Title
Wireless Token Ring Protocol

Author
Lee, Duke

Publication Date
2001-04-01

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Wireless Token Ring Protocol

Duke Lee

California PATH Research Report
UCB-ITS-PRR-2001-10

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

Final Report for MOU 334

April 2001

ISSN 1055-1425

Wireless Token Ring Protocol [1]

Duke Lee

March 23, 2001

[1]This is based on work jointly done with Roberto Attias, Anuj Puri, Raja Sengupta, and Stavros Tripakis

Abstract

The Wireless Token Ring Protocol (WTRP) is a medium access control protocol for wireless networks in Intelligent Transportation Systems. It supports quality of service in terms of bounded latency and reserved bandwidth. WTRP is efficient in the sense that it reduces the number of retransmissions due to collisions. It is fair in the sense that each station takes a turn to transmit and is forced to give up the right to transmit after transmitting for a specified amount of time. It is a distributed protocol that supports many topologies since not all stations need to be connected to each other or to a central station. It can be used with an admission control agent for bandwidth or latency reservations. WTRP is robust against single node failure. WTRP is designed to recover gracefully from multiple simultaneous faults. It has applications to inter-access point coordination in ITS DSRC, and safety-critical vehicle-to-vehicle networking.

Contents

Chapter 1

Introduction

One of the challenges in a communication network is to guarantee quality of service (often abbreviated as QOS) with high probability. One obvious way to solve the problem is to increase the network capacity. By increasing network capacity, one increases the chance that network resources are available on demand. However, this is not always a practical solution, since increasing the network capacity costs money.

QOS has been tackled in many ways. In wire-line networks, efforts have focused on network layer queuing and routing techniques. In the IP Differentiation of Services[3], routers give higher priority, in terms of transmission latency, to time-critical data such as voice and the video transmission, over non-time-critical data such as email transmissions. Since typical email users are less sensitive to latencies, it is possible to provide the email users reasonable satisfaction while giving priority to the voice user. Another approach is per flow admission control [4]. The idea behind this is to reserve bandwidth for the data flow before making the connection. The routers along the connection path need to maintain per flow information about the reserved traffic. The routers control the queuing policy to ensure that each flow traffic receives the promised network resources.

In an unreliable medium such as wireless, problem of providing quality of service (QOS) at the network layer using queuing and routing techniques is not sufficient. QOS must also be addressed at the data-link layer. The IEEE 802.11[5] in PCF (Point Coordination Function) mode, the HiperLAN[6], and Bluetooth[7] achieve bounded latency by having a central station poll the slave stations. Most academic research has focused on this centralized approach [9] [8]. The centralized approach is suitable for networks where only the last hop is wireless. In the centralized approach, the network is managed centrally from a central station. This is a limitation in wireless networks with ad-hoc topologies.

The Wireless Token Ring Protocol (WTRP) discussed in this paper is a distributed medium access control protocol for ad-hoc networks. The advantages of a distributed medium access control protocol are its robustness against single node failure, and its support for flexible topologies, in which nodes can be partially connected and not all nodes need to have a connection with a master.

As in the IEEE 802.4[2] standards, WTRP is designed to recover from multiple simultaneous failures. One of the biggest challenges that the WTRP overcomes is partial connectivity. To overcome the problem of partial connectivity, management, special tokens, additional fields in the tokens, and new timers are added to the protocol. When a node joins a ring, it is required that the joining node be connected to the prospective predecessor and the successor. The joining node obtains this information by looking up its connectivity table. When a node leaves a ring, the predecessor of the leaving node finds the next available node to close the ring by looking up its connectivity table. Partial connectivity also affects the multiple token resolution protocol (deleting all multiple tokens but one). In a partially connected network, simply dropping the token whenever a station hears another transmission is not sufficient. To delete tokens that a station is unable to hear, we have designed a unique priority assignment scheme for tokens. Stations only accept a token that has greater prioirity than the token the station last accepted. The WTRP also has algorithms for keeping each ring address unique, to enable the operation of multiple rings in proximity.

1.1 Related Works

Wireless Token Ring is a wireless LAN protocol inspired by the IEEE 802.4[2] Token Bus Protocol. The protocol guarantees bounded delay and a share of bandwidth to all stations in the network. A version of token ring protocol has been implemented by the author and used for the automated highway project[11] in 1998. There has also been work on the token ring protocol by Chao Chen[21]. To date, none of the work has seriously considered the problems of partial connectivity, fast and graceful recovery time, multiple rings, and admission control. This report documents a design that solves these problems.

1.2 Application

The Wireless Token Ring has been conceived initially for the UC Berkeley PATH Automated Highway Project[11], and the Berkeley Aerobot project[12]. These two projects impose a stringent requirements on the medium access protocol in terms of bandwidth, latency, and speed and grace of failure recovery. The platoon mode of the automated highway project involves up to 20 nodes in each platoon, and requires that information (approximately 100 bytes for vehicular speed, acceleration, and coordination maneuvers) be transmitted every 20ms. The failure recovery time for the communication system must be within 40ms.

However the usefulness of the protocol is not limited to these target applications. As stated in [13], the wireless Ad-Hoc network has many applications:

1. military (tactical communication) - rapid establishment of a communication infrastructure during deployment of forces in a foreign/hostile terrain
2. rescue missions - for communications in areas without adequate wireless coverage
3. national security - for communication during times of national crisis, where the existing communication infrastructure is non-operational due to a natural disasters or a global war
4. law enforcement - similar to tactical communication
5. commercial use - for set up of communication in exhibitions, conferences, or sale presentations
6. education - for operation of virtual classrooms
7. sensor networks - for communication between intelligent sensors

1.3 Wireless Environment

Inspired by the IEEE 802.4 [2] standards, the WTRP builds a logical ring that defines a transmission sequence among the nodes in the ring. It also provides a distributed way to add and delete stations from the ring. Additional challenges encountered in the wireless environment are:

1. Stations may not be fully connected. (i.e. not all nodes in the ring are directly connected)
2. Radio range can be asymmetrical.

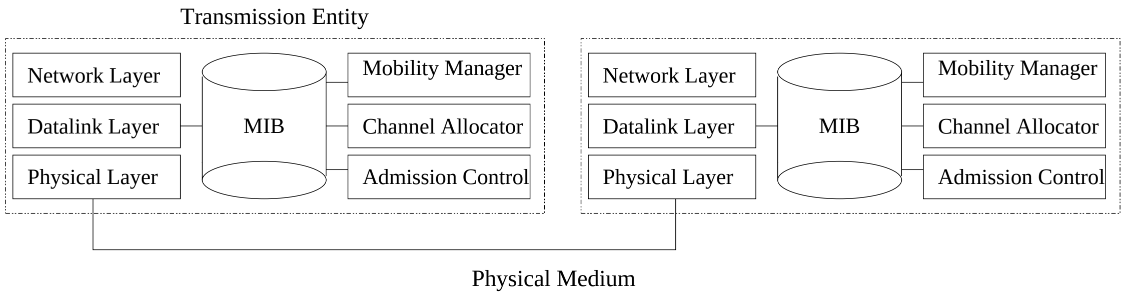

Figure 1.1: System Architecture

3. Multiple rings can exists in proximity.

1.4 Overall System Architecture

To put WTRP into a context in terms its placement in the communication system, we describe the overall system architecture in Figure 1.1. In addition to the communication stack including the Datalink Layer where WTRP will be located, we need Mobility Manager, Channel Allocator, Management Information Base (MIB), and Admission Control Manager. We assume that multiple channels are available, and that interfering rings are on different channels. Interfering rings are assigned to different channels by a channel allocator (Section 1.4.2).

The development of Mobility Here we outline the functions of each module, and discuss the context in which these modules are designed.

1.4.1 Medium Access Control

Medium Access Control (MAC) enables multiple nodes to transmit on the same medium. This is where WTRP would be located. The main function of MAC is to control the timing of the transmissions to increase the chances of successful transmission.

In our architecture, the MAC layer manages the ring and the timing of the transmissions. The ring management involves the following:

1. Ensuring that each ring has an unique ring address.
2. Ensuring that one and only one token exists in a ring.
3. Ensuring that the rings are proper.
4. Managing the joining and the leaving operations.

We will describe the operations of the MAC layer in Section 2 and Section 3.

1.4.2 Channel Allocator

In a general sense, the channel allocator chooses the channel on which the station should transmit. If a large number of token rings exist in proximity, their efficiency can be increased by achieving spatial reuse through sensible channel allocation. The idea of spatial reuse is one of the core ideas of the wireless cellular community. The same channel (or a set of channels) can be reused in region A and B, if the two regions are separated by sufficient distance measured in terms of the signal to interference ratio. One way to increase spatial reuse is to reduce the cell size. Reducing the cell size (thus reducing the transmission power) has the following benefits:

1. Increase in capacity
2. Increase in battery life
3. Decrease in equipment costs

In addition, dividing the nodes into multiple rings would reduce the number of nodes in a ring. This would decrease the token rotation time which results in decreased maximum medium access time.

Finding the globally optimal[1] solution is a challenging problem in any large deployment of many mobile nodes. First, collecting and maintaining channel allocation information can be difficult and burdensome. This is because the collection and maintenance of information may involve frequent packet transmissions. Second, the optimal allocation computation is complex. The complexity of the problem is greater than that of allocating channels to already divided regions, allocating with the restriction that no adjacent regions can have the same channel. This problem of allocating each region a channel, using the minimum number of channels, is an NP-hard problem.

The problem of finding an optimal channel allocation is further complicated by the following factors in the wireless Ad-Hoc environment.

1. The transmission ranges of the radios are limited.

2. No stationary base station exists.

3. The boundary of each channel is fluid.

Moreover, in our applications, the network capacity must be maintained without violating the latency and the bandwidth requirements of each node.

A much more scalable solution could be a distributed one that uses a greedy algorithm. And this is the method that is being studied for our design. In our implementation, the channel allocator is local to each station, and the channel allocator can access the network topology information through the MIB. Each node decides on which channel to join in a

[1]By globally optimal channel allocation we mean an allocation that maximizes the capacity of the network

distributed manner using the information collected. What information is collected and how it is used still being investigated.

Mobility Manager

The Mobility Manager decides when a station should join or leave the ring. The problem that the Mobility Manager has to solve is similar to the mobile hand-off problem. When a mobile node is drifting away from a ring and into the vicinity of another ring, at some threshold the Mobility Manager decides to move to the next ring. The level of connection of a node to a ring can be found from the connectivity table described in Section 2.

Admission Control

The Admission Control Manager limits the number of stations that can transmit on the medium. This is to ensure that a level of quality of service in terms of bounded latency and reserved bandwidth is maintained for stations already granted permission to transmit on the medium. There is an Admission Control Manager in each ring. The Admission Control Manager may move with the token but does not have to move every time the token moves. The Admission Control Manager periodically solicits other stations to join if there are "resources" available in the ring. The "resource" of the token ring can be defined in the following way. The MAX_MTRT is the minimum of the maximum latency that each station in the ring can tolerate. RESV_MTRT is the sum of token holding time (THT) of each station. The Admission Control Manager has to ensure the inequality: $RESV\_MTRT < MAX\_MTRT$. This means that the resource left in the ring is $MAX\_MTRT - RESV\_MTRT$. Only if there are enough resources left, may the Admission Control Manager solicit another station to join. During the solicitation, the Admission

Control Manager also advertises the available resources. Only stations that require less resource than that available in the ring may join.

Policer

The policer monitors the traffic generated by the application. It throttles the application when more traffic than reserved is produced. In the WTRP, because the token holding timer polices the traffic generated by a station, no special policer module is necessary.

Management Information Base (MIB)

The Management Information Base holds all the information that each management module needs to manage the MAC module. Majority of this information is collected by the MAC module and stored there. However, some of the information may need to be communicated. This is gathered and refreshed by the SNMP agent. Details on this are still being investigated.

Chapter 2

The Description of the Protocol

2.1 Definitions

1. WTRP refers to Wireless Token Ring Protocol, the topic of this report.

2. The term "frame" refers to what we pass to the physical layer interface. A "frame" does not include the preambles, the start delimiter, the CRC check, and the end delimiter.

3. The terms "station" and "node" are used interchangeably to describe the communication entities on the shared medium.

4. The predecessor and the successor of station X describe the station that X receives the token from and the station that the X passes the token to respectively.

5. "Incorrect state" means that a node's view of the topology is wrong. For example node X may believe that node Y is its predecessor, but node Y does not.

6. "Stable environment" refers to a state in which the topology of the network is fixed and there are no transmission errors.

7. “Proper ring” refers to a ring where the next station and previous station fields of a node are correct. It is more precisely defined in Section 5.

8. Capacity of the network refers to the total bandwidth.

9. The Channel Allocator, Mobility Manager, and Admission Control Manager introduced in Section 1.4 are referred to as “management modules”.

10. THT refers to the Token Holding Time, i.e., the amount of time that a station can hold the token for transmission of data.

2.2 Observations

1. Not all stations need to be involved in token passing. Only those stations which desired to initiate the data transmission need to be involved.

2. Any station may detect multiple tokens and lost tokens. There are no special “monitor” station required to perform token recovery functions.

3. Due to errors, stations may not have a consistent view of the ring.

2.3 Protocol Overview

In the WTRP, the successor and the predecessor fields of each node in the ring define the ring and the transmission order. A station receives the token from its predecessor, transmits data, and passes the token to its successor. Here is an illustration of the token frame.

FC	RA	DA	SA	Seq	GenSeq	
1	6	6	6	4	4	bytes

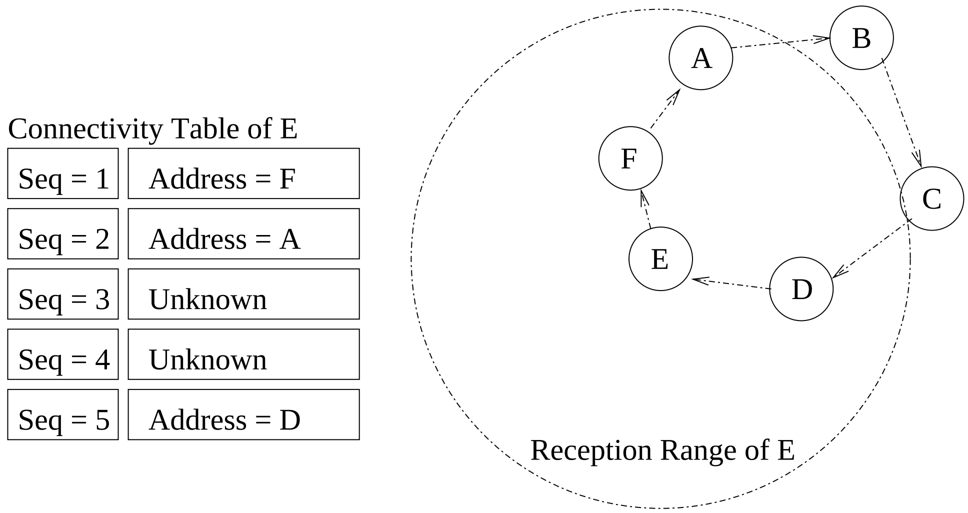

Figure 2.1: Connectivity Table

FC stands for Frame Control and it identifies the type of packet, such as Token, Solicit Successor, Set Predecessor, etc. In addition, the source address (SA), destination addresses (DA), ring address(RA), sequence number(Seq) and generation sequence(GenSeq) number are included in the token frame. The ring address refers to the ring to which the token belongs. The sequence number is initialized to zero and incremented by every station that passes the token. The generation sequence number is initialized to zero and incremented at every rotation of the token by the creator of the token.

