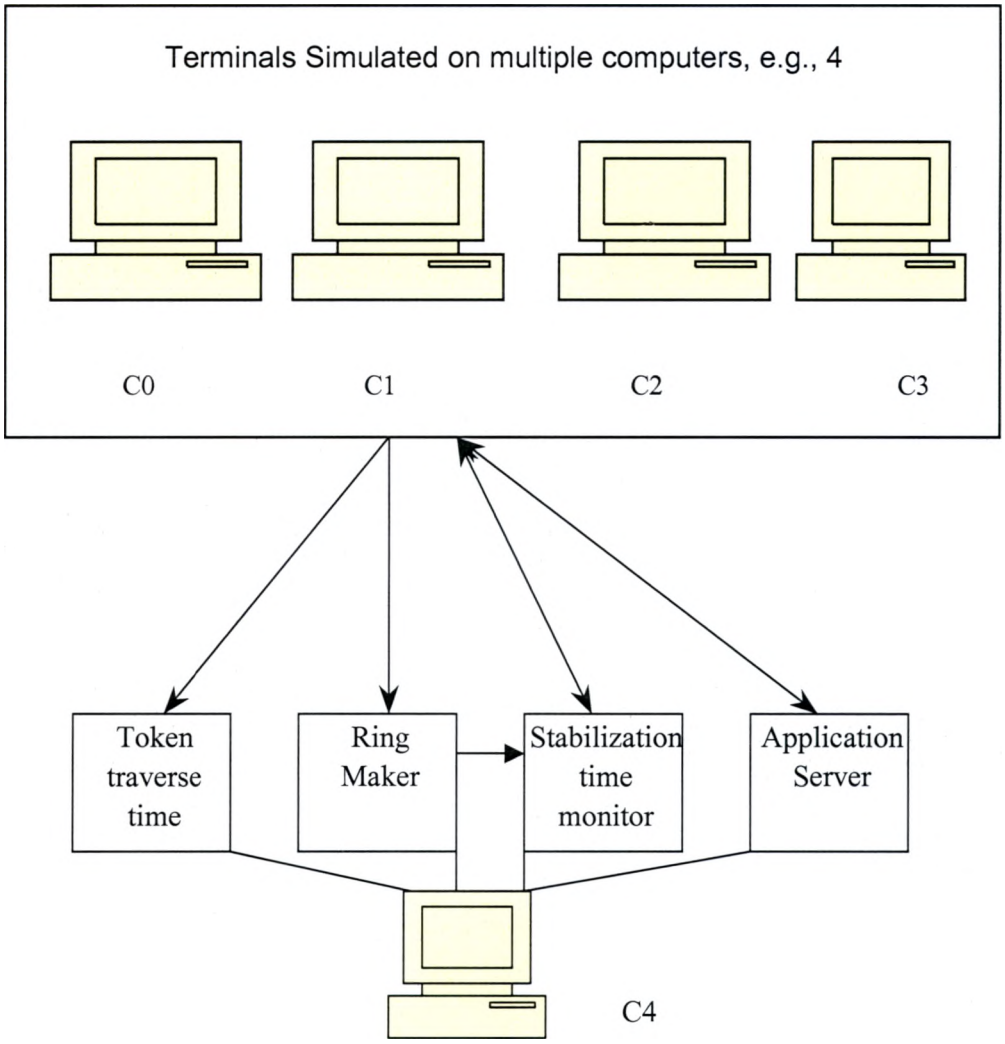

Figure 5: Relationships between servers

CHAPTER 10 PERFORMANCE COMPARISON

In Chapter 7, 8 and 9, the implementation for the 4 token ring algorithms combined with the test application, and the implementation of performance measurement were described. Now we need compare the performance of each token ring algorithm. We will start with an overview of performance comparison.

10.1 The Upper Bound Overview for Each Token Ring

As mentioned in chapter 1, the comparison of performance will be focused on the stabilization and token delivery, and therefore, only the upper bounds for these two metrics are listed here.

Token Ring	Stabilizes in	Delivers Token in
Enabler	O (n)	O (n^2)
Alternator	O (n)	O (n)
Shepherd	O (n^2)	O (n)
Companion	O (n)	O (n)

Table 2: Upper bound for 4 algorithms compared

The proof of these upper bounds is out of the scope of this research, and this table is listed here to indicate what we should expect from the sample run of the implementation.

In section 10.2 and 10.3, the performance comparisons of stabilization time and token delivery time are given respectively.

10.2 Stabilization Time Comparison

By observing the results from experimental runs, we can see that the upper bounds for stabilization time given in table 2 are verified in the implementation. Details about the results and analysis are given in 10.2.1 and 10.2.2.

10.2.1 Simulating 4 Terminals per Computer

The following table is the data of stabilization time in milliseconds from the experimental runs, since each run gives slightly different result, average value is calculated by running each set of code for 10 times.

Note that the data in table 3 were collected by letting each computer simulate 4 terminals, namely, when the ring size is 8, we used 2 computers, and when the ring size is 12, we used 3 computers, and when the ring size is 16, we used 4 computers. In this way, performance effects due to multiple simulated terminals executing on each actual computers are made consistent.

Token Ring	Ring size 8	Ring size 12	Ring size 16
Enabler	201	311	425
Alternator	291	461	610
Shepherd	137	267	561
Companion	123	201	270

Table 3: Stabilization time 4 processes per computer

The following charts drawn from the data in table 3 help us to see the trend for each algorithm more clearly.

1. The data for enabler token ring shows that it has a good performance in stabilization, and the chart drawn from the data is nearly a line, which verifies that its upper bound is O (n), where n is the size of the ring.

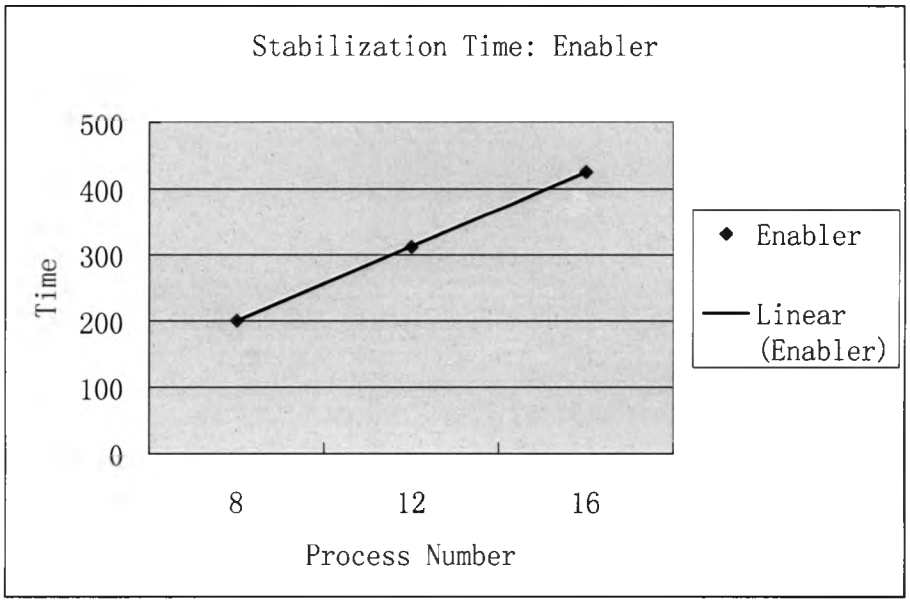

Figure 6: Stabilization time: Enabler token ring

2. The data for alternator token ring shows that even though it has an upper bound of O (n), it takes longer time to stabilize compared with other algorithms with same upper bounds, namely, the companion token ring and the enabler token ring. And for the ring sizes we compared, it even takes longer time compared with shepherd, which has an upper bound of O (n^2). But the chart drawn from the data is nearly a line, which verifies that its upper bound is O (n), where n is the size of the ring.

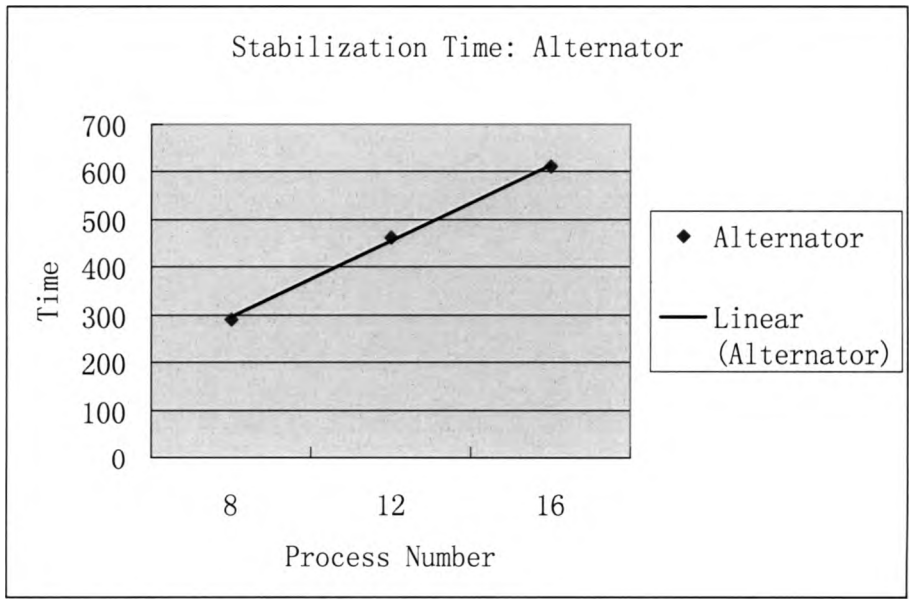

Figure 7: Stabilization time: Alternator token ring

3. The data for shepherd token ring shows that the time used for it to stabilize increases very quickly with the ring size increases, and the

chart drawn from the data is a curve of square, which verifies that its upper bound is O (n^2), where n is the size of the ring.

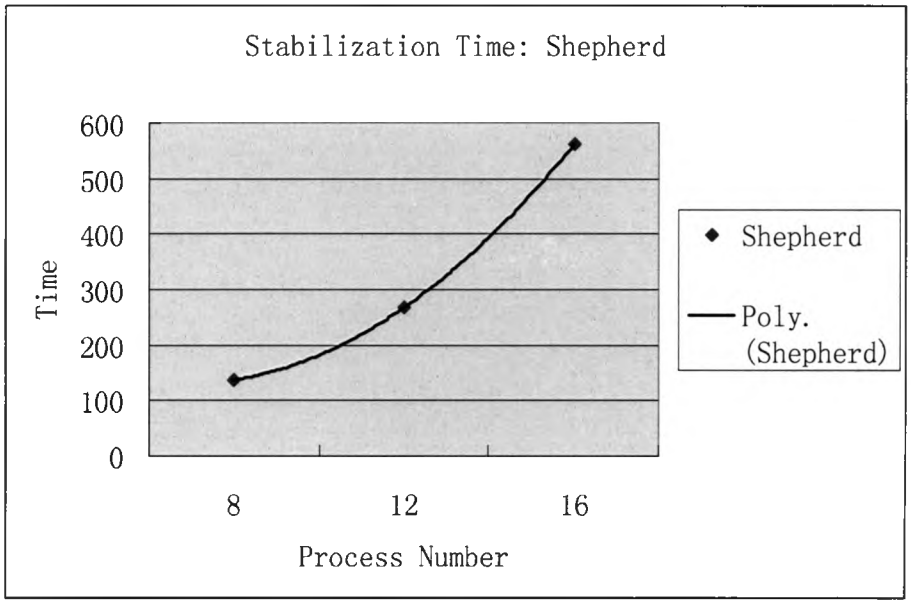

Figure 8: Stabilization time: Shepherd token ring

4. The data for companion token ring shows that it has a good performance in stabilization, and the chart drawn from the data is nearly a line, which verifies that its upper bound is O (n), where n is the size of the ring.

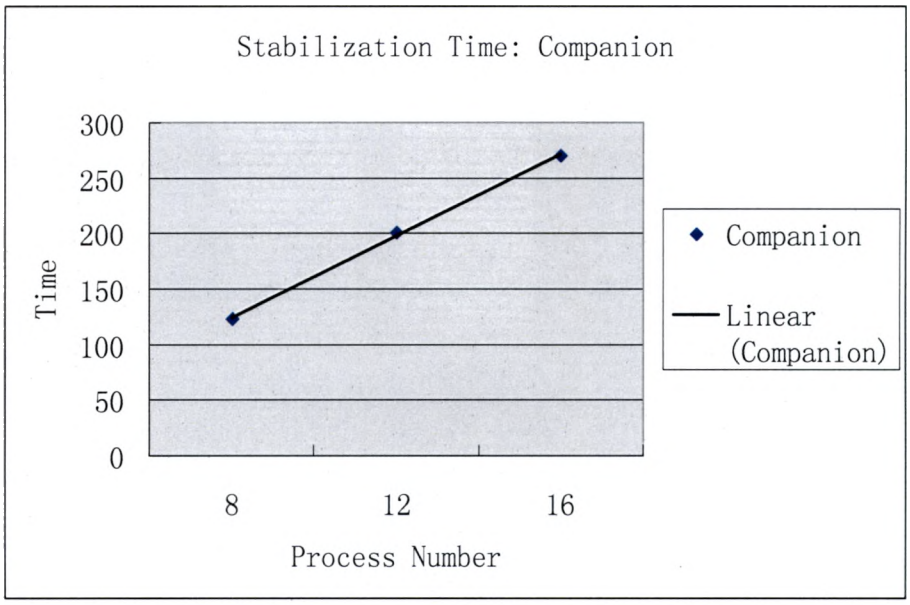

Figure 9: Stabilization time: Companion token ring

After viewing the charts of stabilization time for each individual token ring algorithm, now we generate a chart with the data from 4 algorithms on it.

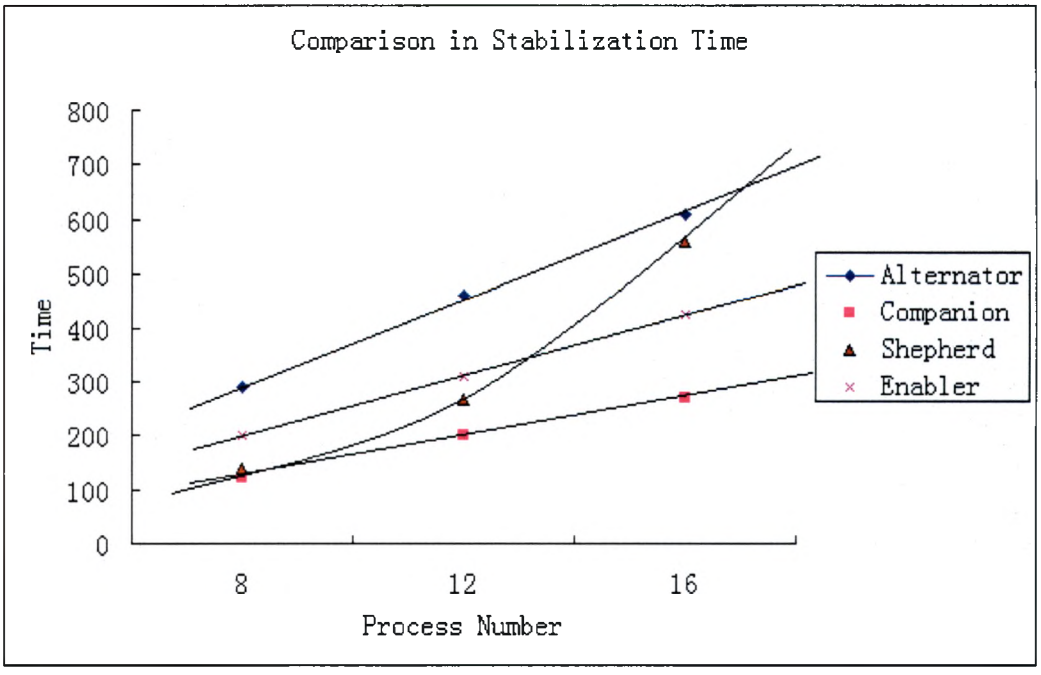

Figure 10: Stabilization time comparison for 4 token rings

From figure 10, we can see that each algorithm has the trend as their upper bounds indicate. We can see clearly that the stabilization time for shepherd increases very fast, even though it has a good performance when the token size is small, and the stabilization time for the other three algorithms increases linearly when the ring size increases.

As alternator token ring has a lower level protocol---the ring alternator, and it needs to communicates with two neighbors, it takes more time to stabilize, and therefore, though it has an upper bound of O (n), it uses more time than the other two algorithms with O (n) upper bound.

10.2.2 Simulating 8 Terminals per Computer

We also have some experimental runs with each computer simulating 8 terminals to see the performance difference. And table 4 listed the data for stabilization time with different token ring sizes.

Token Ring	Ring size 16	Ring size 32
Enabler	4061	7853
Alternator	8716	16284
Shepherd	5682	24067
Companion	2864	6113

Table 4 Stabilization time 8 processes per computer

From table 4, we can see that the trends for stabilization time for each

algorithm still hold, but with the time value increases more than doubling compared with each computer simulating 4 processes. This is because of the limitation of physical memory. And due to this limitation, we did not simulate more processes.

10.3 Token Delivery Time Comparison

By observing the results from experimental runs, we can see that the upper bounds for token delivery time given in table 2 are verified in the implementation. Details about the results and analysis are given in 10.3.1 and 10.3.2

10.3.1 Simulating 4 Terminals per Computer

The following table is the data of token delivery time (time for token traversing a cycle in the ring) in milliseconds from the experimental runs, since each run gives slightly different result, average value is calculated by running each set of code for 10 times.

Note that the data in table 5 were collected by letting each computer simulate 4 terminals, namely, when the ring size is 8, we used 2 computers, and when the ring size is 12, we used 3 computers, and when the ring size is 16, we used 4 computers. In this way, performance effects due to multiple simulated terminals executing on each actual computers are made consistent.

Token Ring	Ring size 8	Ring size 12	Ring size 16
Enabler	464	996	1781
Alternator	569	1011	1422
Shepherd	151	230	284
Companion	57	91	121

Table 5: Token delivery time 4 processes per computer

The following charts drawn from the data in above table help us to see the trend for each algorithm more clearly.

1. The data for enabler token ring shows that it has a bad performance in token delivery time, and the chart drawn from the data is a curve of square, which verifies that its upper bound is O (n^2), where n is the size of the ring.

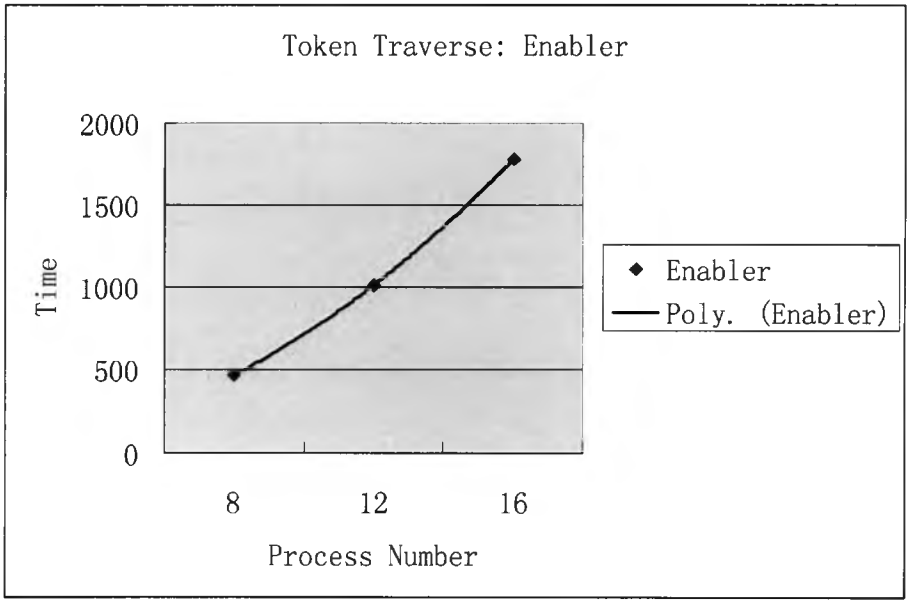

Figure 11: Token delivery time: Enabler token ring

2. The data for alternator token ring shows that the time used for token delivery increases linearly with the linear increase of the ring size, and the chart drawn from the data is also nearly a line, which verifies that its upper bound is O (n), where n is the size of the ring.