The Connectivity manager resident on each node tracks transmissions from its own ring and those from other nearby rings. By monitoring the sequence number of the transmitted tokens, the Connectivity Manager builds an ordered list of stations in its own ring. The Connectivity Table of the manager holds information about its ring (Figure 2.1). In Figure 2.1, station E monitors the successive token transmission from F to D before the token comes back to E. At time 0, E transmits the token with sequence number 0, at time 1, F transmits the token with the sequence number 1, and so on. E will not hear the transmission from B and C, but when it hears transmission from D, E will notice that the sequence number has been increased by 3 instead of 1. This indicates to E that there were two stations that it could not hear between A and D.

The Ring Owner is the station that has the same MAC address as the ring address. A station can claim to be the ring owner by changing the ring address of the token that is being passed around.

Stations rely on implicit acknowledgements to monitor the success of their token transmissions. An implicit acknowledgement is any packet heard after token transmission that has the same ring address as the station. Another acceptable implicit acknowledgement is any transmission from a successive nodes regardless of the ring address in the transmission. A successive node is a station that was in the ring during the last token rotation. In other words, the successor stations are those present in the local connectivity table.

Each station resets its IDLE_TIMER whenever it receives an implicit acknowledgement. If the token is lost in the ring, then no implicit acknowledgement will be heard in the ring, and the IDLE_TIMER will expire. When the IDLE_TIMER expires, the station generates a new token, thereby becoming the owner of the ring.

To resolve multiple tokens (to delete all tokens but one), the concept of priority is used. The generation sequence number and the ring address define the priority of a token. A token with a higher generation sequence number has higher priority. When the generation sequence numbers of tokens are the same, ring addresses of each token are used to break the tie. The priority of a station is the priority of the token that the station accepted or generated. When a station receives a token with a lower priority than itself, it deletes the token and notifies its predecessor without accepting the token. With this scheme, it can be shown that the protocol deletes all multiple tokens in a single token rotation provided no

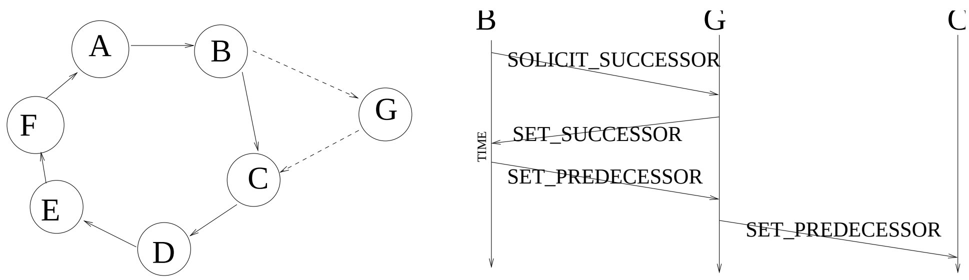

Figure 2.2: Joining

more tokens are being generated.

The ring recovery mechanism is invoked when the monitoring node decides that its successor is unreachable. In this case, the station tries to recover from the failure by reforming the ring. The strategy taken by the WTRP is to try to reform the ring by excluding as small a number of nodes as possible. Using the Connectivity Manager, the monitoring station is able to quickly find the next connected node in the transmission order. The monitoring station then sends the SET_PREDECESSOR token to the next connected node to close the ring.

WTRP allows nodes to join a ring dynamically, one at a time, if the token rotation time (sum of token holding times per node, plus overhead such as token transmission times) would not grow unacceptably with the addition of the new node. As illustrated in Figure 2.2, suppose station G wants to join the ring. Let us also say that the admission control manager on station B invites another node to join the ring by sending out a SOLICIT_SUCCESSOR. The Admission Control Manager waits for the duration of the response window for interested nodes to respond. The response window represents the window of opportunity for a new node to join the ring. The response window is divided into slots of the duration of the SET_SUCCESSOR transmission time. When a node that wants to join the ring, such as G, hears a SOLICIT_SUCCESSOR token, it picks a random slot and transmits a

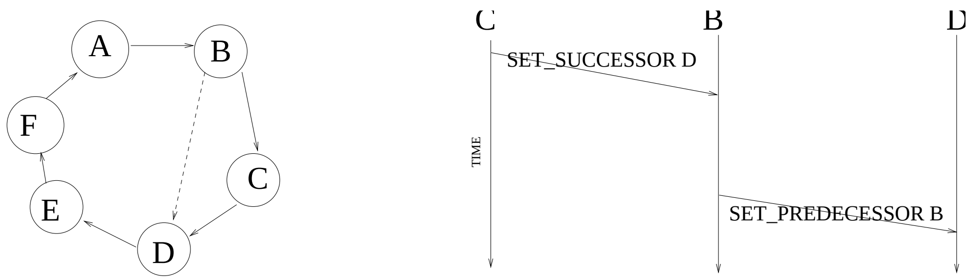

Figure 2.3: Exiting

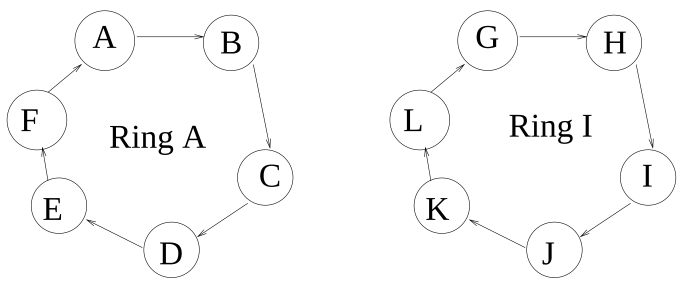

Figure 2.4: Multiple Rings

SET_SUCCESSOR token. When the response window passes, the host node, B can decide among the slot winners. Suppose that G wins the contention, then the host node passes the SET_PREDECESSOR token to G, and G sends the SET_PREDECESSOR to node C, the successor of the host node B. The joining process concludes.

As shown in Figure 2.3, suppose station C wants to leave the ring. First, C waits for the right to transmit. Upon receipt of the right to transmit, C sends the SET_SUCCESSOR packet to its predecessor B with the MAC address of its successor, D. If B can hear D, B tries to connect with D by sending a SET_PREDECESSOR token. If B cannot hear D, B will find the next connected node, in the transmission order, and send it the SET_PREDECESSOR token.

In Figure 2.4, we can see that the ring address of a ring is the address of any one of the stations in the ring. The station is called the owner of the ring. In the example, the owner of ring A is station A. Because we assume that the MAC address of each station is unique the ring address is also unique. The uniqueness of the address is important, since it allows the stations to distinguish between messages coming from different rings.

To ensure that the ring owner is present in the ring, when the ring owner leaves the ring, the successor of the owner claims the ring address and becomes the ring owner. The protocol deals with the case where the ring owner leaves the ring without notifying the rest of the stations in the ring as follows. The ring owner updates the generation sequence number of the token every time it receives a valid token. If a station receives a token without its generation sequence number updated, it assumes that the ring owner is unreachable and it elects itself to be the ring owner.

Chapter 3

Specification of the Protocol

We will describe in this section the timers and the frame formats of the protocol. The timers are important in terms of policing the data flow, regenerating a new token in case of lost token, retransmission of tokens, and recovery from failures as described in Section 2. The frame formats are also precisely defined here. Brief explanation follows each format.

3.1 Timers

1. IDLE_TIMER - The IDLE_TIMER is set to the MAX_IDLE_TIME and starts to count down. It is reset whenever the station receives an implicit acknowledgement. When the IDLE_TIMER expires ($IDLE\_TIMER < 0$), the station claims token.

2. INRING_TIMER - The INRING_TIMER is set to the MAX_NO_TOKEN_RECEIVED and starts to count down whenever the station receives a token. If the INRING_TIMER expires (INRING_TIMER < 0), the station assume that it has been kicked out of the ring, and exits the ring.

3. TOKEN_PASS_TIMER - The TOKEN_PASS_TIMER is set to the MAX_ACK_TIME

whenever a station sends a token and starts to count down. If the timer expires without the station receiving an implicit acknowledgement of the transmission, it assumes that the transmission was unsuccessful and transmits the token again.

4. TOKEN_HOLDING_TIMER - The TOKEN_HOLDING_TIMER is set to the MAX_TOKEN_HOLDING_TIME and starts to count down when token is received. While the timer is positive, the station can transmit data. When the timer expires, the station stops the transmission and passes the token.

The following relationship must hold

1. MAX_TOKEN_HOLDING_TIME < MAX_IDLE_TIME < MAX_NO_TOKEN_RECEIVED

2. MTRT[1] < MAX_IDLE_TIME

3.2 Frame Formats

3.2.1 Frame Control Field

1. Control Frame

0 0	C C C C C C

[1]MTRT is the Maximum Token Rotation Time defined more precisely in Section 5

C	**C**	**C**	**C**	**C**	**C**	Type
0	0	0	0	0	0	TOKEN
0	0	0	0	0	1	CLAIM_TOKEN
0	0	0	0	1	0	SOLICIT_SUCCESSOR
0	0	0	0	1	1	SET_PREDECESSOR
0	0	0	1	0	0	SET_SUCCESSOR
0	0	0	1	0	1	TOKEN_DELETED

2. Data Frame

F F	M M M	P P P

F	**F**		Frame Type
0	1		data
1	0		reserved[2]
1	1		reserved
M	**M**	**M**	Mac Action
0	0	0	Request with no response
0	0	1	Request with response

P	P	P	Priority
1	1	1	highest priority
1	1	0	
1	0	1	
1	0	0	
0	1	1	
0	1	0	
0	0	1	
0	0	0	lowest priority

3.2.2 Sequence Control

1. Sequence Number (Seq)

   Whenever a station passes a token the sequence number is increased. The counter wraps around to 0 when it reaches 2^{32}.

2. Generation Sequence Number (GenSeq)

   Whenever the owner of the token (the station that has the same MAC address as the ring address of the token), the owner increments the generation sequence number. The counter wraps around when it reaches 2^{32}.

   Together the sequence number and the generation sequence number defines the priority of the token. The priority is used for resolving multiple token resolution.

3.2.3 Address Fields

1. Destination Address Field (DA) - The MAC address of the packet destination.

2. Source Address Field (SA) - The MAC address of the packet source.

3. Ring Address Field (RA) - The MAC address of the station that generated the token.

3.2.4 Invalid Frame

We have assumed that the physical layer filters out the garbled packets. In addition The following are invalid frames that the MAC layer can discard.

1. The FC field is undefined.

2. DA and SA are the same.

3.2.5 Enumeration of Frame Types

1. Token

0 0 0 0 0 0 0 0	RA	DA	SA	Seq	GenSeq	
1	6	6	6	4	4	bytes

The token is used to transfer the right to transmit.

2. Claim Token

0 0 0 0 0 0 0 1	RA	DA	SA	
1	6	6	6	bytes

The Claim Token is used when a station generates the token in the case where a station creates a ring. It is also used when a station regains the token in the case of lost token.

3. Solicit Successor Token

0 0 0 0 0 0 1 0	RA	DA	SA	SucAddr	Reserv	
1	6	6	6	6	8	bytes

The SOLICIT_SUCCESSOR token updates the successor field of a station. It is used for inviting another node to join the ring.

4. Set Predecessor Token

0 0 0 0 0 0 1 1	RA	DA	SA
1	6	6	6

bytes

The SET_PREDECESSOR token updates the predecessor field of a station. It is used for both joining the ring and exiting the ring.

5. Set Successor Token

0 0 0 0 0 1 0 0	RA	DA	SA	NS
1	6	6	6	6

bytes

The SET_SUCCESSOR token updates the successor field of a station. It is used for both joining the ring and exiting the ring.

6. Token Deleted Token

0 0 0 0 0 1 0 1	RA	DA	SA
1	6	6	6

bytes

The TOKEN_DELETED token is used to give predecessor notification that the token has been deleted. This is to prevent the predecessor from invoke the ring recovery mechanism.

7. Data

0 1 M M M P P P	RA	DA	SA	Data
1	6	6	6	

bytes

Chapter 4

Implementation

Ideally the implementation should be done on the network card, since

1. access to the physical layer information can improve the performance of the protocol by decreasing the slot time as described in Section 2.3.

2. the context switch between the micro-controller of the network card and the CPU introduces delay.

3. the processing of the protocol can be blocked by other processes.

4. the processing of the protocol in the CPU takes time.

5. the processing of the protocol takes the CPU time and memory away from other resident processes.

We have implemented the token ring protocol on top of the WaveLAN[16] card which implements the IEEE802.11[5] protocol. This is done by prepending the token ring header to the IP packet and broadcasting all packets over the IEEE802.11 link. The implementation of the Wireless Token Ring Protocol is divided into three parts: the kernel module, the

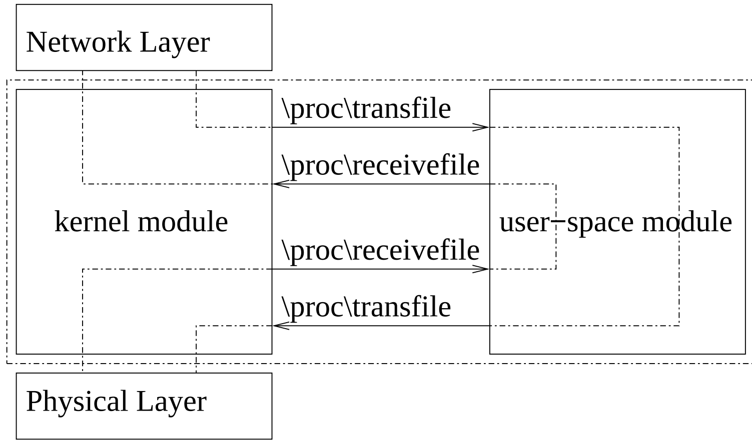

Figure 4.1: Software Architecture

interface between kernel and the user-space module, and the user-space module. The kernel module communicates with the network layer and the physical hardware, and manages the PCMCIA card. The finite state machine of the protocol is implemented in the user-space module. The flow of execution passes from the kernel to the user space and back to the kernel for packet transmissions and receptions.

The goal of software implementation of the protocol is:

1. to study the performance of the protocol.

2. to catch errors missed during the simulation phase.

3. to implement quality of service on top of an arbitrary network card.

In this section, the kernel driver, the interface between the kernel and the user-space module, and the user-space driver are discussed.

4.1 Linux Kernel Driver

The version of the Linux kernel and the PCMCIA package that this implementation based on is 2.2.14.5 (RedHat 6.2 distribution) and 3.1.7 respectively. We have modified the wavelan2_cs.c[18] driver to implement our token ring driver.

4.1.1 Kernel Process

The Linux Kernel is implemented as a single thread. This means that the module we attach to the kernel is not a self-contained process, but a set of functions and data structures that interface with the kernel.

4.1.2 Memory Space

A user and the kernel process are in a different virtual address space. Under normal operations, the memory of a user and the memory of the kernel processes are kept physically separated. This prevents the processes from accessing each other's memory. The separation can be a problem when exchanging data between a user process and the kernel process. System calls are a solution to this problem. Unix transfers execution from user space to kernel space through system calls and hardware interrupts. As described by [14], the "kernel code executing a system call operates on behalf of the calling process and is able to access data in the process's address space." To illustrate this point, we have included excerpts from [15].

Below is the kernel function that executes the write system call. One of the arguments of the function is the *buf* which points to the user space data.

```
static ssize_t write(
   struct file *file,  /* The file itself */
```

```
   const char *buf,    /* The buffer with input */
   size_t length,     /* The buffer's length */
   loff_t *offset)    /* offset to file - ignore */
{
. . .
}
```

4.1.3 User-space Device Driver

We implemented our token ring driver partially as a user-space device driver with an interface to the kernel. [14] discusses the advantages and the disadvantages of implementing a device driver in user space. The following list is a modified one from [14].

1. Advantages
   (a) The full C library can be linked in.
   (b) A conventional debugger can be run on the driver code.
   (c) If the module hangs, you can simply kill it.
   (d) The user memory is swappable, whereas the kernel memory is not.
   (e) Commercial code generating tools such as Teja [17] can be used to generate code.
2. Disadvantages
   (a) Hardware interrupts are not available directly.
   (b) Direct access to memory and I/O ports is cumbersome.
   (c) Response time is slower. If the driver has been swapped to disk, response time is unacceptably long.
   (d) It is difficult to design a driver program that allows concurrent access to a device.

(e) It is difficult to design a driver that will support multiple instances of a device.

The reason for implementing the token ring protocol in the user-space is to use the Teja design environment. Teja allows developers to design, test, and generate code for a protocol from finite state machine specifications.

Among the disadvantages, the slow response time and the difficulty in supporting multiple instances of the device are important when writing a network device driver. The slow response time due to process being swapped out to hard disk can be controlled by a Unix command called **mlock**. The **mlock** or **mlockall** allows one to lock the memory by pages. However, locking the user-space driver in the computer memory is costly if the user process uses a large library. Our implementation grows from 0.17 M-bytes of compiled code to 0.64 M-bytes during run time, due to linkage of large libraries and memory allocation of buffers.

The support for multiple instances of the device driver is not currently supported since we are initially targeting communication networks with one wireless LAN card per station.

4.1.4 Kernel Modules

One of the useful things about Linux is its use of kernel modules. Kernel modules allow for the expansion of the kernel code at run time. This means that we can compile individual modules separately and attach them to the kernel when the system is running, instead of recompiling the entire kernel. The kernel portion of the token ring driver is modified from [18] and is implemented as a kernel module.

4.1.5 Viability of Real Time Processes in Linux

The Linux kernel runs at 100 Hz by default in x86 compatible hardware and 1024 Hz in Sun. The 100 Hz clock translates into the clock granularity of 10 ms. The coarse granularity creates problems because when a Teja process is idle and goes to sleep, the process will sleep for at least the granularity of the clock. This is a problem since the token ring protocol runs much faster than 10 ms, and the protocol uses many timers. To combat this problem, one can increase the granularity of the clock by increasing the HZ value in <asm-i386/param.h> and then recompiling the kernel. Increasing the granularity results in a more responsive system, however the price paid is the additional overhead of processing more clock interrupts. We have increased the granularity of the clock on our P3 500 MHz laptops to 2048 Hz. This has resulted in reduced joining and fault recovery time, and increased stability of the token ring.

4.1.6 Actual Implementation

Below are small portions of the code that illustrate how the interface is implemented. The understanding of the general idea is necessary to understand the implementation. For details on the implementation, please reference the appendix section. A brief explanation follows each function.

At the initialization of the network device driver, all handlers are attached to the device. The dev->hard_start_transmit receives packets from the IP layer and the link->irq.Handler handles packets from the network card. The dev->tbusy flag is used by the logical link control to control the flow. The skbuff (</Linux/skbuff.h>) is the data structure used by the Linux kernel to pass the packet up and down the communication protocol stack without memcopying. The interfaces between the IP and data-link layers, and the data-link layer and the device library use a pointer to the skbuff structure as an argument. When sending a packet down the stack, the data field of the skbuff is an Ethernet packet. When receiving a packet from the network card, the Ethernet header is stripped off.

```
int init_module(void)
{
  register_PCMCIA_driver(&dev_info, &adapter_attach, &adapter_detach);
  proc_register(&proc_root, &Trans_File);
  . . .
}

void cleanup_module(void)
{
  unregister_PCMCIA_driver(&dev_info);
  proc_unregister(&proc_root, Trans_File.low_ino);
  . . .
}
```

A kernel module needs both the init_module and the cleanup_module function to register and deregister itself with the running kernel. In the function, init_module, the module registers itself as a PCMCIA driver. Also in the init_module function, the proc_register function are used to register the necessary files under /proc file systems that is used to communicate the data between the kernel and the user process. One can see the deregistration process in the cleanup_module function.

```
static dev_link_t * adapter_attach(void)
{
  dev_link_t *link;
  struct device *dev;

  . . .
  link->irq.Handler = &wvlan2_isr;
```

```
  . . .
  /* Make up a wvlan2-specific-data structure. */
  dev->priv = kmalloc(sizeof(struct wvlan2_private), GFP_KERNEL);
  . . .
  /* Setup the device defaults as an Ethernet device. */
  ether_setup(dev);
  . . .
  dev->hard_start_xmit = &token_ring_tx;
  . . .
  dev->tbusy = 1;
  . . .
}
```

The above function is needed to attach a PCMCIA driver. Two important handlers are link->irq.Handler to handle hardware interrupts from the network card and dev->hard_start_xmit to handle the packet sent down by the IP layer. One important structure that can be utilized in the implementation of the data-link layer with states is dev->priv. Private states for each instance of the device driver can be implemented using this. For instance, one can implement a finite state machine. ether_setup(dev) lets the kernel know that this is an Ethernet device driver. dev->tbusy is used by the logical link control to do the flow control. When this value is set to 1, the logical link control does not try to send the packet to the Ethernet driver.

```
static int
token_ring_tx(struct sk_buff *skb, struct device *dev)
{
  if (dev->tbusy)
```

```
  {
    if ((jiffies - token_ring_trans_start) >= HCF_TX_TIMEOUT)
      {
        . . .
        token_ring_trans_start = jiffies;
        set_devicetbusy(0);
      }
    }
  else
  {
    trans_data_len = skb->len;
    memcpy(trans_data, skb->data, skb->len);
    token_ring_trans_start = jiffies;
    set_trans_data_flag(1);
    wake_up(&trans_wait);
    . . .
  }
}
```

The above function is called when receiving data from the IP layer. If there has been a timeout, the function resets the transmission timer and turns off the transmission busy flag (set_devicetbusy(0)). If there has not been a timeout, the function consumes the buffer and saves the buffer and the buffer size for the user module.

```
static int
my_wvlan2_rx(struct device *dev)
{
```

```
  struct sk_buff *skb;
  struct wvlan2_private  *lp;
  . . .
         skb->dev = dev;
  . . .
         GET_PACKET(dev, skb, pktlen);
  . . .
             receive_data_len = skb->len;
             memcpy(receive_data, skb->data, skb->len);
             set_receive_data_flag(1);
             DEV_KFREE_SKB(skb);
  . . .
}
```

The above function is called when hardware interrupt is raised by the network interface card. This means that the network interface card received a packet from the medium (i.e. packet has arrived). The function GET_PACKET actually moves the packet from the memory of the network card into the computer memory. If the packet is a data packet the device driver forwards it to the top without going through the token ring driver.

4.2 Implementation of the Interface between Kernel and User Module

From Figure 4.2, one can see how the kernel and the user modules interact in more detail. The following are the details of each operation.

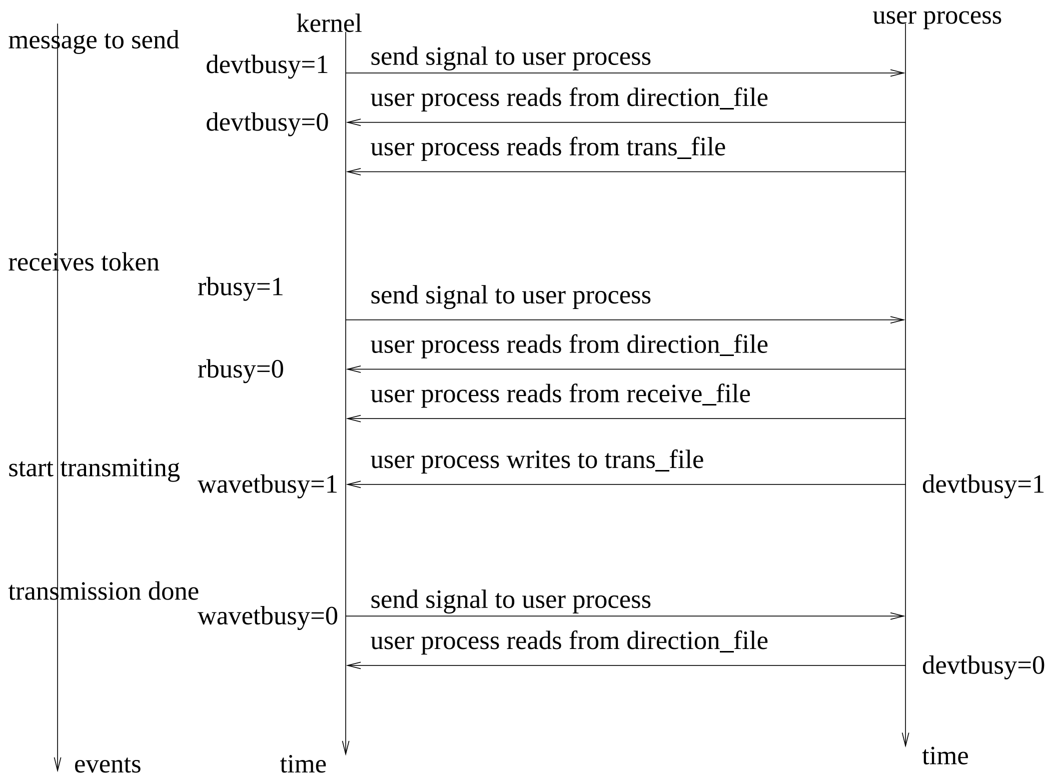

Figure 4.2: Interaction between the Kernel and User Modules

4.2.1 Sending Messages

When a new message arrives, the kernel module sets dev->tbusy to 1. This variable is used by the logical link control for flow control. By setting dev->tbusy to 1, the kernel module prevents the logical link control from sending another packet until the user module has received the packet. When receiving the signal from the kernel, the user process checks the direction_file to find out why the kernel has interrupted the user module. The user module finds out that the kernel module has a message to be transmitted by examining the file. Then the user module proceeds to receive the message by reading from the trans_file.

Although not shown in the figure, it is important to mention that the user module can set the dev->tbusy flag at the kernel side by writing into a file called dev->tbusy_file. The user process sets dev->tbusy to 1 whenever its queue is full. And when space becomes available in the queue, the user module sets the dev->tbusy flag to 0.

4.2.2 Receiving the Token

When the token arrives at the network interface card, the interface card sends a hardware interrupt to the kernel module. The kernel module then sets rbusy to 1 and sends the signal to the user. The rbusy flag indicates the user module is busy, receiving information from the kernel and the kernel should refrain from giving the user module another packet. The user module finds out that the kernel module has received a token by inspecting the direction_file. Then the user module proceeds to receive the packet by reading from the receive_file.

4.2.3 Packet Transmission

The transmission is allowed when the station has the right to transmit. In such cases where the station receives a valid token as in Figure 4.2, the station is allowed to transmit. At this time, the user module writes into the trans_file. The wavetbusy flag at the user module is set to 1 to prevent the user from sending another packet before the network card finishes transmitting. When the network card finishes transmission, it gives the kernel module a hardware interrupt. At this moment the kernel module sends a signal to the user, prompting it to inspect the direction_file, and the user module finds out that the transmission is over.

4.2.4 Sending a Signal to the User Process

There are two ways to interrupt the user. The first way is to implement the polling function at the user side to poll on a *proc* file. The second way is to directly send the signal to the user process. The second method is implemented in the final version of the software. However, the first version has also merits. It was implemented in the first implementation and the method performed well. In addition, no documentation in the linux community exists showing how to do this. The second method is an improvement on the first method.

Method 1 The user process polls on the direction_file to see if there are any "changes" in the file. The trick here is to write our own polling function on the kernel side and wake up the user process whenever the kernel module wants to. The following is the polling function at the kernel module.

```
struct wait_queue * direction_wait = NULL;
static unsigned int direction_poll(struct file *file, poll_table * wait)
{
  poll_wait(file, &direction_wait, wait);
  if (direction_data_flag == 1)
    {
      return POLLIN | POLLRDNORM;
    }
  else
    return 0;
}
```

The linux kernel periodically checks the wait queues to see if any process is sleeping on a poll function. If there are any such process in the queue, it evaluates the poll function that is attached to the file. In our case, we attached the above poll function to direction_file. According to our function, whenever the `direction_data_flag` is set, the poll function will return non zero, waking up the user process. In a time critical application, it may be necessary for the kernel to evaluate the poll function right away after the `direction_data_flag` is set. And this is done using the following command.

```
wake_up(&direction_wait);
```

The command forces any polling functions that are associated with the direction_wait queue to be evaluated.

Method 2 To be able to send a signal to a user process, the kernel module needs to be able to find the task_struct that is associated with the process. This is done by the following function.

```
static ssize_t getpid_input(struct file *file, const char *buf, size_t len,
                            loff_t *offset)
{
. . .
  tokenring_task = NULL;
  read_lock(&tasklist_lock);
  for_each_task(p) {
  if (p->pid == tokenring_pid)
    {
      tokenring_task = p;
      break;
    }
  }
  read_unlock(&tasklist_lock);
  . . .
}
```

When the user module writes to the file `\proc\getpid_file`, the above procedure is invoked. The user module gives as the argument *buf*, the pid of the module. At this time the kernel module goes through all the task structure that is registered with the kernel and finds the task structure with the matching process id.

The kernel module later sends the SIGUSR1 to the user by calling `send_sig_info(SIGUSR1, ,tokenring_task)`.

4.3 Implementation of the User Module

The user module implements the MAC protocol. We will describe the implementation as a finite state machine. The finite machine essentially is a Mealy machine where the action is associated with the transition. The transient states are represented as small circles. This section will give an overview of the finite state machine without much detail. Please refer to the appendix for further details.