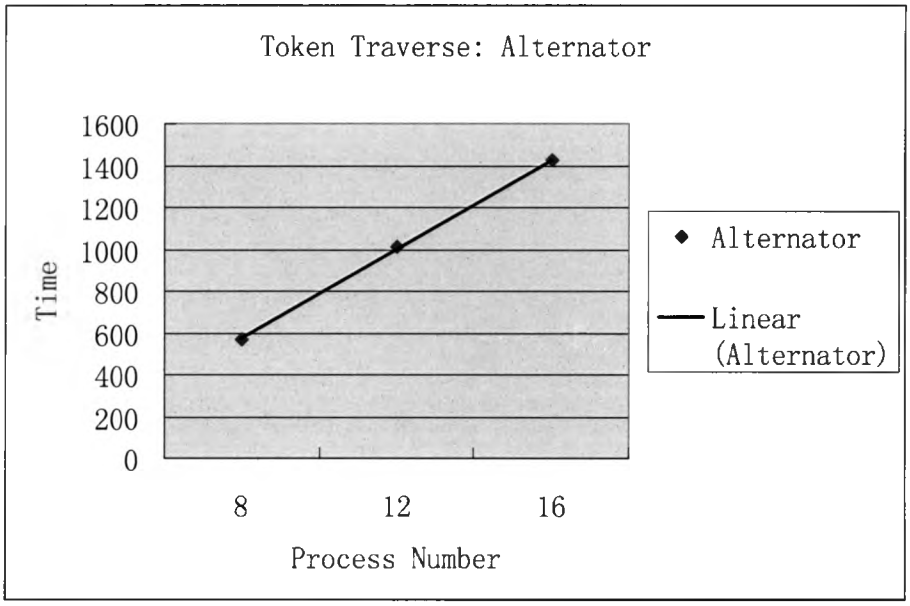

Figure 12: Token delivery time: Alternator token ring

3. The data for shepherd token ring shows that although it has the worst performance in stabilization, it performs pretty well in token delivery, and the chart drawn from the data is linear, which verifies that its upper bound is O (n), where n is the size of the ring.

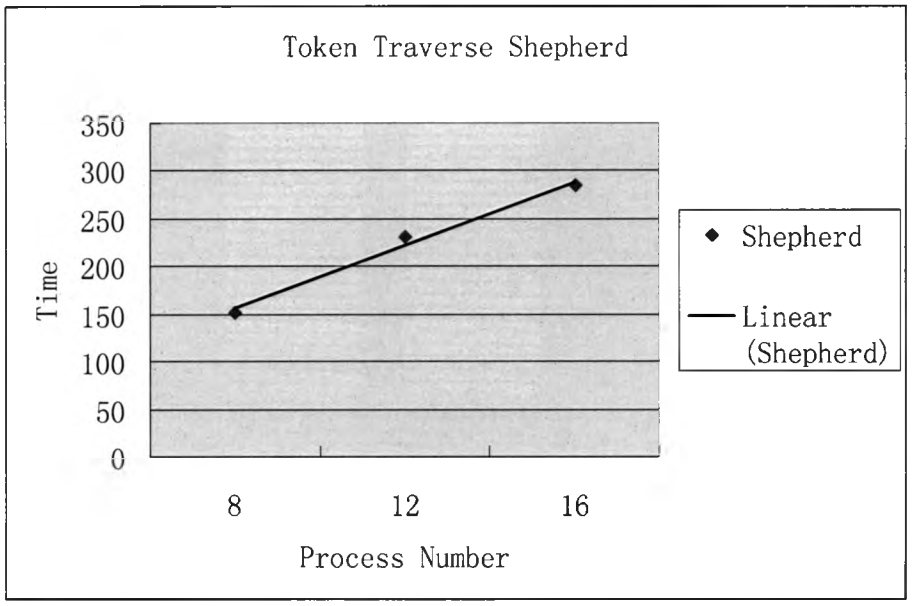

Figure 13: Token delivery time: Shepherd token ring

1. The data for companion token ring shows that it not only has a good performance in stabilization, but also performs well in token delivery. The chart drawn from the data is nearly a line, which verifies that its upper bound is O (n), where n is the size of the ring.

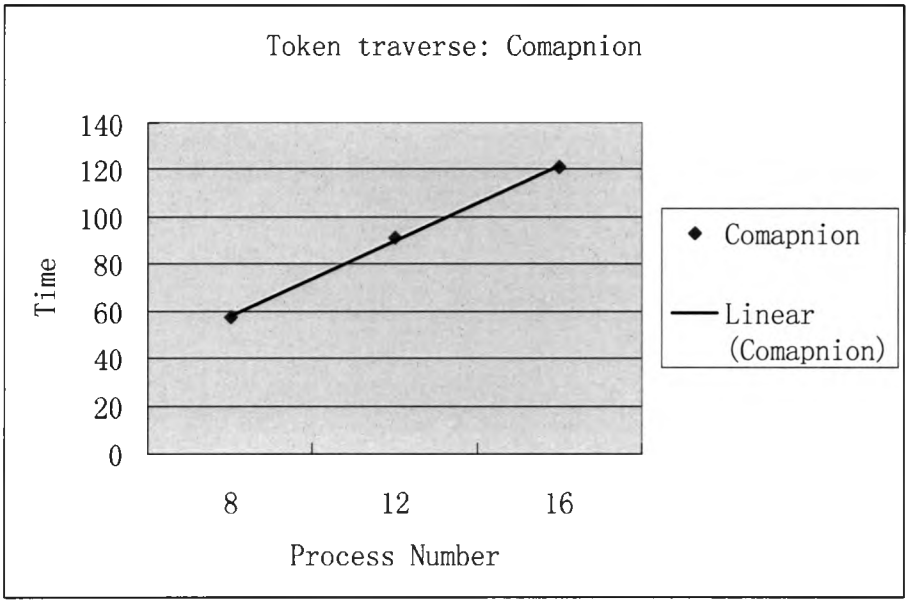

Figure 14: Token delivery time: Companion token ring

After viewing the chart of token delivery time for each individual token ring algorithm, now we generate a chart with the data from 4 algorithms on it.

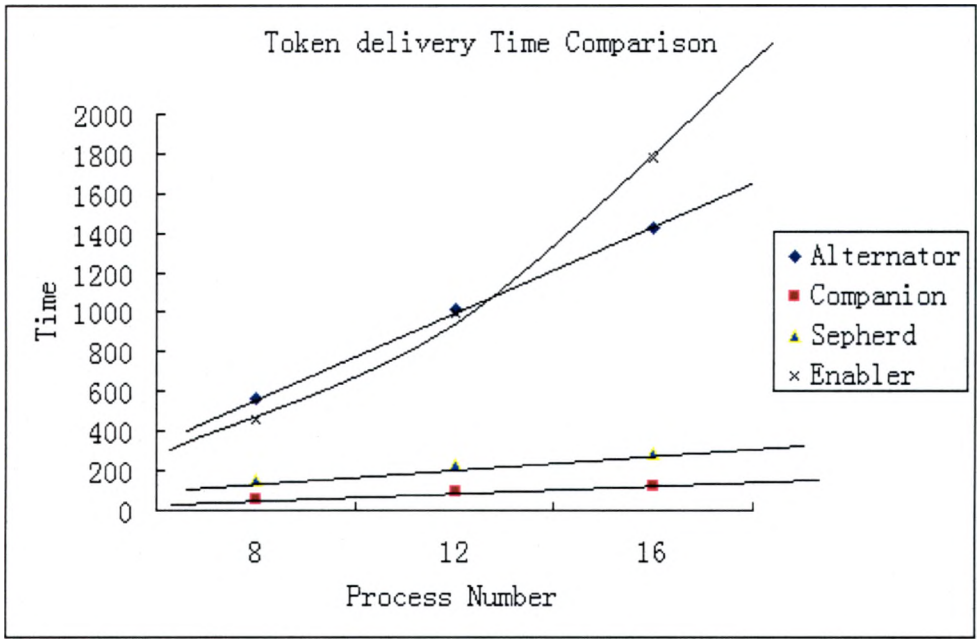

Figure 15: Token delivery time comparison for 4 token rings

From the chart, we can see clearly that each algorithm has the trend as their upper bounds indicate. We can tell that the performance in token delivery time for companion and shepherd token ring is close. The enabler performs worst because the delivery time increases very fast when the ring size increases. Again, although alternator token ring has the upper bound of O (n) in token delivery time, it takes longer than companion and shepherd token because of its lower level protocol and the more messages it has to deal with.

10.3.2 Simulating 8 Terminals per Computer

We also have some experimental runs with each computer simulating 8 terminals to see the performance difference. And table 5 listed the data for stabilization time with different token ring sizes.

Token Ring	Ring size 16	Ring size 32
Enabler	10268	57398
Alternator	9923	20482
Shepherd	2678	5073
Companion	1896	4136

Table 6 Token delivery time 8 processes per computer

From table 5, we can see that the trends for token delivery time for each algorithm still hold, but with the time value increases more than doubling compared with each computer simulating 4 processes. This is because of the limitation of physical memory. And due to this limitation, we could not simulate more processes.

10.4 Comparisons under Different Degrees of Contention

The tests in section 10.2 and 10.3 take no consideration of the requests from application, in another word, it only concerns with the pure stabilization time and token delivery time with no interference from outside. In this section, we consider the requests to access critical section from the simulated terminals and to see the performance of each algorithm.

In the tests, we make each algorithm combined with the simulated application system. The interest of measurement is the waiting time and total time needed to finish certain amount of critical executions. By saying waiting time, here we mean the time for a simulated terminal to get the privilege after it requests access to critical section.

We only test a token ring with size 16 simulated on 4 computers and combined with the application, and the number of critical section to be executed is set to 100. Table 5 shows the average waiting time with different sleeping ranges for each simulated terminal to get privilege after it requests.

10.4.1 Waiting Time Comparisons

We tried different numbers to find appropriate values to determine the sleeping range so that we can manipulate the contention levels. What listed in table 7 is the waiting time for different algorithms under different average sleeping times.

R \ T	70000	50000	35000	30000	25000	22500	17500	12500	10000	9000	8000
Enabler	2050	2519	2915	3260	3613	3865	5188	5387	6878	8684	8769
Alternator	1804	2134	2471	2605	2996	3524	4139	4961	6297	7052	7123
Shepherd	175	230	261	490	607	1284	1644	2334	2524	5764	6278
Companion	149	183	255	424	523	903	1462	2058	2281	5721	6027

Table 7: Waiting time under different average sleeping times

To view the data more obviously, we created charts for different algorithms as the following:

1. Waiting time for enabler token ring.

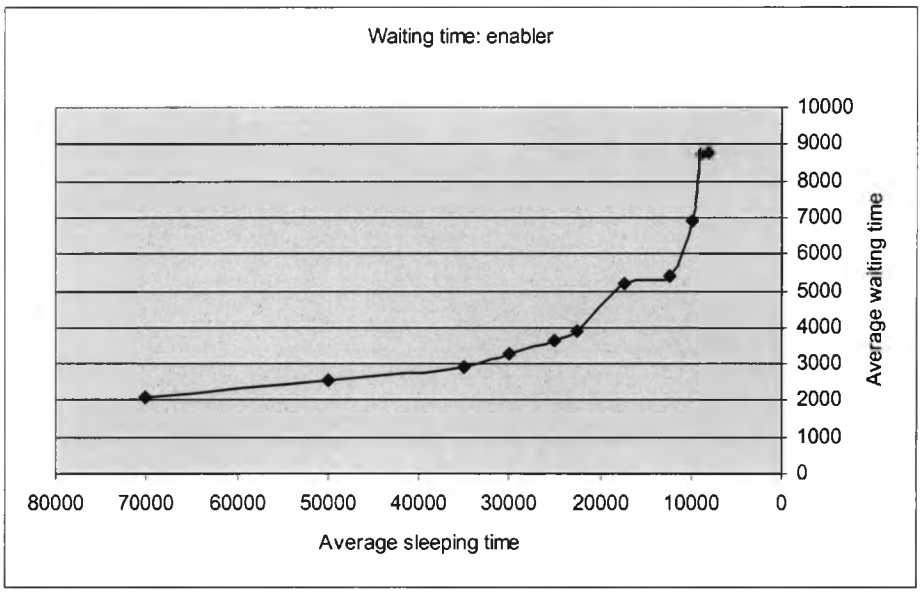

Figure 16: Waiting time: Enabler token ring

2. Waiting time for alternator token ring

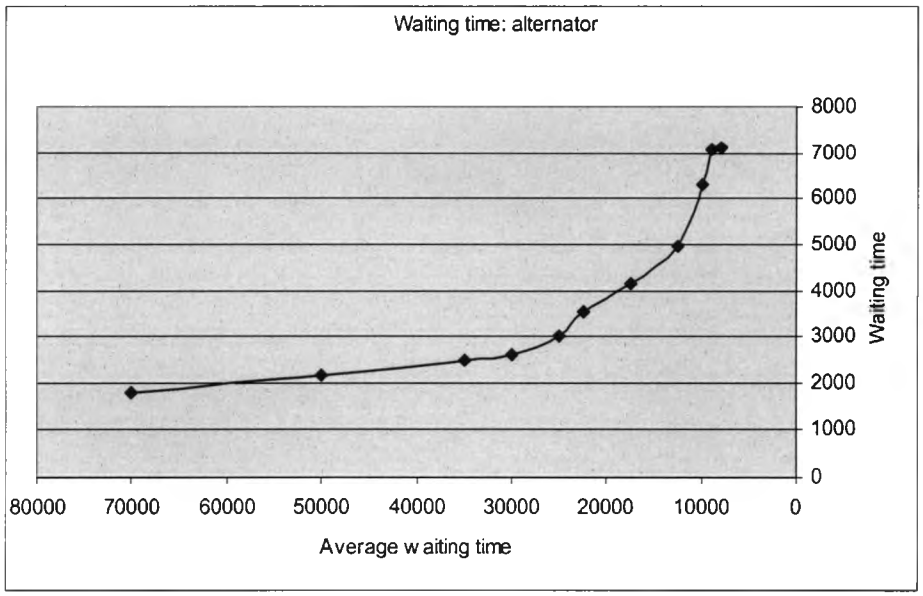

Figure 17: Waiting time: Alternator token ring

3. Waiting time for shepherd token ring

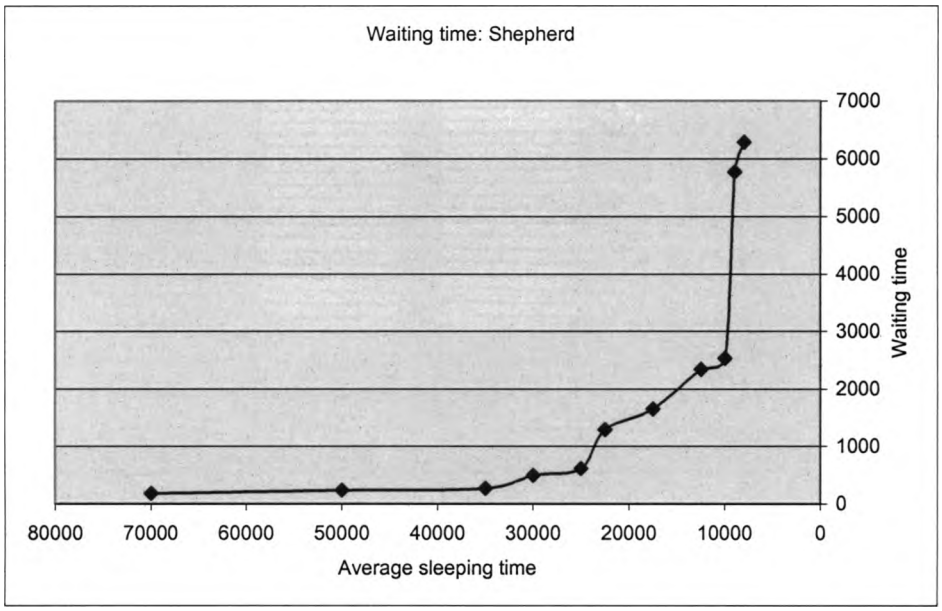

Figure 18: Waiting time: Shepherd token ring

4. Waiting time for companion token ring

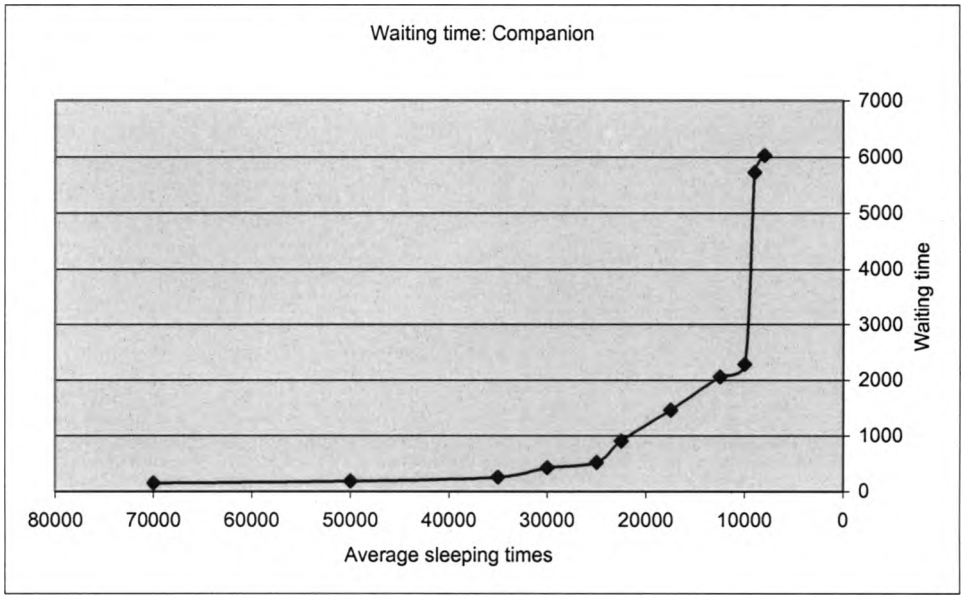

Figure 19: Waiting time: Companion token ring

The graphs in above 4 charts have similar shapes, and this verifies the delineation we described in section 9.3.1. At the low level contention, there is

no much difference in average waiting time, but with the contention increases, the average waiting time increases sharply.

From the data given above, it is clear to see that under low-level contention, the waiting time for each algorithm is relatively short. And the waiting time under high-level contention is impressively long.

10.4.2 Number of Finished Critical Section Comparisons

Similar to the way we used to measure the waiting time, we measured the number of critical sections executed for each algorithm under different average sleeping times, given a time period of 2000,000 milliseconds.