4.3.1 Mac Macro State (Figure 4.3)

The Macro State is the main finite state machine of the protocol. The finite state machine begins with the offline state, and the transition from the offline state to the offring state is immediate. This transition allocates all memories that are needed during the execution[1] and sets up all the dependencies between modules.

As one can see from the finite state machine, there are two ways to go to the inring state. They are the path that goes though the enter state and the path that goes through the claim token state, which corresponds to the joining process, and claiming token process respectively.

[1]The NIMR connectivity table is the only dynamic memory allocation after this point.

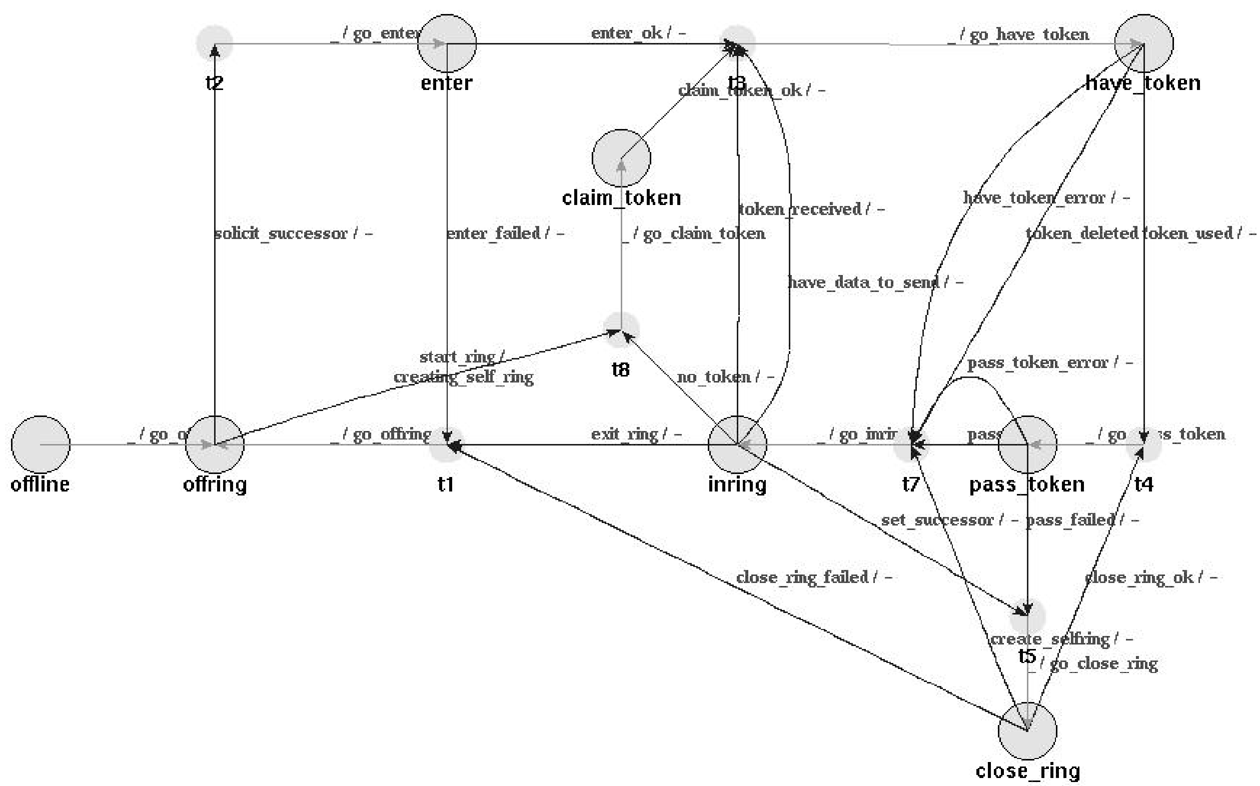

Figure 4.3: Mac Macro State

During normal operations, the state will circulate from the inring to the have_token, the have_token to the pass_token, and the pass_token back to the inring.

The close_ring state is reached when there is need for ring repair.

from	**to**	**reason**
offline	offring	initialized
	enter	received the `solicit_successor`
	claim_token	the `idle_timer` expired
enter	have_token	received the `set_successor`
	offring	enter failed
claim_token	have_token	broadcasted the `claim_token` now use the token
inring	have_token	received the `token`
	offring	heard a transmission from an another ring
have_token	inring	token deleted or heard an another transmission
	pass_token	used the token
pass_token	close_ring	didn't receive the implicit acknowledgement
	inring	heard the implicit acknowledgement
	inring	heard transmission from an another ring
close_ring	inring	close ring attempt was a success
	pass_token	close ring attempt was a failure, try an another node
	offring	the close ring attempt failed and tried enough times

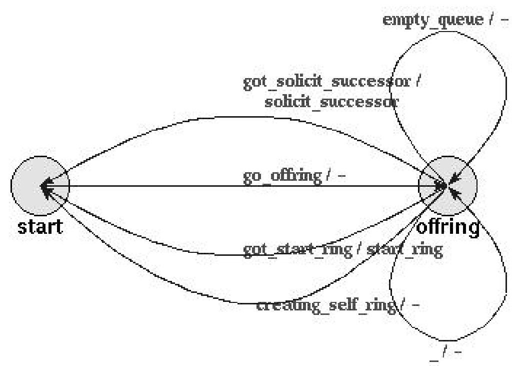

Figure 4.4: Offring Micro State

4.3.2 Offring Micro State (Figure 4.4)

In the offring state, the MAC layer waits until it either receives the invitation to join a ring, decides that there is no ring in the medium, or receives an explicit command to create a ring from a management module defined in Section 1.4.

from	**to**	**reason**
start	offring	start the Offring Micro State
offring	start	got the `solicit_successor`, move to the Enter Micro State
	offring	received an irrelevant packet, update the connectivity cache and discard
	start	the `idle_timer` has expired, create own ring
	start	an explicit command to create ring from a management module is received

4.3.3 Claim Token Micro State (Figure 4.5)

In the Claim Token Micro State, the station broadcasts `claim_token`.

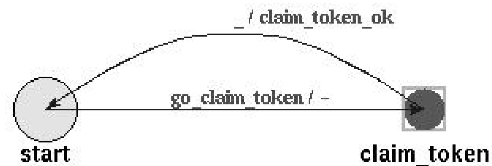

Figure 4.5: Claim Token Micro State

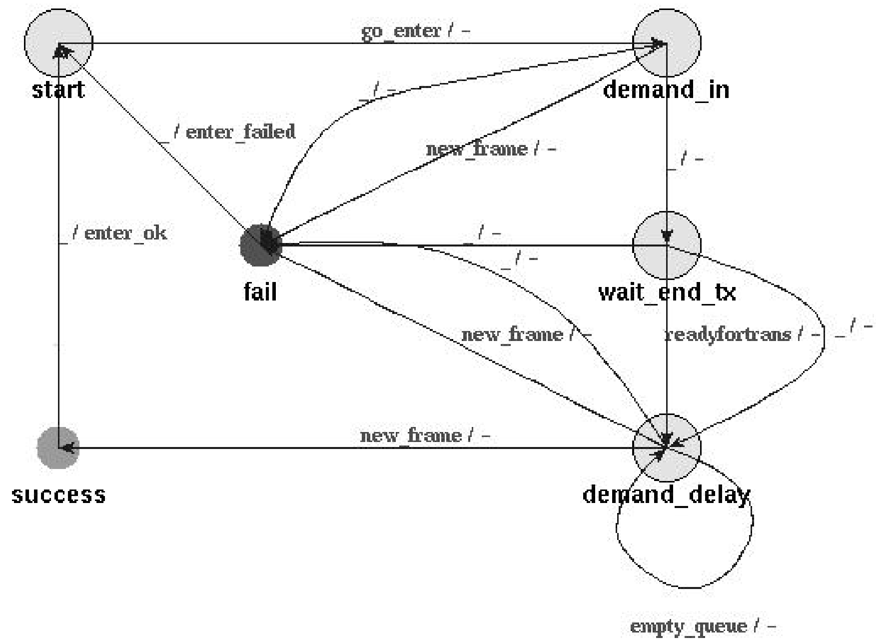

Figure 4.6: Enter Micro State

from	to	reason
start	claim_token	start the Claim Token Micro State
claim_token	start	received the “done transmitting” signal from the kernel

4.3.4 Enter Micro State (Figure 4.6)

The contention for the joining process is implemented in this state. When the station solicits successor from the Offring Micro State, it enters this state. During the transition from demand_in to wait_end_tx, the stations contend by sending the set_successor. After the transmission, the station waits for an acknowledgement in the demand_delay state.

from	**to**	**reason**
start	demand_in	start the Enter Micro State
demand_in	fail	heard an another transmission
	fail	not connected with the successor
	wait_end_tx	wait a random slot and transmit the `set_successor`
wait_end_tx	fail	heard an another transmission
	demand_delay	failed to receive the "done transmission" signal from the kernel
	demand_delay	received the "done transmission" signal from the kernel
demand_delay	fail	contention timer expired
	success	received set successor from the soliciting node
	demand_delay	heard an another contender

4.3.5 Inring Micro State (Figure 4.7)

The Inring Micro State corresponds to the "idle" state of the protocol. In this state the station monitors incoming packets and builds the connectivity table. When the station receives a token destine for itself, the flow of execution moves to the Have Token Micro State.

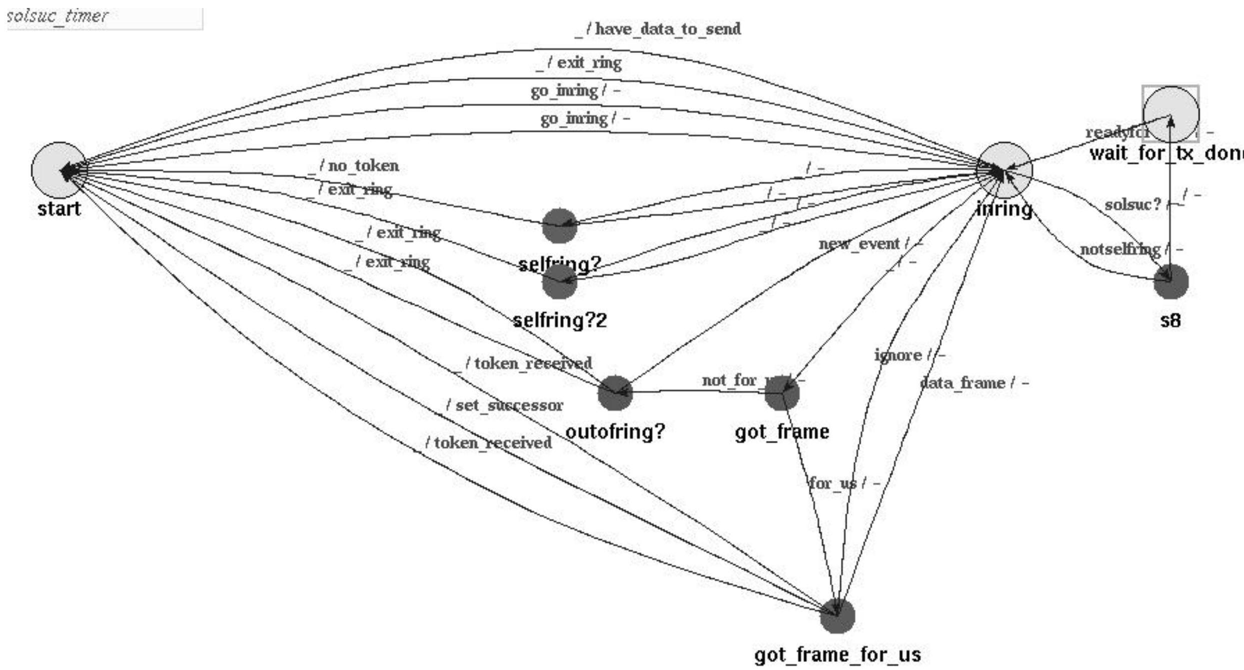

Figure 4.7: Inring Micro State

from	to	reason
start	inring	start Inring Micro State
inring	s8	the `sol_suc_timer` expired
	selfring?	the `idle_timer` expired
	selfring?2	the `inring_timer` expired
	got_frame	received a frame from the physical layer
s8	inring	not selfring, reset the `sol_suc_timer`
	wait_for_tx_done	transmit the `solicit_successor`
wait_for_tx_done	inring	received the “done transmitting” signal from the kernel

from	**to**	**reason**
selfring?	start	this station is not a selfring, reset timers and go to the Claim Token Micro State
	inring	this station is a selfring, reset timers
selfring?2	start	this station is not a selfring, reset timers and go to the Claim Token Micro State
	inring	this station is a selfring, reset timers
got_frame	got_frame_for_us	the frame is intended for this station
	outofring?	the frame is not intended for this station
outofring?	start	this station is out of the ring, go to the Offring Micro State
	inring	not out of ring, process the frame and go back to inring state
got_frame_for_us	inring	the frame is a data frame, send it to the IP sender and go back to inring state
	inring	received an invalid frame, ignore
	start	the `token` received, go to the Have Token Micro State
	start	the `set_successor` received, go to the Have Token Micro State
	start	the `solicit_successor` token received, go to Have Token Micro State

4.3.6 Have Token Micro State (Figure 4.8)

The MAC finite state machine enters this Micro State when it receives the right to transmit. One of the main functions of this Micro state is the implementation of the multiple token

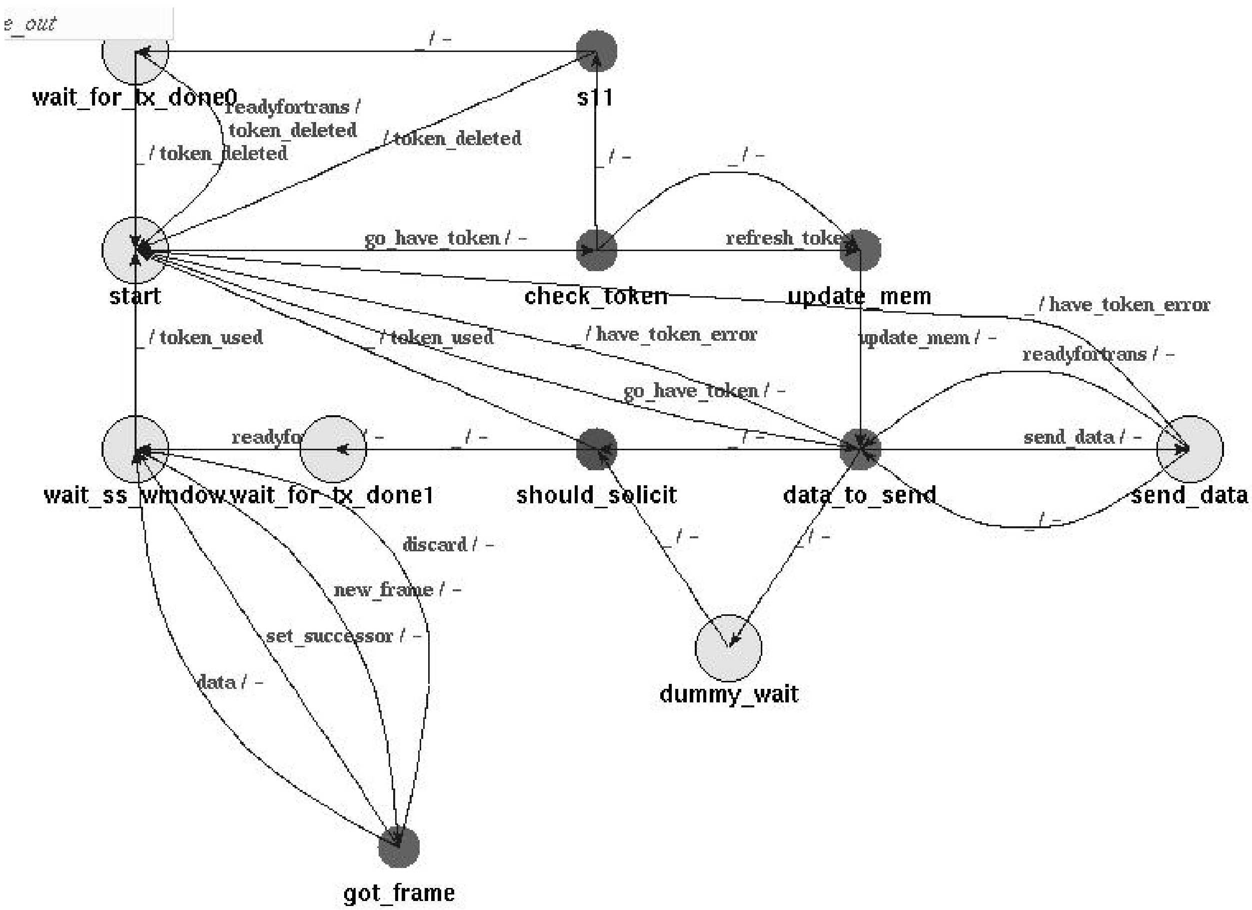

Figure 4.8: Have Token Micro State

resolution protocol, which is implemented in a loop that involves the state check_token, s11 and wait_for_tx_done0. The token inspects the token and deletes it if it has lower priority than that of the station. If the token is accepted, the sequence numbers of the token is updated (update_mem).

If there is data to be transmitted, it is transmitted up to the token holding time, while switching between the state data_to_send and send_data. After the transmission of data, the station solicits a successor before sending the token.

from	**to**	**reason**
start	check_token	received a token before, invoke the multiple token resolution protocol
	update_mem	token is generated by this station, refresh the token
	update_mem	token is not generated by this station, do not refresh the token
	data_to_send	no need to invoke the multiple token resolution protocol, since no token has been accepted in the past (probably just came on line)
check_token	s11	this token has equal or less priority than the token that this station has accepted the last time.
s11	start	this station is the sole station, so this station is not required to send the `token_deleted` token.
wait_for_tx_done0	start	received an another transmission
	start	received the "done transmitting" signal from the kernel
data_to_send	send_data	there is data to be transmitted and the token holding time is still positive
	dummy_wait	used for testing purposes, not used any more

from	to	reason
send_data	data_to_send	received the "done transmitting" signal from the kernel
	data_to_send	failed to receive the "done transmitting" signal from the kernel
	start	heard an another transmission
	should_solicit	flip a coin to decide to solicit successor
should_solicit	start	do not solicit successor
	wait_for_tx_done1	solicit successor by sending the `solicit_successor` packet
wait_for_tx_done1	wait_ss_window_wait	finished transmitting
wait_ss_window_wait	got_frame	received a new frame from the physical layer
	start	the `token_holding_timer` is expired
got_frame	wait_ss_window_wait	identify and process the frames accordingly

4.3.7 Pass Token Micro State (Figure 4.9)

In the Pass Token Micro State, the station transmits the token (pass_token). After transmission, the station monitors if its token transmission has been successful (check_pass). This is done by monitoring successive transmissions. If the station does not hear an acknowledgement for the first token transmission, it tries again by retransmitting from check_pass to wait_for_tx2. If there is no acknowledgement for the second transmission, it moves into the Close Ring Micro State.

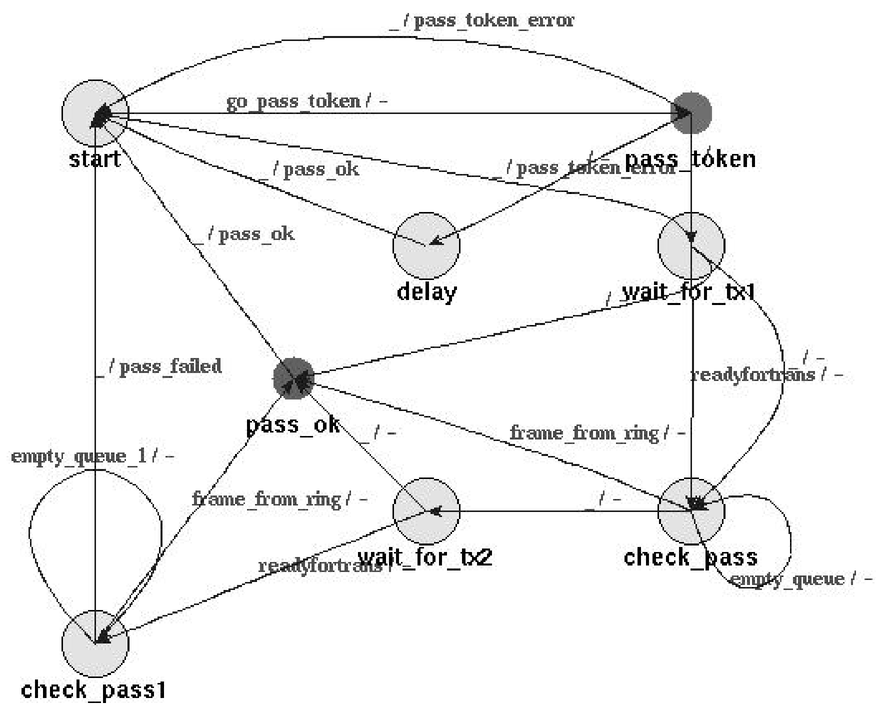

Figure 4.9: Pass Token Micro State

from	**to**	**reason**
start	pass_token	start the Pass Token Micro State
pass_token	wait_for_tx1	transmitted the `token`
	delay	this is a selfring, don't need to send out anything
delay	start	the `delay_timer` expired
wait_for_tx1	check_pass	received the "done transmitting" signal from the kernel
check_pass	pass_ok	heard an another transmission
	start	received the implicit acknowledgement
wait_for_tx2	check_pass1	received the "done transmitting" signal from the kernel
check_pass1	pass_ok	heard an another transmission
	start	received the implicit acknowledgement

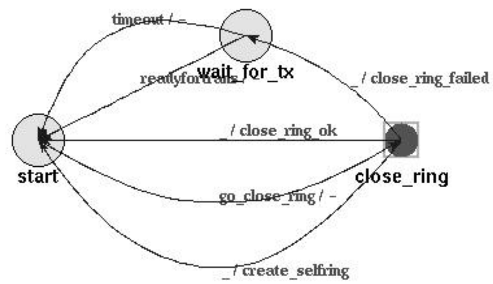

Figure 4.10: Close Ring Micro State

4.3.8 Close Ring Micro State (Figure 4.10)

The Close Ring Micro State checks to see if the station is connected. If the station is not connected, it kicks itself out of the ring. If this station is the sole station left, it creates a self ring. Otherwise, the station updates the successor of the current station and goes back to the Pass Token Micro State.

from	**to**	**reason**
start	close_ring	start the Close Ring Micro State
close_ring	start	this station is connected with the new station, try to close the ring
	start	this station is not connected with any other node, create a self ring
	wait_for_tx	close ring has failed, kick itself out of the ring by sending the `set_successor` packet
	start	received the "done transmitting" signal from the kernel
	start	failed receive the "done transmitting" signal from the kernel

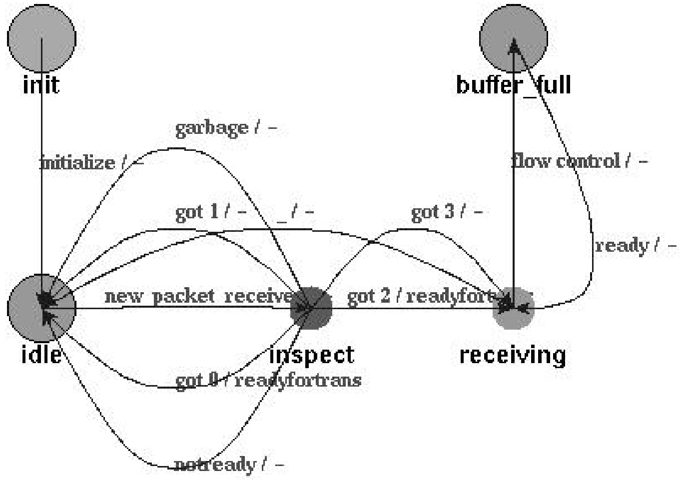

Figure 4.11: Physical Receiver

4.3.9 Physical Receiver (Figure 4.11)

This module is not part of the MAC protocol. This is one of the four helper finite state machines along with the Physical Sender, the IP receiver, and the IP sender. These modules act as the interface between the kernel and the MAC finite state machine.

After catching the interrupt from the kernel, the Physical Receiver needs to inspect the `direction_file` to see what kind of message is received (Section 4.2). One of the difficulties in designing these modules was the fact that the signal between the user and the kernel modules (Section 4.2) can be lost when the linux system is highly loaded. This can put the kernel or the user module into a inconsistent state, which may deadlock the two modules.

This is what is behind the four different signals used in this module. In the state inspect, there are four different branches that the state machine can take, based on the signal described in the following table:

signal received from kernel module	**kernel busy transmitting**	**kernel is not busy transmitting**
kernel received a packet	3	2
kernel didn't receive a packet	1	0

If none of the signals are lost, there is no need to have four different signals. Two signals are enough.

from	**to**	**reason**
init	idle	initialize
idle	inspect	received signal from the kernel
inspect	idle	received the "done transmitting" signal from the kernel
	idle	received garbage
	idle	received no data at this time
	receiving	received the "done transmission" signal and data from the kernel
	receiving	received data from the kernel
receiving	idle	the physical receiver can receive more packets
	buffer_full	the physical receiver is unable to receive more packets, wait until the queue is not full

4.3.10 Physical Sender (Figure 4.12)

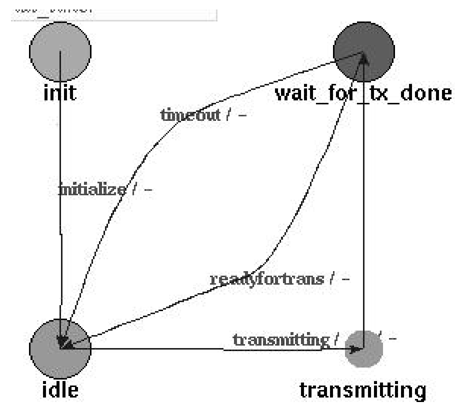

Figure 4.12: Physical Sender

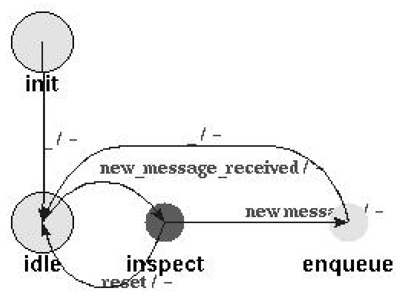

Figure 4.13: IP Receiver

from	**to**	**reason**
init	idle	initialize
idle	transmitting	has packets to transmit
transmitting	wait_for_tx_done	transmit the packet
wait_for_tx_done	idle	received the "done transmitting" signal from the kernel
	idle	failed to receive the "done transmitting" signal from the kernel

4.3.11 IP Receiver (Figure 4.13)

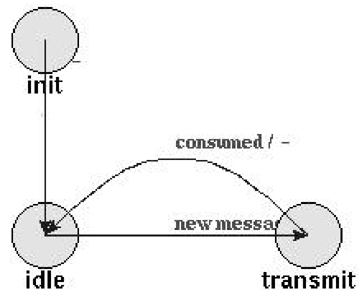

Figure 4.14: IP Sender

from	**to**	**reason**
init	idle	initialize
idle	inspect	received a signal from the kernel
inspect	idle	received a reset signal from the kernel
	enqueue	received data from the kernel

4.3.12 IP Sender (Figure 4.14)

from	**to**	**reason**
init	idle	initialize
idle	transmit	has data to send
transmit	idle	finished transmitting data

Chapter 5

Proof of Token Ring Protocol

5.1 Summary

We will prove that when transmission losses and topological changes of the graph stop at time t, and stations do not go into the OFFRING state voluntarily, then the algorithm will come to a stable state where all stations will belong to a ring at time $s > t$. The following is a brief outline of the proof.

We first prove that all equivalent tokens are deleted by multiple token resolution protocol in finite time. Then we prove that the bijection that represents correct relationship between predecessor and successor (see definitions under Section 5.2.4 for the exact definition) increases monotonically to the number of nodes in the graph in finite time. When the number of bijection converges, then we say that all rings are correct since rings are defined by predecessor and successor relationship.

5.2 Model

5.2.1 Constants

1. MTRT: Maximum Token Rotation Time. (Defined more precisely in lemma 5.3.3)
2. M: set of all MAC addresses
3. IDLE_TIME: the amount of time that a station waits before regenerating a token when the medium is quiescent. IDLE_TIME≥MTRT
4. INRING_TIME: the amount of time that a station waits when the station is not receiving any acceptable token, before going into the OFFRING state. INRING_TIME≥IDLE_TIME≥MTR
   INRING_TIME < 2 IDLE_TIME

5.2.2 Graph

1. Definitions
   (a) The adjacent graph, G(t), is defined by a set of undirected edges, E(t), and a set of stations, V(t), at time t.
   (b) A station represents a data-link layer of a communication station.
   (c) The set of edges, E(t), corresponds to the set of transmission links between stations. e(x,y)∈E(t), if and only if x is in the transmission range of y, x is in the reception range of y, y is in the transmission range of x, and y is in the reception range of x, then we say x and y are connected.
   (d) $|E(t)|$ - the number of the edges in the graph at time t.
   (e) $|V(t)|$ - the number of the stations in the graph at time t.
2. Assumptions

(a) If station x is in transmission range of station y, then y is also in the transmission range of x.

(b) If station x is in reception range of station y, then y is also in the reception range of x.

(c) No station hears corrupted messages (The physical layer filters out all corrupted messages.)

5.2.3 Token

1. Attributes

$p.type \in$ {SET_PREDECESSOR,SET_SUCCESSOR,TOKEN,SOLICIT_SUCCESSOR}

$p.ra \in M$ //ring address of the token

$p.sa \in M$ //source address of the token

$p.da \in M$ //destination address of the token

$p.seq \in Z$ //sequence number of the token

$p.genseq \in Z$ //generation sequence number of the token

2. Definitions

(a) $r_i(t)$ = a set of nodes in ring i.