Table 8 listed the detailed data.

R \ T	70000	50000	35000	30000	25000	22500	17500	12500	10000	9000	8000
Enabler	416	588	786	905	1087	1160	1338	1401	1472	1626	1643
Alternator	419	598	816	921	1122	1182	1396	1427	1506	1723	1842
Shepherd	423	607	859	980	1219	1421	1537	1722	1857	1946	1966
Companion	424	610	864	1100	1287	1425	1561	1819	1908	1951	1979

Table 8: Number of critical section executed under different average sleeping times

To view the data more obviously, we created charts for different algorithms as the following:

1. Number of critical sections executed by enabler token ring in 2000,000 milliseconds under different average sleeping times:

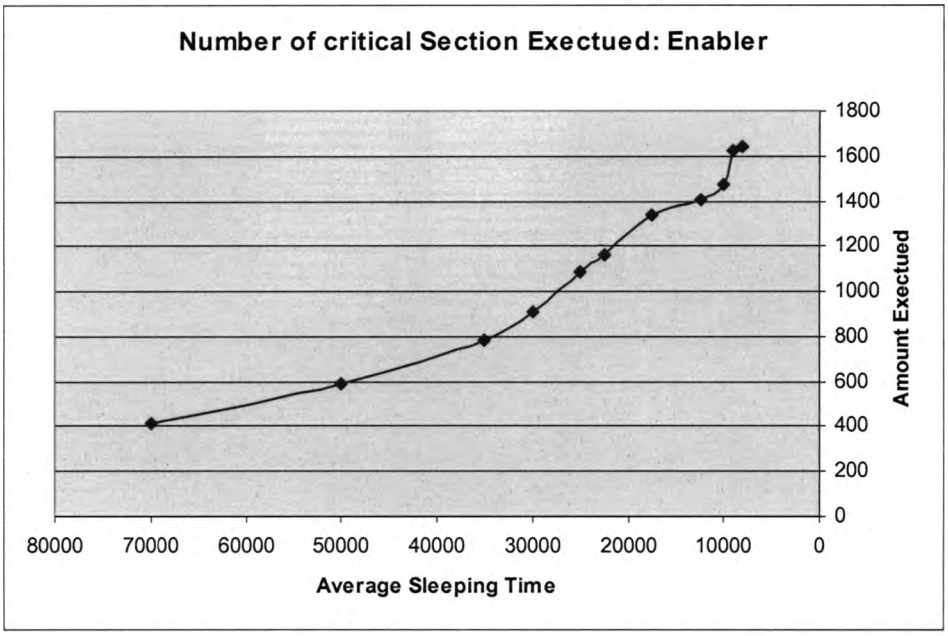

Figure 20: Number of critical section executed: Enabler token ring

2. Number of critical sections executed by alternator token ring in 2000,000 milliseconds under different average sleeping times:

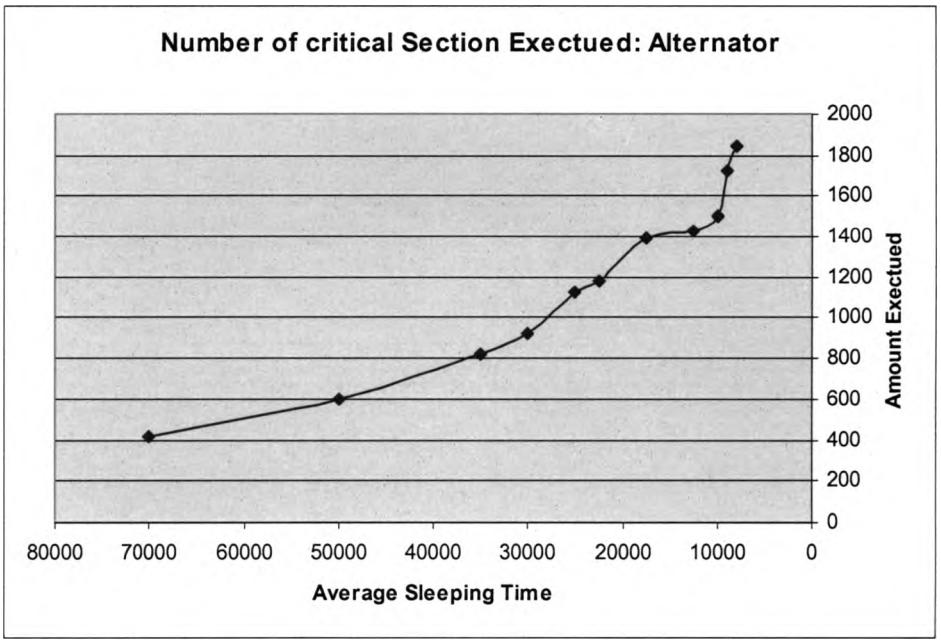

Figure 21: Number of critical section executed: Alternator token ring

3. Number of critical sections executed by shepherd token ring in 2000,000 milliseconds under different average sleeping times:

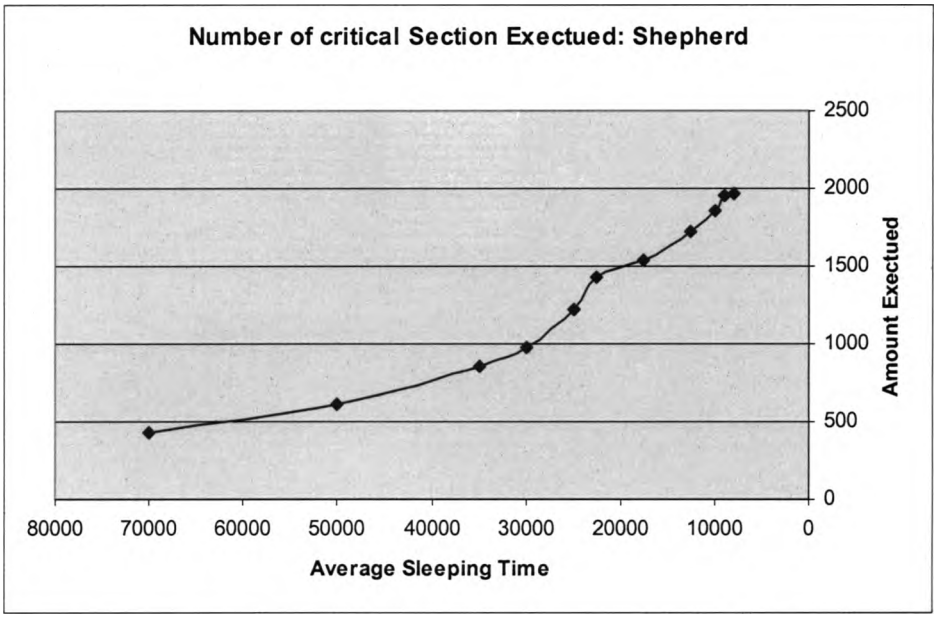

Figure 22: Number of critical section executed: Shepherd token ring

4. Number of critical sections executed by companion token ring in 2000,000 milliseconds under different average sleeping times:

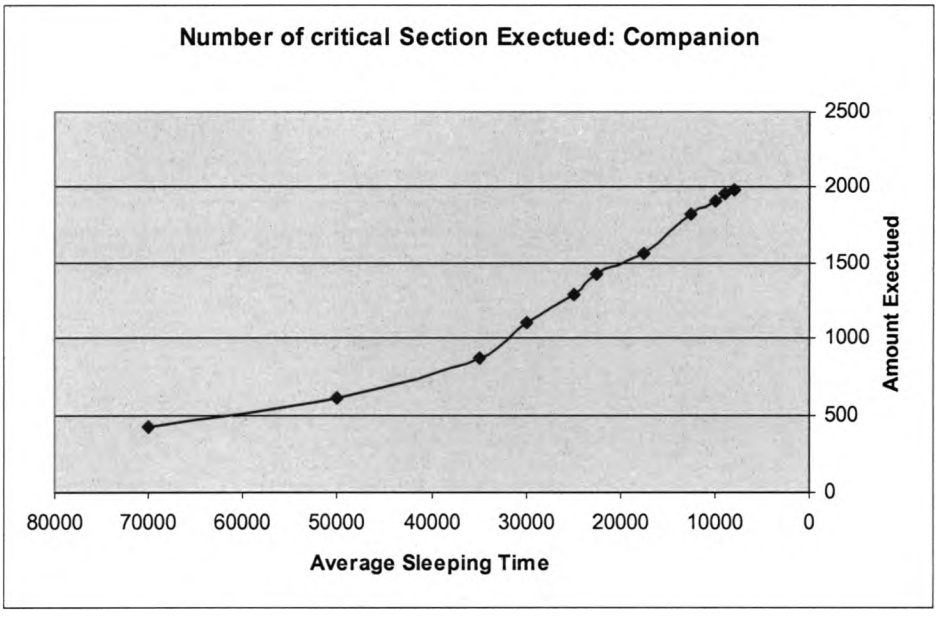

Figure 23: Number of critical section executed: Companion token ring

Again, the graphs in the charts have similar shapes. We can see clearly that although the waiting time under low contention is short, the number of critical section executed is also small, and under high contention, the number of critical section executed is much greater. This is because that given same amount of time, under low contention, less processes request for the access to the critical section, while under high contention, more processes request for execute critical section.

We can also see that under low contention, there is no big difference in the number of critical section executed between different algorithms, while under high contention, the difference is obvious.

From what discussed in section 10.4.1 and 10.4.2, we can see that the performance of the token delivery time affects the waiting time and the number of critical section executed per time period no matter the application system is under what contention level. Put the 4 algorithms under same conditions, it is clearly seen that the faster the token delivered, the less the waiting time, and the more critical section can be executed during a certain amount of time.

CHAPTER 11 CONCLUSIONS

As we can see from the performance comparison in chapter 10, we can safely draw the conclusion that companion token ring has a good performance in general because it takes an upper bound of O (n) both in stabilization time and token delivery time. Though alternator token ring has the same upper bounds with companion token ring, it takes more time compared with the latter because it must wait for the alternator to stabilize and it must communicate with two neighbors. As for shepherd token ring, it has good performance in token delivery aspect, but its stabilization aspect is much worse than the other three algorithms, which means shepherd token ring can cause worse damage in the case of transient faults occurring compared with other algorithms under same conditions. And for the enabler token ring, it can stabilize very fast, but once stabilized, the system with enabler token ring would perform less critical section executions compared with other three algorithms under same conditions due to its slow token traversal time.

What described above is in general. If a choice needs to be made among these different algorithms, more factors, such as the size of the system, the requirement for fault tolerance degree, and so on, need to be considered.

In this research, we focused on the performance evaluation and comparison among these 4 token ring algorithms, and there is some future work left to do. (1) It would be more general if other algorithms (e.g.[FD94]) were also considered. (2) Under low contention level, a central monitor for scheduling permission to execute critical section might be a better choice. (3) Further explanation of relationships between token time, critical section time and non-critical section should be considered.

REFERENCES

1) [GH96] Mohamed G. Gouda, F. Furman Haddix: The Stabilizing Token Ring in Three Bits, Journal of Parallel and Distributed Computing 35(1), 43-48, 1996

2) [GH97] Mohamed G. Gouda, F. Furman Haddix: The Linear Alternator, International Informatics Series 7: Self-Stabilizing Systems, 31-47, 1997

3) [Dijk74] Dijkstra, Edsger W: Self-stabilizing systems in spite of distributed control. Commun. ACM (Nov. 1974)

4) [Hadd91] F. Furman Haddix: Stabilization of Bounded Token Rings. Master's thesis, University of Texas at Austin, Dec 1991. (Also Tech. Rep. ARL-TR-91-31, Applied Research Laboratories, University of Texas at Austin, Dec. 1991).

5) [PV2000] Frank Petit, Vincent Villain: Self-stabilizing depth-first token circulation in asynchronous message-passing systems

6) [Herman98] Ted Herman, Self-stabilization Bibliography: Access guide

7) [Dijk82a] Dijkstra, Edsger W: The solution to a cyclic relaxation problem, Selected writings on computing: a personal perspective, 34-35, 1982

8) [Dijk82b] Dijkstra, Edsger W: Self-stabilization in spite of distributed control, Selected writings on computing: a personal perspective, 41-46, 1982

9) [LL90] L Lamport, L Lynch: Distributed computing: models and methods, Handbook of theoretical computer science, Chapter 18, 1157-1199

10) [BGM93] JE Burns, Mohamed G. Gouda, RE Miller: Stabilization and pseudo-stabilization, distributed computing, 7, 35-42, 1993

11) [BYC88] F Bastani, I Yen, I Chen: A class of inherently fault tolerant distributed programs, IEEE Transactions on software engineering, 14, 1432-1442, 1988

12)[BYZ89] F Bastani, I Yen, Y Zhao: On self-stabilization, non-determinism and inherent fault tolerance, Proceedings of the MCC workshop on self-stabilizing systems, MCC technical report No. STP-379-89, 1989

13)[BY93] F Bastani, IL Yen: Inherent fault tolerance in decentralized process control systems, International symposium on autonomous decentralized systems, Kawasaki Japan, 267-274, 1993

14)[YBL91] IL Yen, F Bastani, EL Leiss: An inherently fault toleranr sorting algorithm, Proceedings of the fifth international parallel processing symposium, 34-42, 1991

15) [FD94] M Flatebo, AK Datta: Two-state self-stabilizing algorithms for token rings, IEEE Transactions on Software Engineering, 20:500-504, 1994