(b) $t_i(t)$ = a set of tokens in $r_i(t)$.

(c) Token x and token y are said to be equivalent when their ring addresses are the same.

(d) The proriority of the token is as follows: Token x is said to have higher priority than y if its generation sequence number is higher than y. Given that x has the same generation sequence number as y, x has higher priority if it has the higher ring address.

(e) In this proof, we often refer to the same token in the future. This is not a cause for confusion since a token does not split into multiple tokens as suggested by our assumption of no transmission error.

5.2.4 Station

1. Attributes

x.genseq(t)∈Z // The generation sequence number of the last token that x accepted.

x.ra(t) ∈M // ring address of x

x.TS ∈M // the MAC address of x

x.NS(t) ∈M // the MAC address of the station to which x forwards tokens.

x.PS(t) ∈M // the MAC address of the station from which x accepts tokens.

2. Definitions

(a) x is said to be the owner of token p if x.TS = p.RA.

(b) The priority of a station is the priority of the token that the station accepted the last.

3. Assumptions

(a) The MAC address is unique to each station. x.TS≠y.TS if x≠y

5.2.5 Network

1. Definitions

(a) Network, N(t), is defined by a set of directed edges, L(t), and a set of vertices, V(t), at time t.

(b) For station x and y, an edge ps(x,y) exists if and only if x.PS = y, and an edge ns(x.y) exists if and only if x.NS = y.

(c) Set L(t) corresponds to the set of PS and NS mapping between stations. L(t) = {ps(x,y), ns(x,y) | for all x in V(t)}

(d) For stations x and y, if x.NS = y, y.PS = x, and if y can receive and accepts NORMAL tokens from x then, we say that x has the bijection with y.

(e) S = {$< x.y >$ | x has the bijection with y}

(f) A set of stations, $r_i(t)$, is called a ring if for all x$\in r_i(t)$, x has the bijection with its successor, y.

5.3 Proof

5.3.1 Assumptions

1. No transmission error occurs after t. (All transmissions are successful.)

2. Graph, G(t), remains constant after t.

3. Stations do not voluntarily go into the OFFRING state.

Lemma 5.3.1 *While a station is not in the OFFRING state, the priority of the station increases with time.*

A station does not accept a token from another station with a equal or lower priority than that of itself, by the implementation. Also, when a station generates a token upon expiration of the IDLE_TIMER, the station increases its priority by increasing the generation sequence number by 2. ■

Lemma 5.3.2 *Choose any token p at time $t = t_0$, and build an ordered list of paths taken by p, say $< (x_0, t_0), (x_1, t_1), (x_2, t_2), ..., (x_m, t_m) >$, where t_n is the time that token p visits the station x_n, and $t_i + 1 > t_i$. If there exists a station $x_i = x_j$ in the pair list such that $0 \leq i < j \leq m$ and $t_j - t_i < MTRT$, then there must be a k such that $i \leq k \leq j$, and x_k owns p.*

Let us assume the contradiction: Suppose we find $x_i = x_j$ such that 0≤i,j≤m and $t_j - t_i <$ MTRT, but we cannot find the owner of token p, x_k, such that i≤k≤j. This means that the generation sequence numbers of the token when it arrives at x_i and the generation sequence number when it arrives at x_j is the same, because no station other than the owner of the token modifies the generation sequence number.

Also, x_j could not have been in the OFFRING state at any time since t_i, because x_i could not have been able to rejoin another ring after exiting a ring for one MTRT (or more precisely, exiting of the OFFRING state for one MTRT), by the implementation. Because a station is not allowed to receive a token when it is in the OFFRING state, it could not have received p before time t_i + MTRT. Thus, x_i could not have been in the OFFRING state since time t_i. Because of lemma 5.3.1, the priority of a station can only increase while it is not in the OFFRING state and thus, it does not make sense that token p could have survived station x_j. ■

Lemma 5.3.3 *Token p must have visited a station twice if it survives until time $t + MTRT$.*

We define MTRT to be the maximum time it takes for a token to visit a station twice if it survives, under our assumption of no transmission errors and no topological change. This cannot be longer than the amount of time for all stations to transmit in the graph, because the token must run out of stations that it can visit and choose among one of the stations that it has already visited to visit. ■

Lemma 5.3.4 *If no multiple equivalent tokens exist at time t, then no multiple equivalent tokens exist at time $s > t$.*

Suppose that there exist multiple equivalent tokens at time s when there were no multiple equivalent tokens at time t, such that s > t. Because of our assumption, it is impossible for a station to generate multiple equivalent tokens from transmission errors. Then a station must have generated a token when a token that it has previously generated is still in the graph. But the IDLE_TIME is greater than MTRT. And we know that a token dies when it doesn't see its owner for MTRT. Thus the station could not have generated the equivalent token when the token that it has previously generated is still in the graph. ■

Lemma 5.3.5 *No multiple equivalent tokens exist at time t + 2MTRT.*

All surviving equivalent tokens will go though the owner of the token in one MTRT as shown in 5.3.3. After one MTRT, the owner will remember the highest priority token among them. Within the next MTRT, all or all but one equivalent token will be deleted, because the owner will not pass any token that has a lower priority than the highest priority token that it received.

If the owner of the token leaves the ring at any time, all tokens will be deleted since the owner is not able to come back to a ring in less than one MTRT. ■

Lemma 5.3.6 *There exist a time s, such that $s < t$ + 2MTRT and no multiple equivalent tokens exist any time $u > s$.*

This directly follows from 5.3.4 and 5.3.5. From 5.3.5, we know that no multiple equivalent tokens exist at time t + MTRT. This means all multiple equivalent tokens must have been removed at time s before t + MTRT. From 5.3.4, no multiple equivalent tokens exist at u > s. ■

Lemma 5.3.7 *If station x has the bijection with its successor y, then the INRING_TIMER of x goes off before y.*

The fact that x has the bijection with y shows that the last token y accepted was from x. Then x must have reset its INRING_TIMER before y had. Thus, INRING_TIMER of y cannot go off before x. ■

Lemma 5.3.8 *When a station goes out of ring (into the OFFRING state), $|S|$, the number of the bijections, does not decrease.*

Let us say the predecessor of y is x, and the successor of y is z. When station y goes into the OFFRING state, it forms a ring of its own by the following assignment: y.NS = y and y.PS = y. We distinguish the two cases where a station can be kicked out. The first case is when the INRING_TIMER expires (Section 3.1). In this case, from the lemma 5.3.5, x could not have the bijection with y, because if it did, the INRING_TIMER of x would have gone off before y. In this case, regardless of whether y had bijection with z or not, $|S|$ will not decrease, because in the worst cases $|S|$ stays the same if y had the bijection with z. The second case is when y is kicked out because it is not successful in finding a successor. Again regardless of whether x had the bijection with y or not, $|S|$ will not decrease, because in the worst case we lose the bijection from x to y, but gains a self-bijection of y. ■

Lemma 5.3.9 *When a station accepts a token some time after t, $|S|$, the number of the bijections, does not decrease.*

After accepting a token, station y goes into the OFFRING state, attempts to pass the token to its successor, or sends the SOLICIT_SUCCESSOR token. According to lemma 5.3.8, $|S|$

is non-decreasing in the first case when the station involuntarily goes into the OFFRING state. So we are left to prove that $|S|$ is non-decreasing in the last two cases where station y attempts to pass the token to its successor, or sends the SOLICIT_SUCCESSOR token.

Let us see what happens if y decides to pass the token to its successor, z. If y has the bijection with z, then it will pass the token successfully and there will be no change in the network. In the case that y does not have the bijection with z, y will try to find a station to form the bijection with. If y is not successful within a certain window of time, it will go into the OFFRING state. And we have already shown in the lemma 5.3.8, that $|S|$ does not decrease. Now Let us consider the case where y successfully finds a station to form the bijection with. U is the station that y finally forms the bijection with. W is the predecessor of u, before y became its predecessor. Suppose w had the bijection with u, before y came along. Then $|S|$ is the same as before because we gained one bijection from x to u, but lost one from w and u. If there was no bijection from w to u to begin with, then, we would have gained on $|S|$ by one.

Now Let us see what happens if y decides to sends a SOLICIT_SUCCESSOR token. If no station wins the contention, then y will proceed to pass the token to its successor. And we have already shown in the previous paragraph that $|S|$ does not decrease. If station z wins the contention and successfully sends the SET_SUCCESSOR token, then y now has the bijection with z. Station z inherits the generation sequence number, the ring address, and the PS pointer from y, allowing successful establishment of the bijection with w the successor of y. If y did not have the bijection with w, then $|S|$ will likely stay the same. With any luck, $|S|$ will actually increase by one if w and z has the same ring address and w does not have a higher priority than z. Otherwise, the result will be the same as the case where a station tries to pass a token to a successor that it does not have the bijection with, as discussed in the previous paragraph. ∎

Lemma 5.3.10 *A ring will not break after time t + 2MTRT + INRING_TIME, and the number of stations in the ring will not decrease.*

A station updates its NS pointer when it is unable to pass the token to its successor, leaves the ring, or receives SET_SUCCESSOR. Because all stations in a ring have the bijections with its successor, it does not make sense that it is unable to pass the token to its successor. Thus, no station will be kicked out of the ring. Also, according to our assumption, a station will not leave the ring voluntarily. When a station receives the SET_SUCCESSOR, the NS pointer of the station will change. However, the ring will still not break since all contending stations must have a connection with the successor of the soliciting station, according to our assumption.

A station updates its PS pointer when it accepts a token from a station different from its predecessor. A station may accept the SET_PRED token from another station, if the station has the same ring address, causing the ring to break. This situation cannot possibly arise. A station in the ring could not possibly have received a token from a station in the same ring that is not its predecessor since all stations in the ring has the bijection with its successor and could not have failed to pass a token to its successor. Moreover, from our assumption, a station is not allowed to leave the ring voluntarily and induce its predecessor to generate SET_PRED.

Now we will show after t + MTRT + INRING_TIME, a station outside a ring cannot possibly send a SET_PRED token to a station inside the ring with the same ring address. Let us suppose that station y, with the ring address B, receives a token from station w, outside of the ring. Let us label the predecessor of station y as x. From time t + MTRT and on, no more multiple equivalent tokens exist according to lemma 5.3.6. A station cannot possibly remember about a token that did not exist at time t + MTRT, and at time t + MTRT + INRING_TIME, because a station must have accepted a token during [t + MTRT, t + MTRT + INRING_TIME] or have formed a self-ring. Thus, by t + MTRT + INRING_TIME, if a station remembers anything about the token with a particular ring address, B, they are remembering the same token. If a station receives a token from a station outside the ring, then the token must have made a loop from station y to w and back to y. This means that there was a breach in the ring that x and y belongs to, because when a token travels it makes bijection between the sender and the acceptor of the token. For station x and y to be in the same ring at time t + MTRT + INRING_TIME, a new token must have been regenerated by a station in the loop after the token with ring address B has passed them by. But this does not make sense because y must have received the token with the ring address B within MTRT since the last time it saw it, and all stations in the loop have seen the token after station y accepted it. ∎

Lemma 5.3.11 *$|S|$ monotonically increases and will converge to $|V|$ in a finite time.*

From lemma 5.3.8 and lemma 5.3.9, we have proven that $|S|$ is non-decreasing. We have also shown in lemma 5.3.10 that a ring will not break nor decrease in size after time t + MTRT + INRING_TIME. In addition a station will not solicit another station to join unless it sees two successful token rotations. If there are no topological change, a station will not join a node unless it is part of a ring. This means that the number of stations in a ring does not decrease.

If $|S| \neq |V|$ at some time s such that s > t + 2MTRT + INRING_TIME, then there exists station y that will not receive a NORMAL token from its predecessor at time t. If station y does not accept a token within the INRING_TIME since the last time it accepted a token, it will form a self-ring (Section 3.1). In this case, we gained a station that belongs to a ring. If the station y accepts a token, allowing another station to form the bijection with y, then we gain in the size of $|S|$. This is because the former predecessor did not have the bijection with y, and the new predecessor did not have the bijection with its former successor since it could not have generated SET_PRED token to form the bijection with y. Since within every INRING_TIME, the number of bijection that belongs to a ring, or the number of bijections increase, $|S|$ will converge to $|V|$ in finite time.

When $|S|$ finally converges to $|V|$ at time u > s, using the multiple token resolution lemma 5.3.9, we know that there exists one and only one token in all rings within u + IDLE_TIME + INRING_TIME. ■

Lemma 5.3.12 *If $|S| = |V|$ at time $s > t$, then within s + IDLE_TIME + 3MTRT, there exists one and only one token in any ring.*

There could be multiple tokens in a ring at time s. Either one and only one of these tokens will survive or none will survive by time s + 2MTRT. Station y in a ring will only accept a token if it has higher priority than the last token that it accepted. This means that after one revolution, the priority of all existing tokens must be increasing in terms of its order of visits to station y. All of these tokens must visit station y within another MTRT, and will be deleted but one token.

Even if all tokens get deleted at time u > s, within u + IDLE_TIME, there exists at least a one token in the ring. From the bijection, we know that if x has the bijection with y, y must accept tokens from x. This means that the station that holds the token has the higher priority than its successor. The only station that is an exception to this rule is the owner of the token, because the owner increments the generation sequence number by one when it passes the token. The only generation sequence number assignment that will satisfy these constraints is the following. The generation sequence number of the stations from the successor of the owner to the station to the station with the token has the same generation sequence number as the token. The generation sequence number from the successor of the station with the token to the owner is one less than that of the token. This means that when one or more stations regenerate token during [u, u + IDLE_TIME], the generation sequence number of these tokens will be higher than that of any stations at time s. Because these tokens will be passed around as a NORMAL token, only the highest priority token will survive within one MTRT, and lower priority tokens will be deleted. ■

5.4 Conclusion

For this proof to be practical, the assumptions must be reasonable. The assumption of this proof was that after a certain time t, transmission errors and the topological changes stop. One of the things that we can hope for when using this kind of assumption is that the algorithm reaches the correct state fast enough when the assumption holds. However, we found that the protocol, at worst case, can take time in the order of magnitude of MTRT. One can argue for the first assumption of fixed topology by supposing that the rate topology changes will probably be slow compare to the rate of transmission. But the assumption that there will be no transmission errors for the duration in magnitude of MTRT may not be valid when considering the fact that collision is a type of transmission error. This is especially true when there are multiple tokens in the ring, or when there are multiple rings that run on the same channel near one another.

According to this proof, the IDLE_TIME can be very large if we are unable to effectively put a bound on MTRT, because we have a constraint — IDLE_TIME > MTRT. A large IDLE_TIME can significantly degrade the performance of the network because there will be a long duration before the network regenerates the token in case of loss of token. One way to get around is to put an upper bound on the number of stations that can be in the graph. One solution based on this idea would be channel assignment based on geographical locations. For instance, automated highway system, adjoining sections in the freeway can be assigned distinct channels and thus limiting number of cars that can be in one token ring.