APPENDICES

APPENDIX 1 ALTERNATOR TOKEN

```
mport java.io.*;
mport java.net.*;
mport java.lang.*;

**
 This class implements alternator token ring algorithm
/

lass AlternatorToken extends Thread

   private InetAddress ringMakerAddress;
   private InetAddress timeMonitorAddress;
   private InetAddress measureAddress;

   public int serverID;
   public int leftNeighborPort;
   public int rightNeighborPort;
   private int action;


   private InetAddress leftNeighborAddress;
   private InetAddress rightNeighborAddress;

   private DatagramSocket sendSocket;
   private DatagramSocket actionSocket;
   private DatagramSocket talkToRMSocket;   //used to talk to Ring Maker
   private DatagramSocket talkToTimeSocket; //used to talk to timeMonitor
   private DatagramSocket listenSocket;  //socket used to listen from its precessor
   private DatagramSocket talkTokenTraverseSocket; //used to talk to timeMonitor

   public  boolean grant;
   private PollServer pollServerInCharge; //proecess belongs to which application

   private boolean b;    //for general process(1 to Max processID - 1) itself
   private boolean b0;  //for process b0
   private boolean bn;  //for process bn
   private boolean leftb;   //first neighbor
   private boolean rightb;   //second neighbor
   private boolean tkn;
   private boolean pTkn;
   private boolean rdy;  //local
   private boolean T;    //used as time guard
   private long start;  //used to calculate time out
   private final int TimeOut = 2;
   public  boolean ignore = false;  //used to control if to send msg when executing cs
   public  boolean reportCS = false;  //used to control if to send msg when executing cs

   /**
   * constructor
   */
   public AlternatorToken( InetAddress addr, int listenPort, int talkToRMport, PollServer ps↙
       , int sid )
   {
       ringMakerAddress = addr;
       timeMonitorAddress = addr;
       measureAddress = addr;
       pollServerInCharge = ps;
       serverID = sid;

       start = System.currentTimeMillis();

       leftNeighborPort = -1;
       rightNeighborPort = -2;
       leftNeighborAddress = null;
       rightNeighborAddress = null;
       grant = false;
       action = 0;
```

```
    try
    {
        sendSocket = new DatagramSocket();
        actionSocket = new DatagramSocket();
        talkToTimeSocket = new DatagramSocket();
        talkTokenTraverseSocket = new DatagramSocket();
        listenSocket = new DatagramSocket( listenPort, InetAddress.getByName("147.26.101.141"));
        talkToRMSocket = new DatagramSocket(talkToRMport, InetAddress.getByName("147.26.101.141"));
    } catch( SocketException e ) {;}
      catch ( UnknownHostException e ) { System.out.println("UnknownHost: " + e.getMessage());}
    try {
                listenSocket.setSoTimeout(1);
    } catch(IOException ex) {;}
}

private int boolToInt( boolean a )
{
    if( a )
        return 1;
    else
        return 0;
}

//return a string with at most 2 elements, 1st is the token, 2nd, is b0(or b), third is bn
private String processTokenAndAlternatorForTokenTraversal()
{
    String tmp="333";

    if ( serverID == 0 )
    {
        if( b0==rightb && bn!=leftb ) //if G1
        {
            //**************case 1 no request  if G1^G5**********************************
            if( tkn==pTkn &&(tkn || rdy ) && pollServerInCharge.request == false)
            {
                if( ignore == false )
                {
                    byte[]sendData = new byte[10];
                    sendData = (Integer.toString( serverID )+" get").getBytes();
                    DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.MeasurePort);
                    try{talkTokenTraverseSocket.send(outPacket);  }//tell TokenTraversalMonitor its privilege
                    catch(IOException e ){}
                }
                else ignore = false;
                reportCS = false;

                tkn = !tkn;  //A5
                rdy = false;   //A5
                bn = !bn;  //A1
                b0 = !b0;   //A1
                T = false;  //A1

                byte[] sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" release").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket2);  }//tell TokenTraversalMonitor its privilege
                    catch(IOException e ){}
                grant = false;

                tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn)+"";
```

```
            return tmp;

        }//end of if12


         //**************case 1 with request  G1^G5*********************************
        else if( tkn==pTkn &&(tkn || rdy ) && pollServerInCharge.request == true )
        {

            if( ignore == false )  //should send msg
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.
                                      MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket);  }//tell
                    TokenTraversalMonitor its privilege
                catch(IOException e ){}
                ignore = true;   //set it to true, so not to send multiple times
            }

            grant = true;

            if( reportCS == false )
            {
                byte[]sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" cs").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.
                                      MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket2);  }//tell
                    TokenTraversalMonitor its privilege
                      catch(IOException e ){}
                reportCS = true; //reported already
            }
            return tmp;
        }//end of if12

        //**********************case 2 G1^G6************************
        else if( tkn==pTkn && !rdy && !tkn )
        {
            rdy = true; //A6
            b0 = !b0;  //A1
            T =false;  //A1
            bn = !bn;  //A1

            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn)+"";
            return tmp;
        }
        //**********************case 3 G1^!g5^!G6*****************************
        else //if( !G5( tkn, pTkn, rdy ) && !G6( tkn, pTkn, rdy ) )
        {
            b0 = !b0;  //A1
            bn = !bn;  //A1
            T = false;  //A1
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn)+"";
            return tmp;
        }
    }//end of if11

    else if( (System.currentTimeMillis()-start)>TimeOut && T )
    {
        //*****************case 4 if G2**************************
        if( b0==rightb && bn == leftb )
        {
            b0 = !b0;   //A1
            T = false;  //A1
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn)+"";
```

```
            return tmp;
        }
        //*****************case 5 if G3***************************
        else if( b0 !=rightb && bn != leftb )
        {
            bn = !bn;   //A2
            T = false;  //A2
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn)+"";
            return tmp;
        }
    }

    //*****************case 6 if G4***************************
    else if ( ((b0==rightb && bn==leftb) || (b0!=rightb && bn!=leftb))   && !T )
    {

        T = true;
        start = System.currentTimeMillis();
        return tmp;
    }
    return tmp;
}//end of if serverID == 0

else  //other processes
{

    if( (rightb == b) && (b != leftb) ) //if Bi
    {
        if( tkn != pTkn && (rdy || tkn ) && pollServerInCharge.request == false) //no↙
            request
        {
            if( ignore == false )
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.   ↙
                                      MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket);  }//tell                ↙
                    TokenTraversalMonitor its privilege
                catch(IOException e ){}
            }
            else ignore = false;   //set ignore to false because for case 2
            reportCS = false;  //set for case 2

            tkn = !tkn;
            b = !b;
            rdy = false;

            tmp = boolToInt(tkn)+""+boolToInt(b)+" ";


            byte[] sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" release").getBytes();

            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.  ↙
                                  MeasurePort);
            try{talkTokenTraverseSocket.send(outPacket2);  }//tell                  ↙
                TokenTraversalMonitor its privilege released
                      catch(IOException e ){}
            grant = false;
            return tmp;
        }
        //with request==true
        else if( tkn != pTkn && (rdy || tkn ) && pollServerInCharge.request == true)
        {
            if( ignore == false )
            {
                byte[]sendData = new byte[10];
```

```
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.
                                      MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket);  }//tell
                    TokenTraversalMonitor its privilege
                      catch(IOException e ){}
                ignore = true;   //set it to true, so not to send multiple times
            }

            grant =  true ;   //  setGrant( true);

            if ( reportCS == false )
            {
                byte[]sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" cs").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.
                                      MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket2);  }//tell
                    TokenTraversalMonitor cs
                      catch(IOException e ){}
                reportCS = true;
            }
            return tmp;
        }

        else if( (tkn != pTkn) && !tkn && !rdy  )
        {
            b = !b;
            rdy = true;
            tmp = boolToInt(tkn)+""+boolToInt(b)+" ";
            return tmp;
        }

        else
        {
            b = !b;
            tmp = boolToInt(tkn)+""+boolToInt(b)+" ";
            return tmp;
        }
    }//end of if11
    return tmp;
  }
}


//return a string with at most 2 elements, 1st is the token, 2nd is b0(or b), third is bn
//"1" means high, "0" means low, and 3 means no change
private String processTokenAndAlternatorNoRequest()
{
    String tmp="333";

    if ( serverID == 0 )
    {

        if( b0==rightb && bn!=leftb ) //if G1
        {
            //**************case 1 no request  if G1^G5*********************************
            if( tkn==pTkn &&(tkn || rdy ) )  //privileged
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )).getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, timeMonitorAddress, FinalVariables
                                      .PrevilegePort);
                try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
```

```
                    catch(IOException e ){}
            tkn = !tkn;  //A5
            rdy = false;   //A5
            bn = !bn;  //A1
            b0 = !b0;   //A1
            T = false;

            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
        }//end of if12

        //**********************case 2************************
        else if( tkn==pTkn && !rdy && !tkn )
        {
            rdy = true; //A6
            b0 = !b0;  //A1
            bn = !bn;  //A3
            T = false;
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
        }

        //************************case 3 G1^!g5^!G6*****************************
        else //if( !G5( tkn, pTkn, rdy ) && !G6( tkn, pTkn, rdy ) )
        {
            b0 = !b0;  //A1
            bn = !bn;  //A1
            T = false;  //A1
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
        }
    }//end of if G1

    else if( (System.currentTimeMillis()-start)>TimeOut && T )
    {
        //*****************case 4 if G2**************************
        if( b0==rightb && bn == leftb )
        {
            b0 = !b0;   //A1
            T = false;  //A1
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
        }
        //*****************case 5 if G3**************************
        else if( b0 !=rightb && bn != leftb )
        {
            bn = !bn;   //A2
            T = false;  //A2
            tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
        }
    }

    //*****************case 6 if G4**************************
    else if ( ((b0==rightb && bn==leftb) || (b0!=rightb && bn!=leftb))    && !T )
    {
        T = true;
        start = System.currentTimeMillis();
        tmp = boolToInt(tkn)+""+boolToInt(b0)+""+boolToInt(bn);
            return tmp;
    }
    return tmp;
}//end of if serverID == 0

else  //other processes
{

    if( (rightb == b) && (b != leftb) ) //if Bi
    {   if( tkn != pTkn && (rdy || tkn ) ) //have privilege
        {
            byte[]sendData = new byte[10];
```

```
                sendData = (Integer.toString( serverID )).getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                sendData.length, timeMonitorAddress, FinalVariables.PrevilegePort);
                try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its previlege
                        catch(IOException e ){}
                tkn = !tkn;
                b = !b;
                rdy = false;

                tmp = boolToInt(tkn)+""+boolToInt(b)+" ";
                return tmp;
            }
            else if( (tkn != pTkn) && !tkn && !rdy  )
            {
                b = !b;
                rdy = true;

                tmp = boolToInt(tkn)+""+boolToInt(b)+" ";
                return tmp;
            }
            else
            {
                b = !b;
                tmp = boolToInt(tkn)+""+boolToInt(b)+" ";
                return tmp;
            }
        }//end of if11
    }
    return tmp;
}



public void run()
{
    ProcessInfo [] p = new ProcessInfo [2];
    for( int q = 0; q < 2; q++)
        p[q] = new ProcessInfo();

    RingInitialization init = new RingInitialization( ringMakerAddress, listenSocket,
        talkToRMSocket, serverID );

    init.findNeighbor( p );

    if( serverID == 0 )
    {
        for( int s=0; s < 2; s++)
        {
            if( p[s].serverID == 1 )
            {
                rightNeighborAddress = p[s].address;
                rightNeighborPort = p[s].lport;
            }
            else
            {
                leftNeighborAddress = p[s].address;
                leftNeighborPort = p[s].lport;
            }
        }
    }
    else   //other process
    {   for( int s=0; s < 2; s++)
        {
            if( serverID - p[s].serverID == 1 )  //it's left neighbor
            {
                leftNeighborAddress = p[s].address;
                leftNeighborPort = p[s].lport;
            }
            else
```

```
        {
            rightNeighborAddress = p[s].address;
            rightNeighborPort = p[s].lport;
        }
    }
}

int flag = 0;  //used only when test stabilization time
boolean firstTime = true;

for( ;; )  //this for loop testing stabilization time
{
    try
    {
        byte[] sendData = new byte[FinalVariables.PKSIZE];

        //stabilization time measurement
        String tmp1 = processTokenAndAlternatorNoRequest();


        if( !tmp1.equals("333") || firstTime )  //if there is change, send it
        {
            sendData = ( tmp1 + serverID).getBytes();
            firstTime = false;
            DatagramPacket leftOutPacket = new DatagramPacket( sendData, sendData.
                length, leftNeighborAddress, leftNeighborPort );
            sendSocket.send( leftOutPacket );
            DatagramPacket rightOutPacket = new DatagramPacket( sendData, sendData.
                length, rightNeighborAddress, rightNeighborPort );
            sendSocket.send( rightOutPacket );

        }

        byte[] recvData = new byte[FinalVariables.PKSIZE];
        DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

        try{
            listenSocket.receive( inPacket );
        }catch(SocketTimeoutException ex ){continue;}

        String tmp = new String(inPacket.getData()).trim();

        if( tmp.equals("stop measure stabilization") )
        {
            byte[]actionData = new byte[1000];
            actionData = (Integer.toString( action )).getBytes();
            DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                length, timeMonitorAddress, FinalVariables.ActionPort);

            try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
            catch(IOException e ){}
            break;
        }

        if( flag == 1 )  //waiting for restart
        {
            if( tmp.equals("restart") )  //initialize
            {
                flag = 0;
                tkn = false;
                b = false;
                b0 = false;
                bn = false;
                leftb = false;
                rightb = false;
                pTkn = false;   //precessor tkn
            }
        }
        else     //must deal with the msg
        {
            if( tmp.equals( "stop" ) )
```

```
    flag = 1;
else
{
    if( serverID == 0 ) //special
    {
        //from n-1, left neighbor, only use tkn and bn, at index 0 and 2
        if( Integer.parseInt( tmp.substring(3))==FinalVariables.
            ServerNumber-1 )
        {
            if( tmp.charAt(0) == '1' )
                    pTkn = true;
            else if ( tmp.charAt(0) == '0' )
                    pTkn = false;
            if( tmp.charAt(1) == '1' )
                    leftb = true;
            else if ( tmp.charAt(1) == '0' )
                    leftb = false;
            continue;
        }
        else if( Integer.parseInt( tmp.substring(3)) == 1 )//from 1,
            should use only  b0, at index 1
        {
            if( tmp.charAt(1) == '1' )
                rightb = true;
            else if ( tmp.charAt(1) == '0' )
                    rightb = false;
            continue;
        }
    }
    else if( serverID == 1 ) //special
    {
            //from 0, left neighbor, only use tkn and b0, at index 0 and
                1
        if( Integer.parseInt( tmp.substring(3))==0 )
        {
            if( tmp.charAt(0) == '1' )
                pTkn = true;
            else if ( tmp.charAt(0) == '0' )
                pTkn = false;
            if( tmp.charAt(1) == '1' )
                leftb = true;
            else if ( tmp.charAt(1) == '0' )
                leftb = false;
            continue;
        }
        else if( Integer.parseInt( tmp.substring(3)) == 2 )//from 2,
            should use only  b, at index 1
        {
            if( tmp.charAt(1) == '1' )
                rightb = true;
            else if ( tmp.charAt(1) == '0' )
                rightb = false;
            continue;
        }
    }
    else if( serverID == FinalVariables.ServerNumber-1 ) //special
    {
        //from 0, right neighbor, only use bn, at index 2
        if( Integer.parseInt( tmp.substring(3))==0 )
        {
            if( tmp.charAt(2) == '1' )
                rightb = true;
            else if ( tmp.charAt(2) == '0' )
                rightb = false;
            continue;
        }
        else if( Integer.parseInt( tmp.substring(3)) == FinalVariables.
            ServerNumber-2 )//from left neighbor,should use token and b,
            at index0, 1
        {
            if( tmp.charAt(0) == '1' )
```

```
                        pTkn = true;
                    else if ( tmp.charAt(0) == '0' )
                        pTkn = false;
                    if( tmp.charAt(1) == '1' )
                        leftb = true;
                    else if ( tmp.charAt(1) == '0' )
                        leftb = false;
                    continue;
                }
            }
            else  //other processes
            {
                //extract the values of preseccor's token and b
                if( Integer.parseInt( tmp.substring(3)) == (serverID -1) )
                {
                    if( tmp.charAt(0) == '1' )
                        pTkn = true;
                    else if ( tmp.charAt(0) == '0' )
                        pTkn = false;
                    if( tmp.charAt(1) == '1' )
                        leftb = true;
                    else if ( tmp.charAt(1) == '0' )
                        leftb = false;
                    continue;
                }
                else //extract the value of preseccor's b
                {
                    if( tmp.charAt(1) == '1' )
                        rightb = true;
                    else if( tmp.charAt(1) == '0' )
                        rightb = false;
                }
            }
        } //end of else
    }  //end of else if( flag == 0 )

} catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
    printStackTrace();}

}

for( ;; )  //this for loop testing token traversal time
{
    try
    {
        byte[] sendData = new byte[FinalVariables.PKSIZE];

        while( grant && pollServerInCharge.request )
            yield();

        String tmp1 = processTokenAndAlternatorForTokenTraversal();

        if( !tmp1.equals("333") || firstTime )
        {
            sendData = (tmp1+serverID).getBytes();
            DatagramPacket leftOutPacket = new DatagramPacket( sendData, sendData.
                length, leftNeighborAddress, leftNeighborPort );
            sendSocket.send( leftOutPacket );
            DatagramPacket rightOutPacket = new DatagramPacket( sendData, sendData.
                length, rightNeighborAddress, rightNeighborPort );
            sendSocket.send( rightOutPacket );
        }

        byte[] recvData = new byte[FinalVariables.PKSIZE];
        DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

        try{
                        listenSocket.receive( inPacket );
        }catch(SocketTimeoutException ex ){continue;}

        String tmp = new String(inPacket.getData()).trim();
```

```
if( tmp.equals("stop") )
{
    byte[]actionData = new byte[1000];
    actionData = (Integer.toString( action )).getBytes();
    DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
        length, measureAddress, FinalVariables.TraverseActionPort);

    try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
    catch(IOException e ){}
    break;
}

if( serverID == 0 ) //special
{
    //from n-1, left neighbor, only use tkn and bn, at index 0 and 2
    if( Integer.parseInt( tmp.substring(3))==FinalVariables.ServerNumber-1 )
    {
        if( tmp.charAt(0) == '1' )
            pTkn = true;
        else if ( tmp.charAt(0) == '0' )
            pTkn = false;
        if( tmp.charAt(1) == '1' )
            leftb = true;
        else if ( tmp.charAt(1) == '0' )
            leftb = false;
        continue;
     }
     else if( Integer.parseInt( tmp.substring(3)) == 1 )//from 1,should use
        only  b0, at index 1
     {
        if( tmp.charAt(1) == '1' )
            rightb = true;
        else if ( tmp.charAt(1) == '0' )
            rightb = false;
        continue;
     }
 }
 else if( serverID == 1 ) //special
 {
     //from 0, left neighbor, only use tkn and b0, at index 0 and 1
     if( Integer.parseInt( tmp.substring(3))==0 )
     {
         if( tmp.charAt(0) == '1' )
             pTkn = true;
         else if ( tmp.charAt(0) == '0' )
             pTkn = false;
         if( tmp.charAt(1) == '1' )
             leftb = true;
         else if ( tmp.charAt(1) == '0' )
             leftb = false;
         continue;
     }
     else if( Integer.parseInt( tmp.substring(3)) == 2 )//from 2,should use
        only  b, at index 1
     {
         if( tmp.charAt(1) == '1' )
             rightb = true;
         else if ( tmp.charAt(1) == '0' )
             rightb = false;
         continue;
     }
 }
 else if( serverID == FinalVariables.ServerNumber-1 ) //special
 {
     //from 0, right neighbor, only use bn, at index 2
     if( Integer.parseInt( tmp.substring(3))==0 )
     {
         if( tmp.charAt(2) == '1' )
             rightb = true;
         else if ( tmp.charAt(2) == '0' )
             rightb = false;
```

```
                    continue;
                }
                else if( Integer.parseInt( tmp.substring(3)) == FinalVariables.
                   ServerNumber-2 )//from left neighbor,should use token and b,at index0
                   , 1
                {
                    if( tmp.charAt(0) == '1' )
                        pTkn = true;
                    else if ( tmp.charAt(0) == '0' )
                        pTkn = false;
                    if( tmp.charAt(1) == '1' )
                        leftb = true;
                    else if ( tmp.charAt(1) == '0' )
                        leftb = false;
                    continue;
                }
            }
            else  //other processes
            {
                //extract the values of preseccor's token and b
                if( Integer.parseInt( tmp.substring(3)) == (serverID -1) )
                {
                    if( tmp.charAt(0) == '1' )
                        pTkn = true;
                    else if ( tmp.charAt(0) == '0' )
                        pTkn = false;
                    if( tmp.charAt(1) == '1' )
                        leftb = true;
                    else if ( tmp.charAt(1) == '0' )
                        leftb = false;
                    continue;
                }
                else //extract the value of preseccor's b
                {
                    if( tmp.charAt(1) == '1' )
                        rightb = true;
                    else if( tmp.charAt(1) == '0' )
                        rightb = false;
                }
            }
        } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
            printStackTrace(); }
    }
}
```

APPENDIX 2 COMPANION TOKEN

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


/**
* This class implements companion token ring algorithm
*/

class CompanionToken extends Thread
{
    private InetAddress ringMakerAddress;
    private InetAddress timeMonitorAddress;
    private InetAddress measureAddress;

    public int serverID;
    public int neighborPort;
    private InetAddress neighborAddress;
    private int action;

    private DatagramSocket sendSocket;
    private DatagramSocket actionSocket;
    private DatagramSocket talkToRMSocket;   //used to talk to Ring Maker
    private DatagramSocket talkToTimeSocket; //used to talk to timeMonitor
    private DatagramSocket listenSocket;  //socket used to listen from its precessor
    private DatagramSocket talkTokenTraverseSocket; //used to talk to timeMonitor

    public  boolean grant;
    private PollServer pollServerInCharge; //proecess belongs to which application
    private boolean tkn;
    private boolean cmp;
    private boolean pTkn;
    private boolean pCmp;
    public  boolean ignore = false;  //used to control if to send msg when executing cs
    public  boolean reportedCS = false;  //used to control if to send msg when executing cs

    /**
    * constructor
    */
    public CompanionToken( InetAddress addr, int listenPort, int talkToRMport, PollServer ps
        , int sid )
    {
        ringMakerAddress = addr;
        timeMonitorAddress = addr;
        measureAddress = addr;
        pollServerInCharge = ps;
        serverID = sid;
        action = 0;

        neighborPort = -1;
        neighborAddress = null;
        grant = false;

        try
        {
            sendSocket = new DatagramSocket();
            talkToTimeSocket = new DatagramSocket();
            actionSocket = new DatagramSocket();
            talkTokenTraverseSocket = new DatagramSocket();
            listenSocket = new DatagramSocket( listenPort, InetAddress.getByName("147.26.101.
                141"));
            talkToRMSocket = new DatagramSocket(talkToRMport, InetAddress.getByName("147.26.
                101.141"));
        } catch( SocketException e ) {System.out.println("Socket: " + e.getMessage());}
          catch ( UnknownHostException e ) { System.out.println("UnknownHost: " + e.
            getMessage());}
    }
    private int boolToInt( boolean a )
```

```
{
    if( a )
        return 1;
    else
        return 0;
}


//return a string with at most 2 elements, 1st is the token, 2nd, is companion
//"1" means high, "0" means low, and 3 means no change
private String processTokenAndCcompanionForTokenTraversal()
{
    String tmp = "33";

    if ( serverID == 0 )
    {
        //case 1
        if( (tkn == pTkn) && ( tkn == true && cmp ==pCmp ) && pollServerInCharge.request
            == false )
        {

            if(ignore == false)
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();
                DatagramPacket outPacket = new DatagramPacket(sendData,
                                        sendData.length, measureAddress, FinalVariables.
                                          MeasurePort);

                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                          catch(IOException e ){}
            }
            else ignore = false;
            reportedCS = false;

            tkn = !tkn;
            cmp = !cmp;

            tmp = boolToInt(tkn) + ""+boolToInt(cmp);

            byte[] sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" release").getBytes();

            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                        sendData2.length, measureAddress, FinalVariables.
                                          MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket2);  }
            catch(IOException e ){}
            grant = false;
        }

        //case 2
        else if( tkn == pTkn && ( tkn == true && cmp == pCmp ) && pollServerInCharge.
            request == true)
        {
            if( ignore == false )  //should send msg
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                        sendData.length, measureAddress, FinalVariables.
                                          MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                catch(IOException e ){}
                //set it to true, so not to send multiple times
                ignore = true;
```

```
    }

    grant = true;

    if( reportedCS == false )
    {
        byte[]sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" cs").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket2);  }
        catch(IOException e ){}
        reportedCS = true; //reported already
    }

}

//case 3
if( (tkn == pTkn) && ( tkn == false && cmp ==pCmp ) && pollServerInCharge.request
    == false )
{

    if(ignore == false)
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);

        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
                  catch(IOException e ){}
    }
    else ignore = false;
    reportedCS = false;

    tkn = !tkn;

    tmp = boolToInt(tkn) + ""+boolToInt(cmp);

    byte[] sendData2 = new byte[10];
    sendData2 = (Integer.toString( serverID )+" release").getBytes();

    DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
    //tell TokenTraversalMonitor its privilege
    try{talkTokenTraverseSocket.send(outPacket2);  }
    catch(IOException e ){}
    grant = false;
}

//case 4
else if( tkn == pTkn && ( tkn == false && cmp == pCmp ) && pollServerInCharge.
    request == true)
{
    if( ignore == false )  //should send msg
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
```

```
            //set it to true, so not to send multiple times
            ignore = true;
        }

        grant = true;

        if( reportedCS == false )
        {
            byte[]sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" cs").getBytes();

            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket2);  }
            catch(IOException e ){}
            reportedCS = true; //reported already
        }

    }

    else if( (tkn != pTkn) && cmp == pCmp )
    {
        cmp = !cmp;   //pass only cmp
        if( cmp == true )
            tmp = "31";
        else
            tmp = "30";
    }
}
else
{
    //case 1
    if( tkn != pTkn && ( tkn == true && cmp != pCmp) && pollServerInCharge.request
        == false )
    {
        reportedCS = false;  //set for case 2

        if(ignore == false)
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )+" get").getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
            try{talkTokenTraverseSocket.send(outPacket);  }//tell
                TokenTraversalMonitor its privilege
                      catch(IOException e ){}
        }
        else ignore = false;   //set ignore to false because for case 2
        tkn = !tkn;
        cmp = !cmp;

        tmp = boolToInt(tkn) + ""+boolToInt(cmp);

        byte[] sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" release").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);

        //tell TokenTraversalMonitor its privilege released
        try{talkTokenTraverseSocket.send(outPacket2);  }
                      catch(IOException e ){}
        grant = false;
    }
    //case 2
    else if( tkn != pTkn && ( tkn == true && cmp != pCmp) && pollServerInCharge.
```

```
    request == true )
{
    if( ignore == false )
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
                  catch(IOException e ){}
        ignore = true;   //set it to true, so not to send multiple times
    }

    grant =  true ;

    if ( reportedCS == false )
    {
        byte[]sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" cs").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
        try{talkTokenTraverseSocket.send(outPacket2);  }//tell
            TokenTraversalMonitor cs
                  catch(IOException e ){}
        reportedCS = true;
    }

}


//case 3
if( tkn != pTkn && ( tkn == false && cmp != pCmp) && pollServerInCharge.request
    == false )
{
    reportedCS = false;  //set for case 2

    if(ignore == false)
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        try{talkTokenTraverseSocket.send(outPacket);  }//tell
            TokenTraversalMonitor its privilege
                  catch(IOException e ){}
    }
    else ignore = false;   //set ignore to false because for case 2
    tkn = !tkn;

    tmp = boolToInt(tkn) + ""+boolToInt(cmp);

    byte[] sendData2 = new byte[10];
    sendData2 = (Integer.toString( serverID )+" release").getBytes();

    DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);

    //tell TokenTraversalMonitor its privilege released
    try{talkTokenTraverseSocket.send(outPacket2);  }
                  catch(IOException e ){}
    grant = false;
}
//case 4
```

```
        else if( tkn != pTkn && ( tkn == false && cmp != pCmp) && pollServerInCharge.
            request == true )
        {
            if( ignore == false )
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                        sendData.length, measureAddress, FinalVariables.
                                          MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                          catch(IOException e ){}
                ignore = true;   //set it to true, so not to send multiple times
            }

            grant =  true ;

            if ( reportedCS == false )
            {
                byte[]sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" cs").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                        sendData2.length, measureAddress, FinalVariables.
                                          MeasurePort);
                try{talkTokenTraverseSocket.send(outPacket2);  }//tell
                    TokenTraversalMonitor cs
                          catch(IOException e ){}
                reportedCS = true;
            }

        }

        else if( tkn == pTkn && cmp != pCmp )
        {
            cmp = !cmp;
            if( cmp == true )
                tmp = "31";
            else
                tmp = "30";
        }
    }
    return tmp;
}



//return a string with at most 2 elements, 1st is the token, 2nd, is companion
//"1" means high, "0" means low, and 3 means no change
private String processTokenAndCcompanionNoRequest()
{
    String tmp;
    int temp = action;

    if ( serverID == 0 )
    {
        if( (tkn == pTkn) && ( (tkn == true) && (cmp == pCmp) ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
                                        sendData.length, timeMonitorAddress, FinalVariables
                                          .PrevilegePort);
            //tell timeMonitor its privilege
            try{talkToTimeSocket.send(outPacket);  }
                          catch(IOException e ){}
```

```
            tkn = !tkn;
            cmp = !cmp;
        }

        else if( (tkn == pTkn) && ( (tkn == false) && (cmp == pCmp) ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
                                       sendData.length, timeMonitorAddress, FinalVariables
                                         .PrevilegePort);
            //tell timeMonitor its privilege
            try{talkToTimeSocket.send(outPacket);  }
                        catch(IOException e ){}

            tkn = !tkn;  //only pass token
        }

        else if( (tkn != pTkn) && cmp == pCmp )
        {
            cmp = !cmp;   //pass only cmp
        }
    }
    else  //other processes
    {
        if( (tkn != pTkn) && ( (tkn == true) && (cmp != pCmp) ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
            sendData.length, timeMonitorAddress, FinalVariables.PrevilegePort);
            //tell timeMonitor its previlege
            try{talkToTimeSocket.send(outPacket);  }
                        catch(IOException e ){}

            tkn = !tkn;
            cmp = !cmp;
        }
        if( (tkn != pTkn) && ( (tkn == false) && (cmp != pCmp) ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
            sendData.length, timeMonitorAddress, FinalVariables.PrevilegePort);
            //tell timeMonitor its previlege
            try{talkToTimeSocket.send(outPacket);  }
                        catch(IOException e ){}

            tkn = !tkn;
        }
        else if( tkn == pTkn && cmp != pCmp )
        {
            // System.out.print("\nprocess "+ serverID +"  pass companion msg");
            cmp = !cmp;
        }
    }
    tmp = boolToInt(tkn)+""+boolToInt(cmp);
    if(temp!= action)
    {
        byte[]sendData = new byte[10];
        sendData = ("a").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
            timeMonitorAddress, FinalVariables.PrevilegePort);
        try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
             catch(IOException e ){}
    }
    return tmp;
}
```

```
public void run()
{
    RingInitialization init = new RingInitialization( ringMakerAddress, listenSocket,
        talkToRMSocket, serverID );

    ProcessInfo p = (ProcessInfo) init.findNeighbor();
    neighborAddress = p.address;
    neighborPort = p.lport;

    int flag = 0;  //used only when test stabilization time
    String tmp="", t="";
    for( ;; )  //this for loop testing stabilization time
    {
        try
        {
            byte[] sendData = new byte[FinalVariables.PKSIZE];

            t = processTokenAndCompanionNoRequest();
            sendData = t.getBytes();

            DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
                neighborAddress, neighborPort );

            sendSocket.send( outPacket );

            byte[] recvData = new byte[FinalVariables.PKSIZE];
            DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

            listenSocket.receive( inPacket );

            tmp = new String(inPacket.getData()).trim();
            if( tmp.equals("stop measure stabilization") )
            {
                byte[]actionData = new byte[1000];
                actionData = (Integer.toString( action )).getBytes();
                DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                    length, timeMonitorAddress, FinalVariables.ActionPort);

                //tell timeMonitor its privilege
                try{actionSocket.send(actionPacket);  }
                catch(IOException e ){}
                break;
            }
            if( flag == 1 )  //waiting for restart
            {
                if( tmp.equals("restart") )  //initialize
                {
                    flag = 0;
                    tkn = false;
                    cmp = false;
                    pTkn = false;   //precessor tkn
                    pCmp = false;
                }
            }
            else if( flag == 0 )   //must deal with the msg
            {
                if( tmp.equals( "stop" ) )
                    flag = 1;
                else
                {
                    //extract the value of preseccor's token
                    if( tmp.charAt(0) == '1' )
                        pTkn = true;
                    else if ( tmp.charAt(0) == '0' )
                        pTkn = false;

                    //extract the value of preseccor's companion
                    if( tmp.charAt(1) == '1' )
                        pCmp = true;
                    else if( tmp.charAt(1) == '0' )
```

```
                    pCmp = false;
                }
            }
        } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
            printStackTrace(); }
}


byte[] flagData = new byte[FinalVariables.PKSIZE];
flagData = "now".getBytes();

DatagramPacket nowPacket = new DatagramPacket( flagData, flagData.length,
    neighborAddress, neighborPort );
try
{
    sendSocket.send( nowPacket );
    for(; ; )
    {
        byte[] rfData = new byte[FinalVariables.PKSIZE];
        DatagramPacket rfPacket = new DatagramPacket( rfData, rfData.length);
        listenSocket.receive( rfPacket );
        String rtmp = new String(rfPacket.getData()).trim();
        if(rtmp.equals("now"))
            break;
    }
}catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
    printStackTrace(); }
tkn = false;
cmp = false;
pTkn = false;
pCmp = false;

action = 0;
for( ;; )  //this for loop testing token traversal time
{
    try
    {
        byte[] sendData = new byte[FinalVariables.PKSIZE];

        while( grant && pollServerInCharge.request )
            yield();
        //token traversal time measurement
        sendData = (processTokenAndCompanionForTokenTraversal()).getBytes();

        DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
            neighborAddress, neighborPort );

        sendSocket.send( outPacket );

        byte[] recvData = new byte[FinalVariables.PKSIZE];
        DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

        listenSocket.receive( inPacket );

        tmp = new String(inPacket.getData()).trim();

        if( tmp.equals("stop") )
        {
            byte[]actionData = new byte[1000];
            actionData = (Integer.toString( action )).getBytes();
            DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                length, measureAddress, FinalVariables.TraverseActionPort);

            try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
            catch(IOException e ){}
            break;
        }

        //extract the value of preseccor's token
        if( tmp.charAt(0) == '1' )
            pTkn = true;
```

```
                else if ( tmp.charAt(0) == '0' )
                    pTkn = false;

                //extract the value of preseccor's companion
                if( tmp.charAt(1) == '1' )
                    pCmp = true;
                else if( tmp.charAt(1) == '0' )
                    pCmp = false;

            } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
                printStackTrace(); }
        }
    }
}
```

APPENDIX 3 CURRENT POLL RESULT

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

/**
* This class is used to store the support infomation for each song
*/
class PollInfo
{
    public int songID;
    public int supportNumber;

    PollInfo()
    {
        supportNumber = 0;
        songID = -1;
    }
}

/**
* Application server, used for updating data
*/
public class CurrentPollResult
{
    private PollInfo [] pollResult;  //support number for each song


    /**
    * a constructor without any parameter
    */
    CurrentPollResult()
    {
        pollResult = new PollInfo[FinalVariables.SongSize];

        for( int i = 0; i < FinalVariables.SongSize; i++ )
        {
            pollResult[i] = new PollInfo();
            pollResult[i].songID = i;
        }
    }

    /**
    * self explanatory
    * @param s a response to be dealt with
    */
    public void updateResult( String s )
    {
        int temp = 0;
        int fromIndex = 0;
        int endIndex = 0;
        s = s + " ";  //last " " was trimmed while sent through network, so add to retreive
            number
        while( (endIndex = s.indexOf( ' ', fromIndex ) ) != -1 )
        {
            temp = Integer.parseInt( s.substring( fromIndex, endIndex ) );
            (pollResult[temp].supportNumber)++;   //update the data
            fromIndex = endIndex + 1;
        }
        //need sort the array
        Slow_Sort( pollResult, FinalVariables.SongSize );


        try {
          File f = new File("pollResult.log");  // delete the file if it already exists
          if (f.exists())
             f.delete();
```

```
      PrintWriter out = new PrintWriter(new FileWriter(f));
      out.println("The top twenty songs are: ");
      for( int i = 0; i < 20; i++ )
        out.println( pollResult[i].songID + "support number is "+pollResult[i].
            supportNumber );
      out.close();  // We're done writing
    }
    catch (IOException e) { /* Handle exceptions */ }

    int total = 0;

    //to check if the response is complete, for monitor purpose
    for( int i = 0; i < 1000; i++ )
    {
            total += pollResult[i].supportNumber;
    }
    System.out.print("\nsong number and supportNumber[i]:********" + total);

}

/**
* Utility function used by the Slow_Sort and Quick_Sort functions
*/
private void Swap( PollInfo array[ ], int p, int q )
{
    int temp = array[ p ].supportNumber ;
    array[ p ].supportNumber = array[ q ].supportNumber ;
    array[ q ].supportNumber = temp ;
}

/**
* bubble sort, upper bound is n^2
*/
public void Slow_Sort( PollInfo array[ ], int array_size )
{
    Random rand = new Random();

    //random int ranging from 0 to maxResponseTime - 1
    Integer anInt1 = new Integer( rand.nextInt( 250 ) );
    int a = anInt1.intValue();   //will range from 0 to 199

    try{Thread.sleep( a + 900 ) ;}  //set an upper bound for each process to hold
        privilege
     catch ( InterruptedException e ) { System.out.println("Interrupted: " + e.getMessage
        ());}
    for ( int pass = 0; pass < array_size - 1; pass++ )
        for ( int i = 0; i < array_size - 1 - pass; i++ )
            if ( array[i].supportNumber < array[ i + 1].supportNumber)
                Swap ( array, i, i + 1 );
}


/**
* Accept and deal with responses from poll server
*/
public static void main( String[] args )
{
    DatagramSocket socket = null;
    CurrentPollResult result = new CurrentPollResult();

    try
    {
        socket = new DatagramSocket ( FinalVariables.PollResultPort );

        //write output to this file, record which process sends result
        File f = new File("resultSending.log");
        // delete the file if it already exists
        if (f.exists())
            f.delete();

        PrintWriter out = new PrintWriter(new FileWriter(f));
```

```
        System.out.println( "Application Server Started" );

        for(;;)
        {
            byte[] recvData = new byte[FinalVariables.PKSIZE];
            DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

            byte[] sendData = new byte[1000];
            sendData = "OK".getBytes();

            socket.receive( inPacket );
            String tmp = new String(inPacket.getData()).trim();

            out.println(" received from port " + inPacket.getPort());

            DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
                inPacket.getAddress(), inPacket.getPort() );

            result.updateResult( tmp );
            //to notify that update was done
            socket.send( outPacket );
        }
    } catch ( SocketException e ) { System.out.println("Socket: " + e.getMessage());}
      catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
      finally { if(socket != null ) socket.close(); /*out.close();*/ }
}
```

APPENDIX 4 ENABLER TOKEN

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

/**
* This class implements enabler token ring algorithm
*/

class EnablerToken extends Thread
{
    private InetAddress ringMakerAddress;
    private InetAddress timeMonitorAddress;
    private InetAddress measureAddress;

    public int serverID;
    public int neighborPort;
    private InetAddress neighborAddress;
    private int action;

    private DatagramSocket sendSocket;
    private DatagramSocket actionSocket;
    private DatagramSocket talkToRMSocket;   //used to talk to Ring Maker
    private DatagramSocket talkToTimeSocket; //used to talk to timeMonitor
    private DatagramSocket listenSocket;  //socket used to listen from its precessor
    private DatagramSocket talkTokenTraverseSocket; //used to talk to timeMonitor

    public  boolean grant;
    private PollServer pollServerInCharge; //proecess belongs to which application,used to
        read poll server's request
    public  boolean ignore = false;  //used to control if to send msg when executing cs
    public  boolean reportCS = false;  //used to control if to send msg when executing cs

    //own token, rd and enabler
    private boolean tkn;
    private boolean rd;
    private boolean en;

    //precessor's token and shepherd
    private boolean pTkn;
    private boolean pEn;

    public EnablerToken( InetAddress addr, int listenPort, int talkToRMport, PollServer ps,
        int sid )
    {
        ringMakerAddress = addr;
        timeMonitorAddress = addr;
        measureAddress = addr;
        pollServerInCharge = ps;
        action=0;

        neighborPort = -1;
        neighborAddress = null;
        serverID = sid;
        grant = false;

        try
        {
            sendSocket = new DatagramSocket();
            actionSocket = new DatagramSocket();
            talkToTimeSocket = new DatagramSocket();
            talkTokenTraverseSocket = new DatagramSocket();
            listenSocket = new DatagramSocket( listenPort, InetAddress.getByName("147.26.101.
                141"));

            talkToRMSocket = new DatagramSocket(talkToRMport, InetAddress.getByName("147.26.
                101.141"));
        } catch( SocketException e ) {System.out.println("Socket: " + e.getMessage());}
```

```
      catch ( UnknownHostException e ) { System.out.println("UnknownHost: " + e.
        getMessage());}
}


private int boolToInt( boolean a )
{
    if( a )
        return 1;
    else
        return 0;
}

private String processTokenAndEnablerForTokenTraversal()
{
    String tmp = "33";

    if ( serverID == 0 )
    {
        //X action: allow an enabler to be passed
        if( tkn != pTkn && en == pEn )
        {
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;
        }

        //Y action: allow an enabler to be passed
        else if( en == pEn && tkn == pTkn && !rd && !tkn )
        {
            en = !en;  //only pass token
            rd = true;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;
        }

        //Z action case 1: allow both token and enabler to be passed
        else if( tkn == pTkn && en == pEn && ( rd || tkn ) && pollServerInCharge.request
            == false )
        {
            if( ignore == false )
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();
                System.out.println("after get process 0");
                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.
                                      MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                          catch(IOException e ){}
            }
            else ignore = false;   //set ignore to false because for case 2
            reportCS = false; //set for case 2

            tkn = !tkn;
            en = !en;
            rd = false;


            byte[] sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" release").getBytes();
            //System.out.println("after release process 0");
            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.
                                      MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket2);  }
                          catch(IOException e ){}
            grant = false;
```

```
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

        }
        //Z action case 2: when request is true
        else if( tkn == pTkn && en == pEn && ( rd || tkn ) && pollServerInCharge.request ↙
            ==true )
        {
            if( ignore == false )  //should send msg
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                        sendData.length, measureAddress, FinalVariables.   ↙
                                          MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                          catch(IOException e ){}

                ignore = true;
            }
            grant = true;

            if( reportCS == false )
            {
                byte[]sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" cs").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                        sendData2.length, measureAddress, FinalVariables.  ↙
                                          MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket2);  }
                catch(IOException e ){}
                System.out.println("after cs "+serverID );
                reportCS = true; //reported already
            }
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;
        }

    }
    else
    {
        //X action: allow an enabler to be passed
        if( tkn == pTkn && en != pEn )
         {
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

         }

        //Y action: allow an enabler to be passed
        else if( tkn != pTkn && en != pEn && !rd && !tkn )
        {
              en = !en;  //only pass enabler
              rd = true;
              tmp=boolToInt(tkn)+""+boolToInt(en);
              return tmp;

        }

        //Z action case 1: allow both token and enabler to be passed
        else if( tkn != pTkn && en != pEn && ( rd || tkn ) && pollServerInCharge.request ↙
            == false )
        {
            if( ignore ==  false )
            {
```

```
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();
                DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.
                                        MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                catch(IOException e ){}
            }
            else ignore = false;   //set ignore to false
            reportCS = false; //set for case 2

            tkn = !tkn;
            en = !en;
            rd = false;

            byte[] sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" release").getBytes();

            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket2);  }
            catch(IOException e ){}
            grant = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