Chapter 6

Performance

6.1 Affective Bandwidth

The affective raw bandwidth was measured in the following way. Using the original WaveLAN driver, one Linux box pinged another machine with different sizes of ping packets. The ping implementation in the Linux kernel 2.2.14.5 uses an ICMP Echo Reply [19] packet. According to [19], the original packet that was pinged is copied and sent back in the acknowledgement. This is illustrated by Figure 6.1.

From the above observations, one can deduce that the difference between the round-trip times of the two different ping transmissions is twice the time taken to transmit the packet size difference.

$$R_1 = 2a + b + 2s \tag{6.1}$$

$$R_2 = 2a + b + 2S \tag{6.2}$$

$$R_2 - R_1 = 2(S - s) \tag{6.3}$$

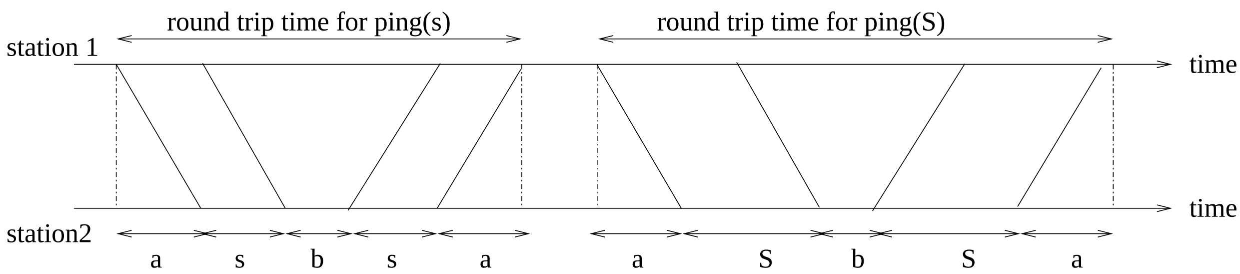

Figure 6.1: Ping Process

In the Equations 6.1, 6.2 and 6.3, we are assuming that a and b stays constant. The assumption is reasonable since we observed very consistent ping rotation times. We have done the ping experiment with various packet sizes from 108 bytes to 1008 at interval of 100 bytes. We observed a linear relationship between the round-trip time and the size of the ping packet. (Figure 6.2).

$$1.5135 * 10^6 = 2\frac{(908 - 208)bytes}{(12.3 - 4.9)ms}\frac{1000ms}{1sec}\frac{8bit}{1byte} \tag{6.4}$$

From Equation(6.4), we find that the raw bandwidth is 1.51M-bit/sec.

6.2 Observed Bandwidth

Using the original driver[18] that uses the IEEE802.11 directly, we were able to obtain 180 to 190 K-bytes/sec which translates to approximately 1.48 M-bits/sec transfer rate. This is measured by a FTP session of a large file (4M-bytes) from one machine to the next. On the other hand, in the token ring, the throughput of 140 K-bytes/sec translates into 1.12 M-bits/sec in a two node ring. This is a reduction in the affective bandwidth from 1.5 M-bits/sec. The reduction in the throughput is the result of added header, and a processing delay of the protocol. The issue of bandwidth will be further discussed in Section 7.1.

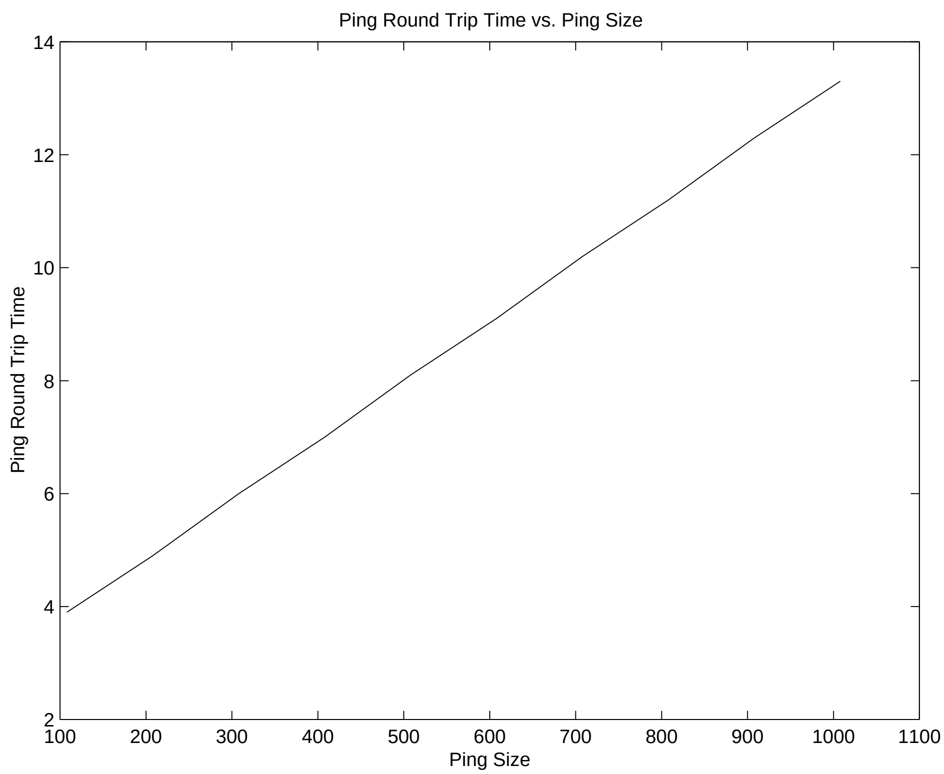

Figure 6.2: Measurement of Affective Bandwidth

6.3 Bound on Latency

In the trial shown in Figure 6.3, a large file size 5830080 Bytes was transferred using FTP between two nodes running the token ring medium access protocol. The token rotation time was measured at the FTP server for each token rotation. The file transfer of the same file was done three times with a single transfer, 2 concurrent transfers, and 3 concurrent transfers. The three peaks in Figure 6.3 correspond to the three FTP transmission periods.

From the figure, one can see that the token rotation time did not increase in spite of an increased number of simultaneous transmissions. Rather, the transfer took longer to complete in direct proportion to the number of concurrent FTP transfers. This bound on latency is the most salient property of the token ring protocol. It enables real time application support.

6.4 Fairness

In the simulation, only one queue was implemented at the MAC layer. Thus the bandwidth was equally divided among the receivers, and all three transmissions finished at approximately the same time. The throughput of each FTP connection was 140 K-bytes/sec, 72 K-bytes/sec, and 49K-bytes/sec during the first, the second and third transmission periods respectively. The test was duplicated with different files of size 7403520 bytes: 145 K-bytes/sec for a single transfer, 75 K-bytes/sec for two simultaneous transfers, and 51 K-bytes/sec for three simultaneous transfers.

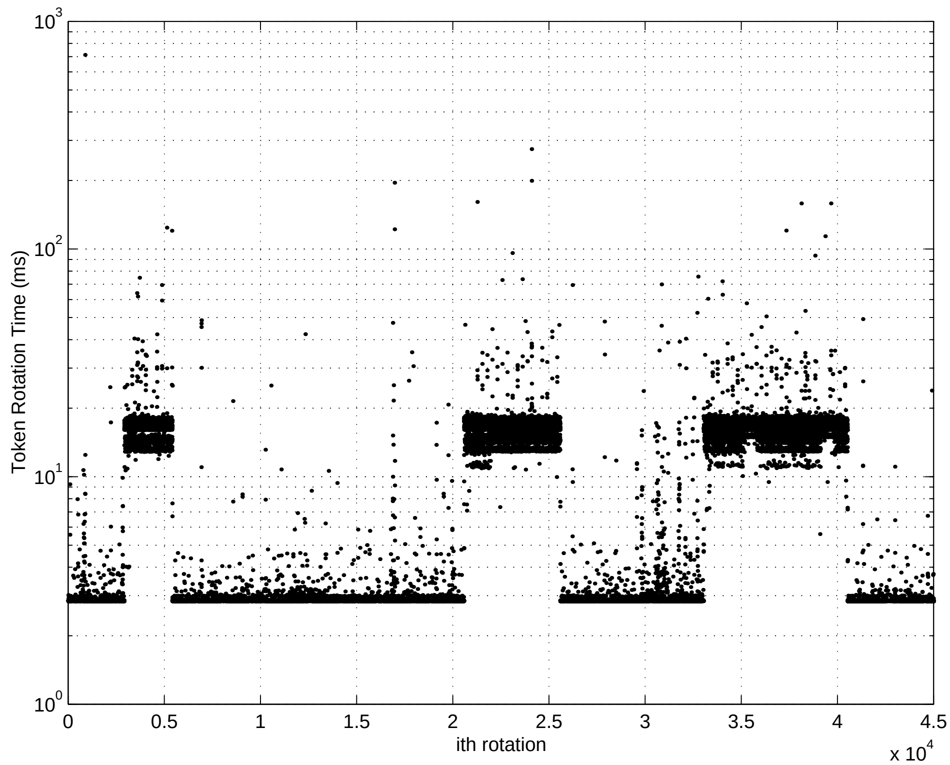

Figure 6.3: Token Rotation Time (Two Nodes, Hz=100)

6.5 Clock Resolution

The second trial was again conducted with two nodes and the same FTP transfers as the first trial but with different LINUX clock rates (Figure 6.4). The clock resolution was 100Hz and 2048Hz for the first (Figure 6.3) and second trials (Figure 6.4) respectively.

The lower clock resolution affects the speed at which the protocol can recover from failure. The failure recovery time can be estimated by observing the token rotation time. We assumed a token rotation time of greater than 20ms to be a failure. This is the token rotation time that is required by the Automated Highway Project[11]. We found that with the slow clock, the mean failure recovery time was 127.2917ms, while we observed 79.9529 ms with the fast clock. The worst case (677.07ms) was observed with the slow clock.

The Figure 6.6 shows the distribution of the token rotation time. We can see the increase in token rotation time as we go from two to three nodes.

6.6 Responsiveness

From Figure 6.4, one can see that the token rotation time is approximately 2 ms in a system of two nodes when there is no data transmission and approximately 18 ms during the FTP transmission. Since the data queue length at the MAC layer was set at 2 during the simulation, two 1500byte packet were transmitted in one token rotation. The responsiveness of the system can be varied by controlling the queue length at the MAC layer and controlling the token holding time.

The responsiveness of the system could be increased by reducing the latency caused by the token ring protocol computation, reducing the header size of the token ring protocol, or using a wireless network interface card with a higher bandwidth. To get an insight into

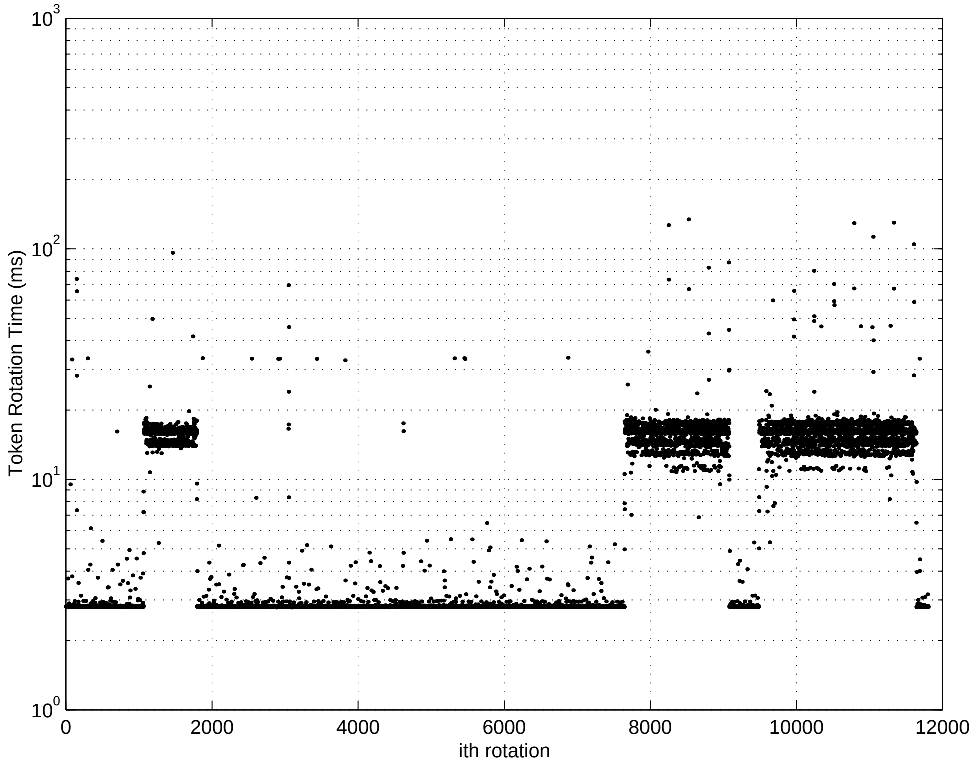

Figure 6.4: Token Rotation Time during FTP (Two Nodes, 2048Hz)

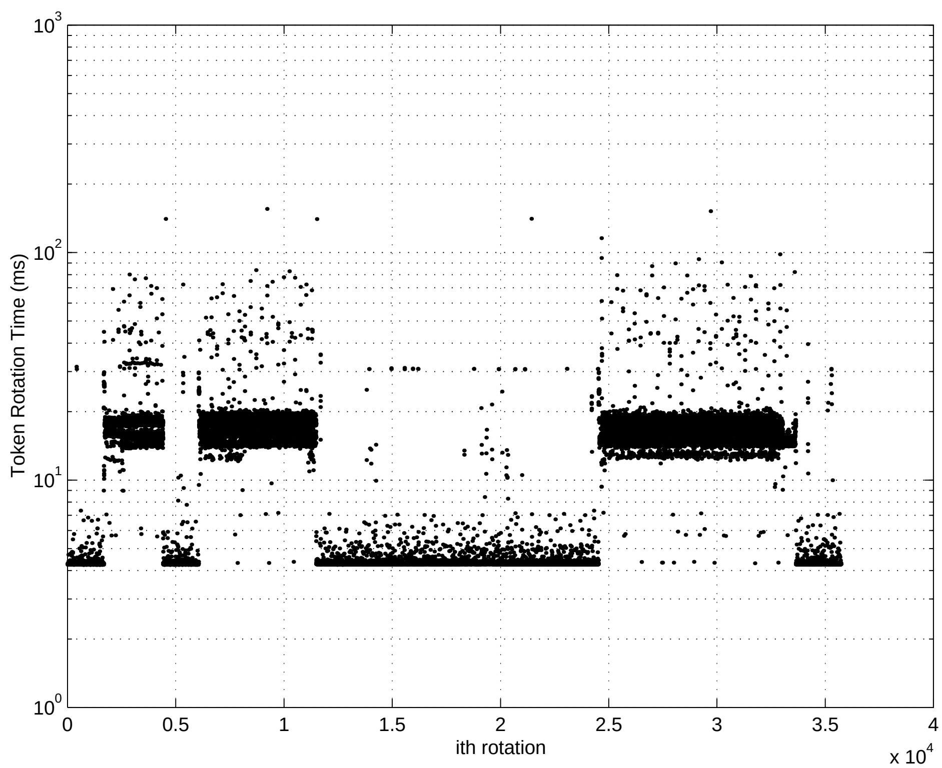

Figure 6.5: Token Rotation Time during FTP (Three Nodes, 2048Hz)