        }
        //Z action case 2: when request is true
        else if( tkn != pTkn && en != pEn && ( rd || tkn ) && pollServerInCharge.request
            == true )
        {
            if( ignore == false )  //should send msg
            {
                byte[]sendData = new byte[10];
                sendData = (Integer.toString( serverID )+" get").getBytes();

                DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket);  }
                catch(IOException e ){}
                ignore = true;
            }

            grant = true;
            if( reportCS == false )
            {
                byte[]sendData2 = new byte[10];
                sendData2 = (Integer.toString( serverID )+" cs").getBytes();

                DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
                //tell TokenTraversalMonitor its privilege
                try{talkTokenTraverseSocket.send(outPacket2);  }
                catch(IOException e ){}
                System.out.println("after cs "+serverID );
                reportCS = true; //reported already
            }
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;
        }
    }
    return tmp;
}
```

```
//return a string with at most 2 elements, 1st is the token, 2nd is shepherd,
//"1" means high, "0" means low, and 3 means not passing
private String processTokenAndEnablerNoRequest()
{
    String tmp = "33";
    int temp = action;

    if ( serverID == 0 )
    {
        //X action: allow an enabler to be passed
        if( tkn != pTkn && en == pEn )
        {
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;
        }

        //Y action: allow an enabler to be passed
        else if( en == pEn && tkn == pTkn && !rd && !tkn )
        {
            en = !en;  //only pass token
            rd = true;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

        }

        //Z action: allow both token and enabler to be passed
        else if( tkn == pTkn && en == pEn && ( rd || tkn ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
                timeMonitorAddress, FinalVariables.PrevilegePort);
            try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
             catch(IOException e ){}

            tkn = !tkn;
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

        }

    }
    else
    {
        //X action: allow an enabler to be passed
        if( tkn == pTkn && en != pEn )
         {
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

         }

        //Y action: allow an enabler to be passed
        else if( tkn != pTkn && en != pEn && !rd && !tkn )
        {
              en = !en;  //only pass enabler
              rd = true;
              tmp=boolToInt(tkn)+""+boolToInt(en);
              return tmp;

        }
```

```
        //Z action: allow both token and enabler to be passed
        else if( tkn != pTkn && en != pEn && ( rd || tkn ) )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )).getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
                timeMonitorAddress, FinalVariables.PrevilegePort);
            try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
             catch(IOException e ){}

            tkn = !tkn;
            en = !en;
            rd = false;
            tmp=boolToInt(tkn)+""+boolToInt(en);
            return tmp;

        }

    }  //end of else in line 266
    if(temp!=action)
    {
        byte[]sendData = new byte[10];
        sendData = ("a").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
            timeMonitorAddress, FinalVariables.PrevilegePort);
        try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
             catch(IOException e ){}
    }

    return tmp;
}


public void run()
{
    RingInitialization init = new RingInitialization( ringMakerAddress, listenSocket,
        talkToRMSocket, serverID );

    ProcessInfo p = (ProcessInfo) init.findNeighbor();
    neighborAddress = p.address;
    neighborPort = p.lport;

    int flag = 0;  //used only when test stabilization time

    for( ;; )
    {
        try
        {
            byte[] sendData = new byte[FinalVariables.PKSIZE];

            sendData = ( processTokenAndEnablerNoRequest() ).getBytes();

            DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
                neighborAddress, neighborPort );

            sendSocket.send( outPacket );

            byte[] recvData = new byte[FinalVariables.PKSIZE];
            DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

            listenSocket.receive( inPacket );

            String tmp = new String(inPacket.getData()).trim();
            if( tmp.equals("stop measure stabilization") )
            {

                byte[]actionData = new byte[1000];
                actionData = (Integer.toString( action )).getBytes();
                DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
```

```
                length, timeMonitorAddress, FinalVariables.ActionPort);

            try{actionSocket.send(actionPacket);  }
            catch(IOException e ){}
            break;
        }
        if( flag == 1 )  //waiting for restart
        {
            if( tmp.equals("restart") )  //initialize
            {
                flag = 0;
                tkn = false;
                en = false;
                pTkn = false;   //precessor tkn
                pEn = false;
                rd = false;
            }
        }
        else if( flag == 0 )   //must deal with the msg
        {
            if( tmp.equals( "stop" ) )
                flag = 1;
            else
            {
                //extract the value of preseccor's token
                if( tmp.charAt(0) == '1' )
                    pTkn = true;
                else if ( tmp.charAt(0) == '0' )
                    pTkn = false;

                //extract the value of preseccor's companion
                if( tmp.charAt(1) == '1' )
                    pEn = true;
                else if( tmp.charAt(1) == '0' )
                    pEn = false;
            }
        }
    } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
        printStackTrace(); }
}

byte[] flagData = new byte[FinalVariables.PKSIZE];
flagData = "now".getBytes();

DatagramPacket nowPacket = new DatagramPacket( flagData, flagData.length,
    neighborAddress, neighborPort );
try
{
    sendSocket.send( nowPacket );
    for(; ; )
    {
        byte[] rfData = new byte[FinalVariables.PKSIZE];
        DatagramPacket rfPacket = new DatagramPacket( rfData, rfData.length);
        try{
        listenSocket.receive( rfPacket );
    }
    catch(SocketTimeoutException c){continue;}
        String rtmp = new String(rfPacket.getData()).trim();
        if(rtmp.equals("now"))
            break;
    }
}catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
    printStackTrace(); }
tkn = false;
rd = false;
en = false;
pTkn = false;
pEn = false;

action = 0;
for( ;; )  //this for loop testing token traversal time
```

```
    {
        try
        {
            byte[] sendData = new byte[FinalVariables.PKSIZE];

            while( grant && pollServerInCharge.request )
                yield();
            //token traversal time measurement
            String tt= processTokenAndEnablerForTokenTraversal();
            sendData = (tt).getBytes();

            DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
                neighborAddress, neighborPort );

            sendSocket.send( outPacket );

            byte[] recvData = new byte[FinalVariables.PKSIZE];
            DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

            listenSocket.receive( inPacket );

            String tmp = new String(inPacket.getData()).trim();
            if( tmp.equals("stop") )
            {
                byte[]actionData = new byte[1000];
                actionData = (Integer.toString( action )).getBytes();
                DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                    length, measureAddress, FinalVariables.TraverseActionPort);

                try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
                catch(IOException e ){}
                break;
            }
            //extract the value of preseccor's token
            if( tmp.charAt(0) == '1' )
                pTkn = true;
            else if ( tmp.charAt(0) == '0' )
                pTkn = false;

            //extract the value of preseccor's companion
            if( tmp.charAt(1) == '1' )
                pEn = true;
            else if( tmp.charAt(1) == '0' )
                pEn = false;

        } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
            printStackTrace(); }
    }
}
```

APPENDIX 5 FINAL VARIABLES

```
mport java.io.*;
mport java.net.*;


**
 This class is used to define some constant variables used by other files
 @version v0.0.0
 @author Rong Wang
/
ıblic class FinalVariables {

   //following three are for RingMaker.java and RingInitialization
   public static final int RingMakerPort = 9185;
   public static final int PKSIZE = 1000;
   public static final int ServerNumber = 16;

   //follwoing are for TaskDistribution.java
   public static final int MaxTaskNumber = 1000;
   public static final int MaxProcessNumber = ServerNumber;

   //following are from TimeMonitor.java
   public static final int TimePort = 9188;
   public static final int PrevilegePort = 9189;
   public static final int InfoPort = 9190;
   public static final int InfoPort2 = 9193;
   public static final int ActionPort = 9191;
   public static final int TraverseActionPort = 9192;
   public static final int MaxRunForStabilization = 1;

   //follwoing is for PollServer.java
   public static final int ThreadNumber = 4;

   //following are for CurrentPollResult.java
   public static final int PollResultPort = 9186;
   public static final int SongSize = 1000;

   public static final int MeasurePort = 9901;
```

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

*
This class is used to simulate the terminal of the application
@version v0.0.0
@author Rong Wang
'
public class PollServer extends Thread {

static final int MachineID = 0;    //need modify according to computer ID
public static int[][] taskDistribution;  //indicates sleeping time
public static long totalTime=0;
public static int countWait=0;
public boolean request;

private InetAddress ringMakerAddress;
private InetAddress pollResultAddress;
private int listenPort;
private int talkToRMPort;
private int pollServerID;
private DatagramSocket aSocket; //will be used by the application, not for token

public PollServer( int lport, int rmport, int sid) {
 try
 {    ringMakerAddress = InetAddress.getByName("147.26.101.144");
      pollResultAddress = InetAddress.getByName("147.26.101.144");
      listenPort = lport;
      talkToRMPort = rmport;
      pollServerID = sid;
      request = false;
      aSocket = new DatagramSocket();
  } catch (SocketException e) { System.out.println("Socket: " + e.getMessage());  }
  catch (UnknownHostException e) { System.out.println("UnknownHost: " + e.getMessage());}
}

//simulate response from people by generating 20 numbers range from 1 to 1000
public String generateResponse()
{
  Random rand = new Random();
  int n = 999;
  String aLine = "";

  for (int i = 0; i < 20; i++) {
    Integer anInt = new Integer(rand.nextInt(n + 1));
    aLine = aLine + anInt + " ";
  }
  return aLine;
}


public void run()
{
  int count = 0;
  int[][] taskDistribution = new int[FinalVariables.MaxTaskNumber][FinalVariables.
      MaxProcessNumber];
  long start=0, end = 0;

   //first read file taskDistribution.txt and store the content in an interger array
  try
  {
    String line;

    BufferedReader in = new BufferedReader(new FileReader(
        "taskDistribution.txt"));

    for (int i = 0; i < FinalVariables.MaxTaskNumber; i++)
```

```
  {
    line = in.readLine(); //read a line from the file
    int temp = 0;
    int fromIndex = 0;
    int endIndex = 0;
    for (int j = 0; j < FinalVariables.MaxProcessNumber; j++)
    {
        if( (endIndex = line.indexOf( ' ', fromIndex )) != -1 );
        {
            temp = Integer.parseInt( line.substring( fromIndex, endIndex ) );
            taskDistribution[i][j] = temp;
            fromIndex = endIndex + 1;
        }
    }//end of outer for (j=0) loop
  }//end of outer for (i=0) loop
}//end of try
catch (FileNotFoundException e){ System.out.println("FileNotFoundEXCEPTION: " + e.
    getMessage());}
catch (IOException e) {System.out.println("IOEXCEPTION: " + e.getMessage());}


//following three lines should be commented/uncommented according to which algorithm to
    test

//ShepherdToken token = new ShepherdToken(ringMakerAddress, listenPort,talkToRMPort, this
    , pollServerID );
//CompanionToken token = new CompanionToken(ringMakerAddress, listenPort,talkToRMPort,
    this, pollServerID );
//EnablerToken token = new EnablerToken(ringMakerAddress, listenPort,talkToRMPort, this,
    pollServerID );
AlternatorToken token = new AlternatorToken(ringMakerAddress, listenPort,talkToRMPort,
    this, pollServerID );

token.start();  //begin to listen token

/* //following two lines should be uncommented if test S, C or E
while (token.neighborPort == -1 )
    yield(); //waiting until the process has been assigned neighbors
*/

//following two lines should be commented if test S, C or E
while (token.leftNeighborPort == -1 || token.rightNeighborPort == -2)
  yield(); //waiting until the process has been assigned neighbors

for (; ; )
{

     //then it should do some non-CS chore
     if( count == FinalVariables.MaxTaskNumber)
        count=0;

     while (count < FinalVariables.MaxTaskNumber)
     {

        String response = "";
        //non-CS, simulate different user's different response time and idle time
        try
        {
            Thread.sleep( taskDistribution[count][token.serverID] );
        }
        catch ( InterruptedException e ) { System.out.println("Interrupted: " + e.
            getMessage());}

        response = generateResponse();

        byte[] sendData = new byte[1000];
        sendData = response.getBytes();

        byte[] recvData = new byte[1000];
        DatagramPacket inPacket = new DatagramPacket(recvData, recvData.length);
```

```
            DatagramPacket outPacket = new DatagramPacket(sendData, sendData.length,
                pollResultAddress, FinalVariables.PollResultPort);

            request = true;
            start=System.currentTimeMillis();
            // then request to update the output file
            while ( token.grant != true)
                yield();
            totalTime += System.currentTimeMillis()-start;
            countWait++;
            System.out.println("Average waiting time   "+((double)totalTime/countWait) );

            try
            {
                aSocket.send(outPacket);
                aSocket.receive(inPacket);

               String tmp = new String(inPacket.getData()).trim();

               if (tmp.equals("OK")) //reset request to false


                request = false;
                while ( token.grant == true)
                    yield();
            } catch (IOException e) {}
            count++; //update count
        } //end of while
      }//end of for
    }//end of run()

    public static void main(String[] args)
    {
        //create poll servers
        for (int i = 0; i < FinalVariables.ThreadNumber; i++)
        {
          System.out.println("poll server started");
          new PollServer(2200 + MachineID+4*i, 2300 + MachineID+4*i, MachineID+4*i).start();
        }
    }  //end of main
    //end of class
```

APPENDIX 7 RING INITIALIZATION FOR ALTERNATOR TOKEN RING

```
mport java.net.*;
mport java.io.*;

**
 This class is used to help create ring, used by alternator token ring only
/

lass RingInitialization

   private ProcessInfo serverInfo;
   private InetAddress ringMakerAddress;
   public DatagramSocket talkRMSocket;
   public DatagramSocket listenSocket;

   public RingInitialization( InetAddress aHost, DatagramSocket listensocket ,
       DatagramSocket talkRMsocket, int sid)
   {
       serverInfo = new ProcessInfo( listensocket.getLocalAddress(), listensocket.
           getLocalPort(), talkRMsocket.getLocalPort(), sid );
       ringMakerAddress = aHost;
       talkRMSocket = talkRMsocket;
       listenSocket = listensocket;
   }

   public void findNeighbor( ProcessInfo [] p )
   {
       DatagramSocket socket = null;
       try
       {
           socket = new DatagramSocket();
       } catch( SocketException e ) {System.out.println("Socket: " + e.getMessage());}

       //send own info. to the RingMakerServer
       TransportTool.sendTo( serverInfo, ringMakerAddress, FinalVariables.RingMakerPort,
           socket );

       //should get neighbor's address and port and its own id
       for( int i = 0; i < 2; i++ )
           p[i] = (ProcessInfo) TransportTool.receiveFrom( talkRMSocket );
   }
```

APPENDIX 8 RING INITIALIZATION FOR E, S AND C

```
mport java.net.*;
mport java.io.*;

**
 This class is used to help create ring, used by S, C, and E token ring
/

lass RingInitialization

   private ProcessInfo serverInfo;
   private InetAddress ringMakerAddress;
   public DatagramSocket talkRMSocket;
   public DatagramSocket listenSocket;

   public RingInitialization( InetAddress aHost, DatagramSocket listensocket ,
       DatagramSocket talkRMsocket, int sid)
   {
       serverInfo = new ProcessInfo( listensocket.getLocalAddress(), listensocket.
           getLocalPort(), talkRMsocket.getLocalPort(), sid );
       ringMakerAddress = aHost;
       talkRMSocket = talkRMsocket;
       listenSocket = listensocket;
   }

   public ProcessInfo findNeighbor()
   {
       DatagramSocket socket = null;
       try
       {
           socket = new DatagramSocket();
       } catch( SocketException e ) {System.out.println("Socket: " + e.getMessage());}

       //send own info. to the RingMakerServer
       TransportTool.sendTo( serverInfo, ringMakerAddress, FinalVariables.RingMakerPort,
           socket );

       //should get neighbor's address and port and its own id
       ProcessInfo p = (ProcessInfo) TransportTool.receiveFrom( talkRMSocket );
       return p;
   }
```

APPENDIX 9 RING MAKER FOR ALTERNATOR

```
nport java.io.*;
nport java.net.*;
nport java.util.*;


**
 This class is used to store the information of every process who wants to join the token ring,
 it implements the interface Serializable so that its object can be transport over the network.
 @version v0.0.0
 @author Rong Wang
/
lass ProcessInfo implements Serializable

   public InetAddress address;
   public int lport;
   public int notifyNeighborPort;
   public int serverID;

   /**
   * a constructor without any parameter
   */
   ProcessInfo()
   {
       address = null;
       lport = -1;
       notifyNeighborPort = -2;
       serverID = -1;
   }

   /**
   * a constructor used for initialization
   * @param addr the address where the process is running
   * @param aPort the port number of the process
   */
   ProcessInfo( InetAddress addr, int aPort , int forRmport, int sid )
   {
       address = addr;
       lport = aPort;
       notifyNeighborPort = forRmport;
       serverID = sid;
   }

   /**
   * impelment the method of interface Serializable
   */
   private void writeObject(java.io.ObjectOutputStream out)
        throws IOException
        {   out.defaultWriteObject(); }

   /**
   * impelment the method of interface Serializable
   */
   private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException
        {   in.defaultReadObject(); }



**
 This class is used to create a ring so that every process who joins the ring can know to whom,
 it should communicate with, it is created for alternator token ring where two neighbors
 need to be assigned.
 @version v0.0.0
 @author Rong Wang
```

```
/

ublic class RingMaker


   private ProcessInfo [] processes;

   /**
   * a constructor without any parameter
   */
   RingMaker()
   {
       processes = new ProcessInfo[FinalVariables.ServerNumber];
       for( int i = 0; i < FinalVariables.ServerNumber; i++ )
           processes[i] = new ProcessInfo();
   }

   /**
   * self explanatory
   * @param p a process to be added
   */
   public synchronized void addProcess( ProcessInfo p )
   {
       processes[p.serverID] = p;  //add to array according to its ID number
   }

   /**
   * Assign neighbors to each process
   */
   public void assignNeighbor()
   {
       ProcessInfo process, talkToNeighbor;
       DatagramSocket aSocket, socket2, socket3;
       ObjectOutputStream os;

       try{
           aSocket = new DatagramSocket();
           socket2 = new DatagramSocket();
           socket3 = new DatagramSocket();

           for( int i =  0; i < FinalVariables.ServerNumber;  i++ )
           {
               //tell timeMonitor and token traverse monitor the address info. of each      ↙
                   process
               TransportTool.sendTo( processes[i], InetAddress.getByName("localhost"),      ↙
                   FinalVariables.InfoPort, socket2 );
               TransportTool.sendTo( processes[i], InetAddress.getByName("localhost"),      ↙
                   FinalVariables.InfoPort2, socket3 );

               //tell the process who are its talk-to-neighbor, processes[i+1]
               TransportTool.sendTo( processes[(i+1)%FinalVariables.ServerNumber], processes↙
                   [i].address, processes[i].notifyNeighborPort, aSocket );

               TransportTool.sendTo( processes[(i-1+FinalVariables.ServerNumber)%          ↙
                   FinalVariables.ServerNumber], processes[i].address, processes[i].       ↙
                   notifyNeighborPort, aSocket );

           }
       } catch ( SocketException e ) { System.out.println("Socket: " + e.getMessage());}
         catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}

   }



   public static void main( String[] args )
   {
       RingMaker ringMaker = new RingMaker();
       DatagramSocket socket = null;
       int count = 0, countSyn = 0;
       byte[] sendData = new byte[100];
```

```
    try{
        socket = new DatagramSocket ( FinalVariables.RingMakerPort );
        while( count < FinalVariables.ServerNumber)
        {
            ProcessInfo p = (ProcessInfo) TransportTool.receiveFrom( socket );
            ringMaker.addProcess(p);
            count++;
        }

        ringMaker.assignNeighbor();

        //send msg to TimeMonitor to notify the time when the ring was established
        long setupTime = (new Date()).getTime();
        sendData = (Long.toString( setupTime )).getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                    sendData.length, InetAddress.getByName("localhost"), FinalVariables↙
                      .TimePort);
        socket.send(outPacket);
    } catch ( SocketException e ) { System.out.println("Socket: " + e.getMessage());}
      catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
      finally { if(socket != null ) socket.close(); }
}
```

```
mport java.io.*;
mport java.net.*;
mport java.util.*;


**
 This class is used to store the information of every process who wants to join the token ring,
 it implements the interface Serializable so that its object can be transport over the network.
 @version v0.0.0
 @author Rong Wang
/
lass ProcessInfo implements Serializable

   public InetAddress address;
   public int lport;
   public int notifyNeighborPort;
   public int serverID;

   /**
   * a constructor without any parameter
   */
   ProcessInfo()
   {
       address = null;
       lport = -1;
       notifyNeighborPort = -2;
       serverID = -1;
   }

   /**
   * a constructor used for initialization
   * @param addr the address where the process is running
   * @param aPort the port number of the process
   */
   ProcessInfo( InetAddress addr, int aPort , int forRmport, int sid )
   {
       address = addr;
       lport = aPort;
       notifyNeighborPort = forRmport;
       serverID = sid;
   }

   /**
   * impelment the method of interface Serializable
   */
   private void writeObject(java.io.ObjectOutputStream out)
        throws IOException
        {   out.defaultWriteObject(); }

   /**
   * impelment the method of interface Serializable
   */
   private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException
        {   in.defaultReadObject(); }



**
 This class is used to create a ring so that every process who joins the ring can know to whom,
 it should communicate with. It is used by the shepherd, companion and enabler token ring
 @version v0.0.0
 @author Rong Wang
/
```

```
ublic class RingMaker


   private ProcessInfo [] processes;

   /**
   * a constructor without any parameter
   */
   RingMaker()
   {
       processes = new ProcessInfo[FinalVariables.ServerNumber];
       for( int i = 0; i < FinalVariables.ServerNumber; i++ )
           processes[i] = new ProcessInfo();
   }

   /**
   * self explanatory
   * @param p a process to be added
   */
   public synchronized void addProcess( ProcessInfo p )
   {
       processes[p.serverID] = p;  //add to array according to its ID number
/          System.out.println("received ports: "+p.lport+p.notifyNeighborPort);

   }

   /**
   * Assign neighbor to each process
   */
   public void assignNeighbor(ProcessInfo processes[])
   {
       ProcessInfo process, talkToNeighbor;
       DatagramSocket aSocket, socket2, socket3;
       ObjectOutputStream os;

       try{
           aSocket = new DatagramSocket();
           socket2 = new DatagramSocket();
           socket3 = new DatagramSocket();

           //tell timeMonitor the address info. of process ServerNumber - 1, "localhonst"
               here is the machine timeMonitor is running on
           TransportTool.sendTo( processes[FinalVariables.ServerNumber - 1], InetAddress.
               getByName("localhost"), FinalVariables.InfoPort, socket2 );
           TransportTool.sendTo( processes[FinalVariables.ServerNumber - 1], InetAddress.
               getByName("localhost"), FinalVariables.InfoPort2, socket3 );
           TransportTool.sendTo( processes[0], processes[FinalVariables.ServerNumber - 1].
               address, processes[FinalVariables.ServerNumber - 1].notifyNeighborPort,
               aSocket );


           for( int i = FinalVariables.ServerNumber - 2; i >= 0;  i-- )
           {
               //tell timeMonitor the address info. of each process
               TransportTool.sendTo( processes[i], InetAddress.getByName("localhost"),
                   FinalVariables.InfoPort, socket2 );
               TransportTool.sendTo( processes[i], InetAddress.getByName("localhost"),
                   FinalVariables.InfoPort2, socket3 );
               //tell the process who is its talk-to-neighbor, processes[i+1]
               TransportTool.sendTo( processes[i+1], processes[i].address, processes[i].
                   notifyNeighborPort, aSocket );
           }
       } catch ( SocketException e ) { System.out.println("Socket: " + e.getMessage());}
         catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}

   }


   public static void main( String[] args )
   {
```

```
      RingMaker ringMaker = new RingMaker();
      DatagramSocket socket = null;
      int count = 0;
      byte[] sendData = new byte[100];

      try{
          socket = new DatagramSocket ( FinalVariables.RingMakerPort );

          while( count != FinalVariables.ServerNumber )
          {
              ringMaker.addProcess( (ProcessInfo) TransportTool.receiveFrom( socket ) );
              count++;
          }
  System.out.println("received ports: "+ count);
          ringMaker.assignNeighbor(ringMaker.processes);

          //send msg to TimeMonitor to notify the time when the ring was established
          long setupTime = (new Date()).getTime();
          sendData = (Long.toString( setupTime )).getBytes();

          DatagramPacket outPacket = new DatagramPacket(sendData,
                        sendData.length, InetAddress.getByName("localhost"), FinalVariables↙
                          .TimePort);
          socket.send(outPacket);

      } catch ( SocketException e ) { System.out.println("Socket: " + e.getMessage());}
        catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
        finally { if(socket != null ) socket.close(); }
  } //end of main
//end of class
```

APPENDIX 11 SHEPHERD TOKEN

```
mport java.io.*;
mport java.net.*;
mport java.lang.*;

**
 This class implements shepherd token ring algorithm
/
lass ShepherdToken extends Thread

   private InetAddress ringMakerAddress;
   private InetAddress timeMonitorAddress;
   private InetAddress measureAddress;
   public static int countMsg=0;
   public long startCycle;
   public long cycleTime=0;

   public int serverID;
   public int neighborPort;
   private InetAddress neighborAddress;
   private int action;
   private int firstGet;

   private DatagramSocket sendSocket;
   private DatagramSocket actionSocket;
   private DatagramSocket talkToRMSocket; //used to talk to Ring Maker
   private DatagramSocket talkToTimeSocket; //used to talk to timeMonitor
   private DatagramSocket listenSocket;  //socket used to listen from its precessor
   private DatagramSocket talkTokenTraverseSocket; //used to talk to timeMonitor

   public  boolean grant;
   private boolean inCS;
   private PollServer pollServerInCharge;  //used to read poll server's request

   //own token, rd and shepherd
   private boolean tkn;
   private boolean rd;     //local guard
   private boolean sh;

   //precessor's token and shepherd
   private boolean pTkn;
   private boolean pSh;
   public  boolean ignore = false;  //used to control if to send msg when executing cs
   public  boolean reportCS = false;  //used to control if to send msg when executing cs

   public ShepherdToken( InetAddress addr, int listenPort, int talkToRMport, PollServer ps, ↙
       int sid )
   {
       ringMakerAddress = addr;
       timeMonitorAddress = addr;
       measureAddress = addr;
       pollServerInCharge = ps;
       action =0;
       inCS = false;
       startCycle=0;
       firstGet =0;

       neighborPort = -1;
       neighborAddress = null;
       serverID = sid;
       grant = false;

      //worst case for stabilization
       rd = true;
       if( (serverID+1)%4 == 1 )
       {
           tkn = true;
           sh = true;
       }
```

```
    if( (serverID +1)% 4== 2 )
    {
        pTkn = true;
        pSh = true;
    }
    try
    {
        sendSocket = new DatagramSocket();
        actionSocket = new DatagramSocket();
        talkToTimeSocket = new DatagramSocket();
        talkTokenTraverseSocket = new DatagramSocket();
        listenSocket = new DatagramSocket( listenPort, InetAddress.getByName("147.26.101.↙
            141"));
        talkToRMSocket = new DatagramSocket(talkToRMport, InetAddress.getByName("147.26. ↙
            101.141"));
    } catch( SocketException e ) {System.out.println("Socket: " + e.getMessage());}
      catch ( UnknownHostException e ) { System.out.println("UnknownHost: " + e.        ↙
        getMessage());}
}


//token is present at instant node
boolean Tvalue( int i, boolean tkn, boolean pTkn )
{
    return (  ((i == 0)&&(tkn == pTkn)) || ((i != 0)&&(tkn==!pTkn))  );
}

//shepherd is present at instant node
boolean Svalue( int i, boolean sh, boolean pSh )
{
    return  (  ((i==0)&&(sh==pSh))  || ((i!=0)&&(sh==!pSh))   );
}

private int boolToInt( boolean a )
{
    if( a )
        return 1;
    else
        return 0;
}


//return a string with at most 2 elements, 1st is the token, 2nd is shepherd,
//"1" means high, "0" means low, and 3 means no change
private String processTokenAndShepherdForTokenTraversal()
{
    String tmp = "33";


    //first situation case 1:
    if( Tvalue( serverID, tkn, pTkn ) && !Svalue( serverID, sh, pSh ) && ( sh || serverID↙
        !=0) && pollServerInCharge.request == false )
    {
        if( ignore == false )
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )+" get").getBytes();
            DatagramPacket outPacket = new DatagramPacket(sendData,
                                       sendData.length, measureAddress, FinalVariables.   ↙
                                         MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket);  }
             catch(IOException e ){}
         }
        ignore = false;   //set ignore to false
        reportCS = false; //set for case 2


        tkn = !tkn;  //only pass token
        rd = false;
        if( tkn )   tmp = "13";
```

```
    else    tmp = "03";

    byte[] sendData2 = new byte[10];
    sendData2 = (Integer.toString( serverID )+" release").getBytes();
    DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
    //tell TokenTraversalMonitor its privilege
    try{talkTokenTraverseSocket.send(outPacket2);  }
    catch(IOException e ){}
    grant = false;
}

//first situation case 2:
if( Tvalue( serverID, tkn, pTkn ) && !Svalue( serverID, sh, pSh ) && ( sh || serverID
    !=0) && pollServerInCharge.request == true )
{

    if( ignore == false )  //should send msg
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
    }
    ignore = true;   //set it to true, so not to send multiple times

    grant = true;

    if( reportCS == false )
    {
        byte[]sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" cs").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket2);  }
        catch(IOException e ){}
    }
    reportCS = true; //reported already
}

//second situation case 1:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && !sh && !rd
    && pollServerInCharge.request == false)
{
    if( ignore == false )
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        try{talkTokenTraverseSocket.send(outPacket);  }//tell TokenTraversalMonitor
            its privilege
         catch(IOException e ){}
     }

    ignore = false;   //set ignore to false because for case 2
    reportCS = false; //set for case 2

    tkn = !tkn;  //only pass token
    rd = true;
    if( tkn )  tmp = "13";
```

```
    else  tmp = "03";

    byte[] sendData2 = new byte[10];
    sendData2 = (Integer.toString( serverID )+" release").getBytes();
    DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
    try{talkTokenTraverseSocket.send(outPacket2);  }//tell TokenTraversalMonitor its
        privilege
        catch(IOException e ){}
    grant = false;
}

//second situation case 2:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && !sh && !rd
    && pollServerInCharge.request == true)
{
    if( ignore == false )  //should send msg
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
    }
    ignore = true;   //set it to true, so not to send multiple times
    grant = true;

    if( reportCS == false )
    {
        byte[]sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" cs").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket2);  }
        catch(IOException e ){}
    }
    reportCS = true; //reported already
}

//third situation case 1:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && ( (sh&&
    serverID!=0)||(!sh&&rd) ) && pollServerInCharge.request == false)
{
    if( ignore == false )
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
     }

    ignore = false;   //set ignore to false because for case 2
    reportCS = false; //set for case 2

    tkn = !tkn;  //pass token
    sh = !sh;
    rd = false;
    if( tkn )
    {
```

```
        if( sh ) tmp = "11";
        else  tmp = "10";
    }

    else
    {
        if( sh ) tmp = "01";
        else   tmp = "00";
    }

    byte[] sendData2 = new byte[10];
    sendData2 = (Integer.toString( serverID )+" release").getBytes();
    DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
    //tell TokenTraversalMonitor its privilege
    try{talkTokenTraverseSocket.send(outPacket2);  }
    catch(IOException e ){}
    grant = false;
}

//third situation case 2:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && ( (sh&&
    serverID!=0)||(!sh&&rd) ) && pollServerInCharge.request == true)
{
    if( ignore == false )  //should send msg
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();

        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
    }
    ignore = true;   //set it to true, so not to send multiple times


    grant = true;

    if( reportCS == false )
    {
        byte[]sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" cs").getBytes();

        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                sendData2.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket2);  }
        catch(IOException e ){}
    }
    reportCS = true; //reported already
}

//fourth situation case 1:
else if( (!Tvalue( serverID, tkn, pTkn ) || serverID== 0) && Svalue(serverID, sh,
    pSh) && sh && pollServerInCharge.request == false)
{
    if( ignore == false )
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )+" get").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,
                                sendData.length, measureAddress, FinalVariables.
                                  MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket);  }
        catch(IOException e ){}
```

```
        }
        ignore = false;   //set ignore to false because for case 2
        reportCS = false; //set for case 2

        sh = !sh; //sh should be low now
        rd = false;
        tmp = "30";

        byte[] sendData2 = new byte[10];
        sendData2 = (Integer.toString( serverID )+" release").getBytes();
        DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.
                                      MeasurePort);
        //tell TokenTraversalMonitor its privilege
        try{talkTokenTraverseSocket.send(outPacket2);  }
        catch(IOException e ){}
        grant = false;
    }

    //fourth situation case 2:
    else if( (!Tvalue( serverID, tkn, pTkn ) || serverID== 0) && Svalue(serverID, sh,
        pSh) && sh && pollServerInCharge.request == true)
    {
        if( ignore == false )  //should send msg
        {
            byte[]sendData = new byte[10];
            sendData = (Integer.toString( serverID )+" get").getBytes();

            DatagramPacket outPacket = new DatagramPacket(sendData,
                                    sendData.length, measureAddress, FinalVariables.
                                      MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket);  }
            catch(IOException e ){}
        }
        ignore = true;   //set it to true, so not to send multiple times


        grant = true;
        if( reportCS == false )
        {
            byte[]sendData2 = new byte[10];
            sendData2 = (Integer.toString( serverID )+" cs").getBytes();

            DatagramPacket outPacket2 = new DatagramPacket(sendData2,
                                    sendData2.length, measureAddress, FinalVariables.
                                      MeasurePort);
            //tell TokenTraversalMonitor its privilege
            try{talkTokenTraverseSocket.send(outPacket2);  }
            catch(IOException e ){}
        }
        reportCS = true; //reported already
    }
    return tmp;
}



//return a string with at most 2 elements, 1st is the token, 2nd is shepherd,
//"1" means high, "0" means low, and 3 means no change
private String processTokenAndShepherdNoRequest()
{
    String tmp = "33";
    int temp = action;
    //first situation:
    if( Tvalue( serverID, tkn, pTkn ) && !Svalue( serverID, sh, pSh ) && ( sh || serverID
        !=0) )
    {
        byte[]sendData = new byte[10];
        sendData = (Integer.toString( serverID )).getBytes();
```

```
    DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
        timeMonitorAddress, FinalVariables.PrivilegePort);

    try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
    catch(IOException e ){}

    tkn = !tkn;  //only pass token
    rd = false;
    tmp = boolToInt(tkn)+""+boolToInt(sh);
}


//second situation:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && !sh && !rd )
{
    byte[]sendData = new byte[10];
    sendData = (Integer.toString( serverID )).getBytes();
    DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
        timeMonitorAddress, FinalVariables.PrivilegePort);

    try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
    catch(IOException e ){}

    tkn = !tkn;  //only pass token
    rd = true;
    tmp = boolToInt(tkn)+""+boolToInt(sh);
}


//third situation:
else if( Tvalue( serverID, tkn, pTkn ) && Svalue( serverID, sh, pSh ) && ( (sh&&
    serverID!=0)||(!sh&&rd) ) )
{
    byte[]sendData = new byte[10];
    sendData = (Integer.toString( serverID )).getBytes();
    DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
        timeMonitorAddress, FinalVariables.PrivilegePort);

    try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
    catch(IOException e ){}

    tkn = !tkn;  //pass token
    sh = !sh;
    rd = false;
    tmp = boolToInt(tkn)+""+boolToInt(sh);
}


//fourth situation:
else if( (!Tvalue( serverID, tkn, pTkn ) || serverID== 0) && Svalue(serverID, sh,
    pSh) && sh  )
{
    byte[]sendData = new byte[10];
    sendData = (Integer.toString( serverID )).getBytes();
    DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
        timeMonitorAddress, FinalVariables.PrivilegePort);

    try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
    catch(IOException e ){}

    sh = !sh; //sh should be low now
    rd = false;
    tmp = boolToInt(tkn)+""+boolToInt(sh);
}

if(temp!=action)
{
    byte[]sendData = new byte[10];
    sendData = ("a").getBytes();
    DatagramPacket outPacket = new DatagramPacket(sendData,sendData.length,
        timeMonitorAddress, FinalVariables.PrivilegePort);
```

```
        try{talkToTimeSocket.send(outPacket);  }//tell timeMonitor its privilege
             catch(IOException e ){}
    }
    return tmp;
}


public void run()
{
    RingInitialization init = new RingInitialization( ringMakerAddress, listenSocket,
        talkToRMSocket, serverID );

    ProcessInfo p = (ProcessInfo) init.findNeighbor();
    neighborAddress = p.address;
    neighborPort = p.lport;

    int flag = 0;  //used only when test stabilization time
    for( ;; )  ////this for loop testing stabilization time
    {
        try
        {
            byte[] sendData = new byte[FinalVariables.PKSIZE];
            sendData = ( processTokenAndShepherdNoRequest() ).getBytes();
            countMsg++;


            DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
                neighborAddress, neighborPort );

            sendSocket.send( outPacket );

            byte[] recvData = new byte[FinalVariables.PKSIZE];
            DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

            listenSocket.receive( inPacket );

            String tmp = new String(inPacket.getData()).trim();
            if( tmp.equals("stop measure stabilization") )
            {
                System.out.println("number of messages sent "+ countMsg );
                byte[]actionData = new byte[1000];
                actionData = (Integer.toString( action )).getBytes();
                DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                    length, timeMonitorAddress, FinalVariables.ActionPort);

                try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
                catch(IOException e ){}
                break;
            }

            if( flag == 1 )  //waiting for restart
            {
                if( tmp.equals("restart") )  //initialize
                {
                    flag = 0;
                    tkn = false;
                    sh = false;
                    pTkn = false;   //precessor tkn
                    pSh = false;
                }
            }
            else if( flag == 0 )   //must deal with the msg
            {
                if( tmp.equals( "stop" ) )
                    flag = 1;

                else
                {
                    //extract the value of preseccor's token
                    if( tmp.charAt(0) == '1' )
```

```
                    pTkn = true;
                else if ( tmp.charAt(0) == '0' )
                    pTkn = false;

                //extract the value of preseccor's companion
                if( tmp.charAt(1) == '1' )
                    pSh = true;
                else if( tmp.charAt(1) == '0' )
                    pSh = false;
            }
        }

    } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
        printStackTrace(); }
    yield();
}  //end of for loop


byte[] flagData = new byte[FinalVariables.PKSIZE];
flagData = "now".getBytes();

DatagramPacket nowPacket = new DatagramPacket( flagData, flagData.length,
    neighborAddress, neighborPort );
try
{
    sendSocket.send( nowPacket );
    for(; ; )
    {
        byte[] rfData = new byte[FinalVariables.PKSIZE];
        DatagramPacket rfPacket = new DatagramPacket( rfData, rfData.length);
        listenSocket.receive( rfPacket );
        String rtmp = new String(rfPacket.getData()).trim();
        if(rtmp.equals("now"))
            break;
    }
}catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
    printStackTrace(); }
tkn = false;
rd = false;
sh = false;
pTkn = false;
pSh = false;

action=0;
countMsg = 0;
for( ;; )  //this for loop testing token traversal time
{
    try
    {
        byte[] sendData = new byte[FinalVariables.PKSIZE];

        while( grant && pollServerInCharge.request )
            yield();

        //token traversal time measurement
        sendData = (processTokenAndShepherdForTokenTraversal()).getBytes();

        DatagramPacket outPacket = new DatagramPacket( sendData, sendData.length,
            neighborAddress, neighborPort );

        sendSocket.send( outPacket );
        countMsg++;

        byte[] recvData = new byte[FinalVariables.PKSIZE];
        DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);

        listenSocket.receive( inPacket );

        String tmp = new String(inPacket.getData()).trim();

        if( tmp.equals("stop") )
```

```
            {
                byte[]actionData = new byte[1000];
                actionData = (Integer.toString( action )).getBytes();
                DatagramPacket actionPacket = new DatagramPacket(actionData,actionData.
                    length, measureAddress, FinalVariables.TraverseActionPort);

                try{actionSocket.send(actionPacket);  }//tell timeMonitor its privilege
                catch(IOException e ){}
                break;
            }

            else
            {
                //extract the value of preseccor's token
                if( tmp.charAt(0) == '1' )
                    pTkn = true;
                else if ( tmp.charAt(0) == '0' )
                    pTkn = false;

                //extract the value of preseccor's companion
                if( tmp.charAt(1) == '1' )
                    pSh = true;
                else if( tmp.charAt(1) == '0' )
                    pSh = false;
            }
        } catch ( IOException e ) {System.out.println("IO: " + e.getMessage()); e.
            printStackTrace(); }
        yield();
    }
 }  //end of method
//end of class
```

APPENDIX 12 TASK DISTRIBUTION

```
mport java.io.*;
mport java.util.*;

**
 This class is used to generate a two dimensional integer array
 @version v0.0.0
 @author Rong Wang
/

ublic class TaskDistribution

   public static void main( String[] args )
   {
       try
       {
           Random rand = new Random();

           int n = FinalVariables.MaxTaskNumber * FinalVariables.MaxProcessNumber;
           int [] taskDistribution = new int[n];
           int size = (int ) (FinalVariables.ActivePercentage * n); //the number of elements↙
               to be assigned 1

           int maxResponseTime =120000;

           for( int a = 0; a < n; a++ ) //initialize to 0
           {
               //random int ranging from 0 to maxResponseTime - 1
               Integer anInt1 = new Integer( rand.nextInt( maxResponseTime ));
               taskDistribution[a] = anInt1.intValue();
           }

           //write to file
           PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter            ↙
               ("taskDistribution.txt")));
           for( int i = 0; i < FinalVariables.MaxTaskNumber; i++ )
           {
               String aLine = "";
               for( int j = 0; j < FinalVariables.MaxProcessNumber; j++ )
               {
                   aLine = aLine + String.valueOf(taskDistribution[i*FinalVariables.       ↙
                       MaxProcessNumber+j]) + " ";
               }
               out.println( aLine );
               }
           out.close();
       } catch( IOException e ) { System.out.println("IOEXCEPTION: " + e.getMessage());}
   } //end of main
 //end of class
```

APPENDIX 13 TIME MONITOR

```
mport java.net.*;
mport java.io.*;
mport java.lang.*;
mport java.util.*;


**
 This class is used to measure stabilization time
 @version v0.0.0
 @author Rong Wang
/

lass TimeMonitor

   private static ProcessInfo [] processes;
   /**
   * a constructor
   */
   TimeMonitor(){

       processes = new ProcessInfo[FinalVariables.ServerNumber];
       for( int i = 0; i < FinalVariables.ServerNumber; i++ )
           processes[i] = new ProcessInfo();
   }
   /**
   * self explanatory
   * @param p a process to be added
   */
   public void addProcess( ProcessInfo p )
   {
       processes[p.serverID] = p;  //add to array according to its ID number
   }

   public static void main( String[] args )
   {
       TimeMonitor tm = new TimeMonitor();
       int firstSender = -1, nextExpected = -1;
       long deliveryStartTime = 0, deliveryEndTime = 0, sTime = 0, total=0;

       DatagramSocket timeSocket = null, previlegeSocket = null, infoSocket = null,
           actionSocket=null;
       try{
           timeSocket = new DatagramSocket( FinalVariables.TimePort );
           previlegeSocket = new DatagramSocket( FinalVariables.PrevilegePort );
           infoSocket = new DatagramSocket( FinalVariables.InfoPort );
           actionSocket = new DatagramSocket( FinalVariables.ActionPort );
       } catch (SocketException s ){}


       //first try to receive process address info. from RingMaker
       int count = 0;
       while( count != FinalVariables.ServerNumber )
       {
           tm.addProcess( (ProcessInfo) TransportTool.receiveFrom( infoSocket ) );
           count++;
           System.out.println("counts " + count );
       }

       //then try to receive info. from RingMaker to know the start time of ring
       byte[] recvData = new byte[FinalVariables.PKSIZE];
       DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);
       try{timeSocket.receive( inPacket );}catch(IOException e ){}

       //retreive the time information from the received msg
       Long ringSetUpTime = new Long( new String(inPacket.getData() ).trim() );
       long ringSetupTime = ringSetUpTime.longValue();
       int counter =0, sAct=0, totalAct=0;
       for( int a = 0; a < FinalVariables.MaxRunForStabilization; a++ )
```

```
{
    for( ;; )
    {
        recvData = new byte[FinalVariables.PKSIZE];
        inPacket = new DatagramPacket( recvData, recvData.length);
        //get the sender's info
        try{previlegeSocket.receive( inPacket );}catch(IOException e ){}
        String tmp = new String(inPacket.getData()).trim();
        if( tmp.equals("a") )
            totalAct++;

        else
        {
            System.out.println("Privileged process " + tmp );
            int intNum = Integer.parseInt(tmp);
            if( firstSender == -1 )
            {
                firstSender = intNum;
                if( firstSender == FinalVariables.ServerNumber -1 )
                    nextExpected = 0;
                else nextExpected = firstSender + 1;
                deliveryStartTime = System.currentTimeMillis();
                sAct = totalAct;
            }
            else if( firstSender == intNum && nextExpected == firstSender ) //a cycle
                finished
            {
                deliveryEndTime = System.currentTimeMillis();
                break;
            }
            else
            {
                if( nextExpected == intNum )   //legal order
                {
                    if( nextExpected == FinalVariables.ServerNumber - 1)
                        nextExpected = 0;
                    else nextExpected++;
                }
                else    //illegal order, record the sender as firstSender
                {
                    firstSender = intNum;
                    if( firstSender == FinalVariables.ServerNumber - 1 )
                        nextExpected = 0;
                    else nextExpected = firstSender + 1;
                    //this should be the stabilization time
                    deliveryStartTime = System.currentTimeMillis();
                    sAct = totalAct;
                }
            }
        }
    } //end of inner for loop

    //for next circle to use
    firstSender = -1;
    nextExpected = -1;
    sTime = deliveryStartTime - ringSetupTime;
    total += sTime;

    try
    { //to write the output to file
        File f = new File("performance.log"); // delete the file if it already exists
        if (f.exists())
            f.delete();
        PrintWriter out = new PrintWriter(new FileWriter(f));

        out.println("Stabilization time:  "+ sTime + " milliseconds" );
        out.close();  // We're done writing
    } catch (IOException e) { /* Handle exceptions */ }

    System.out.println( "Stabilization time:  "+ sTime );
```

```
            try{Thread.sleep( 1 );}  //set an upper bound for each process to hold privilege
            catch ( InterruptedException e ) { System.out.println("Interrupted: " + e.
                getMessage());}

            //after stabilization, send msg to token listener to restart
            for ( int i = FinalVariables.ServerNumber-1; i >= 0; i-- )
            {
                try{
                    byte[] sendData = new byte[100];
                    sendData = ("stop").getBytes();
                    DatagramPacket outPacket = new DatagramPacket(sendData,
                          sendData.length, tm.processes[i].address, tm.processes[i].lport);
                    timeSocket.send(outPacket);
                }catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
            }  //end of for loop

            for ( int j = FinalVariables.ServerNumber-1; j >= 0; j-- )
            {
                try{
                    byte[] sendData = new byte[100];
                    sendData = ("restart").getBytes();
                    DatagramPacket outPacket = new DatagramPacket(sendData,
                          sendData.length, tm.processes[j].address, tm.processes[j].lport);
                    timeSocket.send(outPacket);
                }catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
            }  //end of for loop

            ringSetupTime = System.currentTimeMillis();  //begin to calculate the next run
        }//end of for loop of MaxRunForStabilization

        //after testing stabilization, send msg to token listener to stop test the
            stabilization time
        for ( int i = FinalVariables.ServerNumber-1; i >= 0; i-- )
        {
            try
            {
                byte[] sendData = new byte[1000];
                sendData = ("stop measure stabilization").getBytes();
                DatagramPacket outPacket = new DatagramPacket(sendData,
                          sendData.length, tm.processes[i].address, tm.processes[i].lport);
                timeSocket.send(outPacket);
            }catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
        }
        int totalAction=0;
        for( int i=0; i < FinalVariables.ServerNumber; i++)
        {

            byte[] actionData = new byte[FinalVariables.PKSIZE];
            DatagramPacket actionPacket = new DatagramPacket( actionData, actionData.length);
            try{actionSocket.receive( actionPacket );}catch(IOException e ){}
            String t = new String(actionPacket.getData()).trim();
            int actNum = Integer.parseInt(t);
            totalAction += actNum;
        }
        System.out.println("total action number is "+totalAction );
    } //end of main
//end of class
```

```
mport java.net.*;
mport java.io.*;
mport java.lang.*;
mport java.util.*;

**
 This class is used to store the information used to measure token traversal time for every
 process in token ring,
 @version v0.0.0
 @author Rong Wang
/
lass MeasureInfo

    public boolean request;
    public int csExecuted;
    public long timeForACircle;

    /**
    * a constructor without any parameter
    */
    MeasureInfo()
    {
        request = false;
        csExecuted = 0;
        timeForACircle = 0;
    }


**
 This class is used to measure token traversal time
 @version v0.0.0
 @author Rong Wang
/
ublic class TokenTraverseMonitor

    private static final int MeasureSize = 4;
    private MeasureInfo [][] measureArray;
    private int[] requestIndex;
    private ProcessInfo [] processes;

    /**
    * a constructor
    */
    TokenTraverseMonitor()
    {
        requestIndex = new int[FinalVariables.MaxProcessNumber];
        for ( int a = 0; a < FinalVariables.MaxProcessNumber; a++ )
            requestIndex[a] = -1;

        measureArray = new MeasureInfo[MeasureSize][FinalVariables.ServerNumber];
        for( int i = 0; i < MeasureSize; i++ )
            for ( int j = 0; j < FinalVariables.ServerNumber; j++ )
                measureArray[i][j] = new MeasureInfo();

        processes = new ProcessInfo[FinalVariables.ServerNumber];
        for( int i = 0; i < FinalVariables.ServerNumber; i++ )
            processes[i] = new ProcessInfo();
    }

    /**
    * self explanatory
    * @param p a process to be added
    */
    public void addProcess( ProcessInfo p )
    {
        processes[p.serverID] = p;  //add to array according to its ID number
    }
```

```
/**
* self explanatory
* @param args argument to be passed
*/
public static void main( String[] args )
{
    TokenTraverseMonitor tm = new TokenTraverseMonitor();
    int count =  MeasureSize * FinalVariables.MaxProcessNumber;
    boolean firstCS=true;
    long csStart=0;
    int countCS = 0;

    DatagramSocket measureSocket = null, actionSocket=null, proInfoSocket=null,
        sendSocket=null;
    try{
        actionSocket = new DatagramSocket( FinalVariables.TraverseActionPort );
        proInfoSocket = new DatagramSocket( FinalVariables.InfoPort2 );
        measureSocket = new DatagramSocket( FinalVariables.MeasurePort );
        sendSocket = new DatagramSocket();
    } catch (SocketException s ){}


    //first try to receive process address info. from RingMaker
    int countProcess = 0;
    while( countProcess != FinalVariables.ServerNumber )
    {
        tm.addProcess( (ProcessInfo) TransportTool.receiveFrom( proInfoSocket ) );
        countProcess++;
    }

    //try to receive info. from privileged processes
    for( ;; )
    {

        /*used when meausure average token traversal time with no request from
            application
        if ( count == 0 )   //quit after counting
        {
            break;
        }*/

        byte[] recvData = new byte[FinalVariables.PKSIZE];
        DatagramPacket inPacket = new DatagramPacket( recvData, recvData.length);
        try{measureSocket.receive( inPacket );}catch(IOException e ){} //get the sender's
            info
        String tmp = new String(inPacket.getData()).trim();
        int index = tmp.indexOf(' ');
        int pid = Integer.parseInt(tmp.substring( 0, index ));

        String msg = tmp.substring( index + 1 );
        if ( msg.equals( "cs" ) ) //critical section
        {
            System.out.print(pid +"     ");
            if( firstCS)
            {
                csStart=System.currentTimeMillis();
                firstCS=false;
            }

            countCS++;
            if( countCS == 100 )
            {
                System.out.println("Total time used for 100 tasks is:     "+(System.
                    currentTimeMillis()-csStart));
                break;
            }

            //record for each process who wishes to know how many cs r executed between
                the time it releases privilege to the time it gets privilege again
            if( count!=0)
```

```
        {
            for( int i = 0; i < FinalVariables.ServerNumber; i++ )
            {
                if( tm.requestIndex[i] >= 0 && tm.requestIndex[i] < MeasureSize )
                   if( tm.measureArray[tm.requestIndex[i]][i].request == true )
                      if( pid != i )  //from release to get, so not add to itself
                         tm.measureArray[tm.requestIndex[i]][i].csExecuted++;
            }
        }
    }

    else if( count != 0 )
    {
        //notify the end of record for a process, so  other processes can ignore it
        if ( msg.equals( "get" ) )
        {
            if(tm.requestIndex[pid] >= MeasureSize)
                continue;
            else if( tm.requestIndex[pid] != -1 && tm.measureArray[tm.requestIndex
                [pid]][pid].request == true )
            {
                tm.measureArray[tm.requestIndex[pid]][pid].timeForACircle = System.
                    currentTimeMillis() - tm.measureArray[tm.requestIndex[pid]][pid].
                    timeForACircle;

                tm.measureArray[tm.requestIndex[pid]][pid].request = false;
                count--;
            }
        }
        else //if ( msg.equals( "release" ) )
        {
                tm.requestIndex[pid]++; //how many times have been recorded
                if( tm.requestIndex[pid] >= MeasureSize )
                    continue;
                else
                {
                    if( tm.measureArray[tm.requestIndex[pid]][pid].request == false )
                    {
                        tm.measureArray[tm.requestIndex[pid]][pid].timeForACircle =
                            System.currentTimeMillis();
                        tm.measureArray[tm.requestIndex[pid]][pid].request = true;
                    }
                }
        } //end of else
    }
}//end of for

/* used when count action number
//send each process to ask action number
for ( int i = FinalVariables.ServerNumber-1; i >= 0; i-- )
{
    try{
        byte[] sendData = new byte[100];
        sendData = ("stop").getBytes();
        DatagramPacket outPacket = new DatagramPacket(sendData,
               sendData.length, tm.processes[i].address, tm.processes[i].lport);
        sendSocket.send(outPacket);
    }catch ( IOException e ) { System.out.println("IO: " + e.getMessage());}
}  //end of for loop

//receive action number used for token delivery from each process
int totalAction=0;
for( int i=0; i < FinalVariables.ServerNumber; i++)
{
    byte[] actionData = new byte[FinalVariables.PKSIZE];
    DatagramPacket actionPacket = new DatagramPacket( actionData, actionData.length);
    try{actionSocket.receive( actionPacket );}catch(IOException e ){}
    String t = new String(actionPacket.getData()).trim();
    int actNum = Integer.parseInt(t);
    totalAction += actNum;
}
```

```
      System.out.println("total action number is "+totalAction );
      */

      double a =0.0, total = 0.0, total2 = 0.0;
      for( int i = 0; i < FinalVariables.ServerNumber; i++ )
      {
          for( int j=0;j<MeasureSize;j++)
          {
              a =(double) (tm.measureArray[j][i].timeForACircle -tm.measureArray[j][i].
                  csExecuted*0) / (FinalVariables.ServerNumber-1);
              total += a;

              total2 += tm.measureArray[j][i].timeForACircle;
          }
      }
      System.out.println("\nAverage token traverse time is "+total2/(MeasureSize*
          FinalVariables.MaxProcessNumber) );
  } //end of main
//end of class
```

```
mport java.io.*;
mport java.net.*;

ublic class TransportTool {

 /**
  * to receive an object of ProcessInfo through the network
  * @param socket through which socket the object will be received
  */
 public static Object receiveFrom(DatagramSocket aSocket) {
   Object o = null;

  try {
     byte[] recvData = new byte[5000];
     DatagramPacket inPacket = new DatagramPacket(recvData, recvData.length);

     aSocket.receive(inPacket);

     ByteArrayInputStream byteInStream = new ByteArrayInputStream(recvData);
     ObjectInputStream ois = new ObjectInputStream(new BufferedInputStream(
         byteInStream));

     o = ois.readObject();

     ois.close();

   }
   catch (ClassNotFoundException e) {
     System.out.println("ClassNotFound: " + e.getMessage());
   }
   catch (IOException e) {
     System.out.println("IO: " + e.getMessage());
     e.printStackTrace();
   }

   return o;
 }

 /**
  * to send an object of ProcessInfo through the network
  * @param p an object to be sent
  * @param addr to which address the object will be sent
  * @param aPort to which port the object will be sent
  * @param socket through which socket the object will be sent
  */
 public static void sendTo(Object p, InetAddress addr, int aPort,
                           DatagramSocket socket) {
   try {
     ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream(5000);
     ObjectOutputStream oos = new ObjectOutputStream(new BufferedOutputStream(
         byteOutStream));

     oos.flush();
     oos.writeObject(p);
     oos.flush();

     byte[] sendData = byteOutStream.toByteArray();
     DatagramPacket outPacket = new DatagramPacket(sendData, sendData.length,
         addr, aPort);

     socket.send(outPacket);
     oos.close();
   }
   catch (IOException e) {
     System.out.println("IO: " + e.getMessage());
     e.printStackTrace();
   }
 }
```

VITA

Rong Wang was born in Song Jiang He, Ji Lin, P. R. China, on January 26, 1971, the daughter of Xicui Wang and Zhixue, Wang. After completing her work at Song Lin High School, Song Jiang He, Ji Lin, P. R. China, in 1989, she entered Peking University in Beijing, P. R. China. She received the Bachelor of Art from Peking University in July 1994. During the following years she was employed as a Japanese lecturer with Beijing Machinery Institute in Beijing, P. R. China. In January 2001, she entered the Graduate School of Texas State University, San Marcos, Texas.

Permanent Address: 4-2 Song Jiang Street
Song Jiang He, Ji Lin 134504
P. R. China

This thesis was typed by Rong Wang