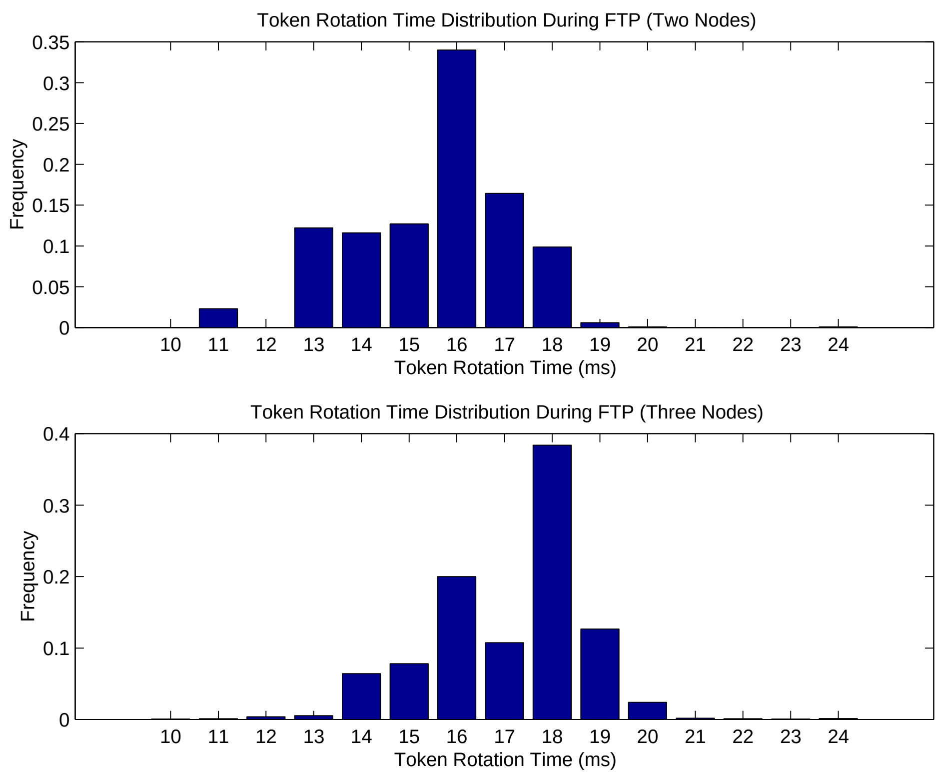

Figure 6.6: Token Rotation Time during FTP

the possible performance improvements, the system was tested with a higher bandwidth wireless network interface card. When tested with 2 M-bit/sec WaveLAN card (Figure 6.5) the token rotation time was approximately 3 ms in a system of three nodes. When tested with the 11 M-bit/sec WaveLAN card, the responsiveness of the system was increased to approximately 2 ms token rotation time when the system was not loaded.

6.7 Scalability

6.7.1 Responsiveness

The scalability of the responsiveness of the system can be measured by observing the increase in token rotation time as the number of the nodes in the ring is increased. As expected, the mean token rotation time increases linearly with the number of nodes in the ring as shown in Figure 6.7.

6.7.2 Variance

Another measure of scalability is the variance of the token rotation time. Figure 6.6 shows the variation in the token rotation time. Figure 6.5 shows the trial done with three nodes. A file of size 7403520 bytes was transferred. As with the above trial, FTP transfer was from one server node to one client node. For single FTP, a transfer rate of 130 K-byte/sec was achieved. In the second trial with two simultaneous transmissions, 69 K-bytes/sec was achieved. Lastly, the third transfer involved three concurrent transfers, achieving 47 K-bytes/sec. The LINUX clock resolution was set at 2048Hz.

Comparing Figure 6.5 to Figure 6.4, we can see that the variance of token rotation time was increased during the FTP transfers and during no transfers. However, the increase in the variance is ameliorated by the fact that token rotation time is well contained. To be

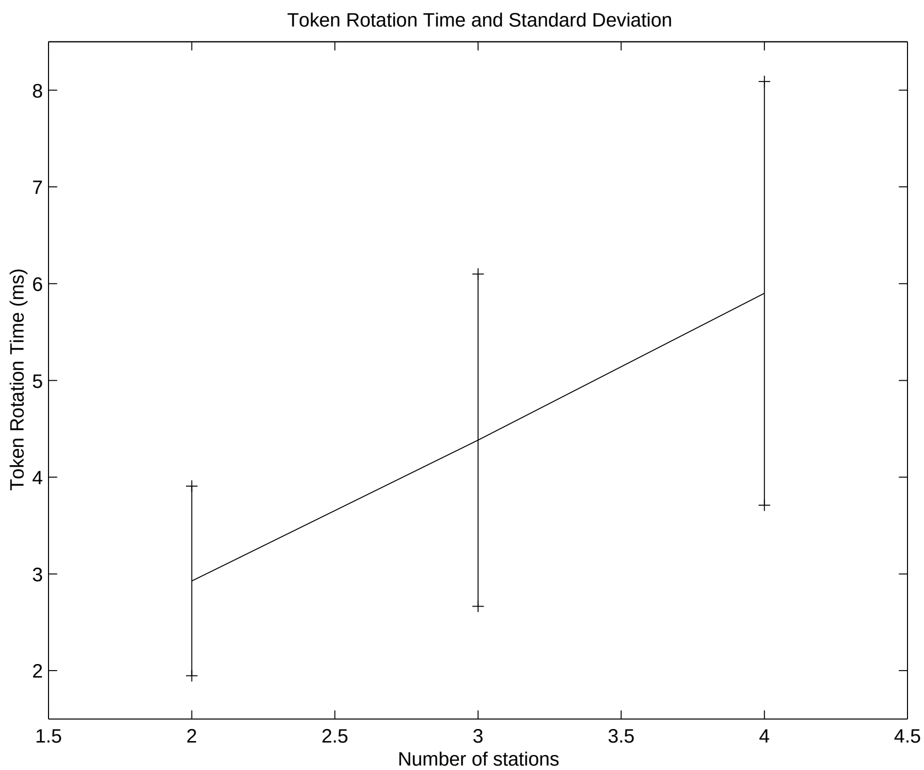

Figure 6.7: Mean Token Rotation Time

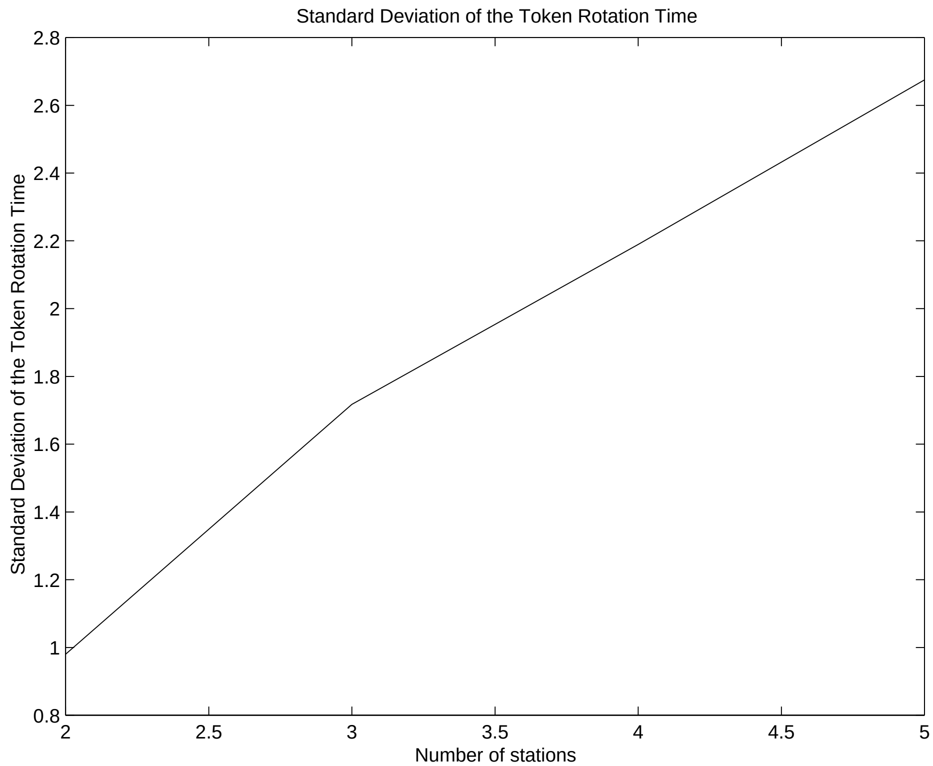

Figure 6.8: Standard Deviation of the Token Rotation Time

more precise, less than 0.01% of the time that the rotation time took longer than 40ms during the trial(0.0017% for Figure 6.5 and 0.0064856%. for Figure 6.4).

Figure 6.8 shows the variance of the token rotation time as the function of the number of nodes. Since the variation was dependent on the token loss rate, the trial was repeated several times to obtain a large number of samples. The result is promising since, at least with small number of node, the variance increased linearly.

6.8 Randomness of the Rotation Time

Access to the hard disk is a factor in the randomness of the rotation time. Access to hard disk occurs for many reasons including data logging of the token rotation time to data access from the FTP agents. The hard drive that was used in the testing platform (Dell

5000 Inspiron) has the following specifications.

Rotational Speed	4200rpm
Average Latency	7.1ms
Average Seek Time	13ms
Data Buffer	512 K-bytes [1]
Data Transfer Rate	108.8-202.9 M-bit/sec [2]

$$13.0ms + 7.1ms + 512Kbytes\frac{8bits}{bytes}\frac{Mbits}{1000Kbits}\frac{sec}{150Mbits}\frac{1000ms}{sec} = 47.4067ms \tag{6.5}$$

The above calculation is for an idealized case where hard disk buffer size (512 KB) is sufficient for entire transfer, and all memory required is contiguous. When starting a large application, the time takes for disk operation will be significantly higher than 47ms.

Chapter 7

Comparison with the IEEE802.11 in DCF mode

7.1 Performance Comparison

The token ring protocol in its current implementation is disadvantaged relative to the original IEEE802.11 driver because the token ring protocol is implemented on top of IEEE802.11 in DCF mode, incurring all the overhead that is associated with IEEE802.11 plus the overhead from the token ring. The overhead is the increased computation time and packet header size. In spite of these disadvantages, we find that under heavy load, the token ring implementation performs better than the IEEE802.11 in DCF mode. This is shown in Figure 7.1. In the figure, the aggregate FTP bandwidth is plotted against the number of simultaneous FTP transfers. Both cases involved five nodes. Regardless of the number of simultaneous transmissions, the ring was formed with five nodes. The FTP was done as follows. For the case of two simultaneous transfers, one transfers went from station 1 to station 2 another from station 2 to station 3. For the case of three simultaneous transfers,

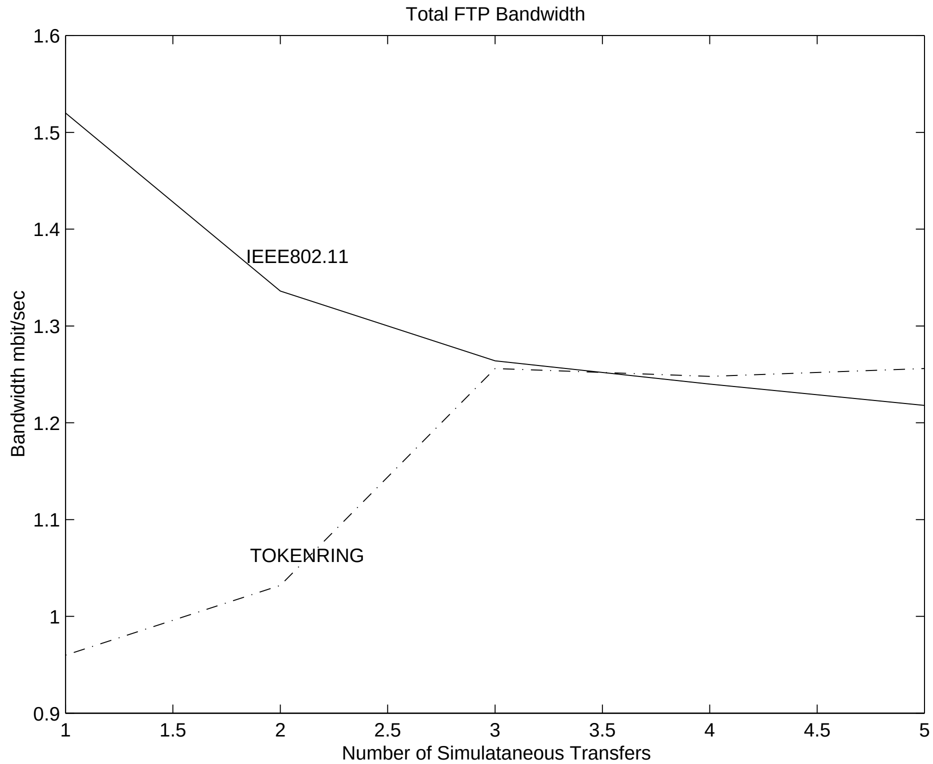

Figure 7.1: FTP performance (IEEE802.11 vs. Token ring)

1 to 2, 2 to 3, and 3 to 1.

In Figure 7.1, the solid line represents the IEEE802.11 in DCF mode and the dotted line represents the token ring protocol. At least with the number of nodes that were involved, we observed a decrease. The decrease in the throughput is expected since the number of collisions increase in a CSMA medium access control.

The performance surprisingly improves in the Wireless Token ring case when going from 1 to 3 simultaneous transfers. This can be explained as follows. Since for all trials in Figure 7.1, the ring size remained constant regardless of the number of simultaneous transfers, the number of token transmissions per token rotation remained constant in all trials. However, on increasing the number of simultaneous transfers, the number of data transmissions per token rotation is increased. This increases the ratio of data to token transmissions. This decreases the overhead per data bit. The need for retransmission of the data due to collisions is eliminated since there is no collision in the token ring implementation. We found that the token ring actually performs better when there are more than three simultaneous transfers.

7.2 Topological Comparison

The Centralized approach (802.11 PCF mode) uses the star topology, meaning that all slave nodes need to have a connection with the master. It is then easier to manage the network since all management information can be stored at the master. However, the approach is vulnerable to a single node failure at the master.

In addition, the star topology can be of a concern in an environment where flexibility in topology is required. As shown in Figure 7.2, the network that is required for the centralized approach is much more complex than the network that can be formed with the token ring. The network formed by the centralized approach requires two base stations that are connected with each other. The link between base station to base station may need to be on a different channel. In the case of the token ring, the network was formed by one ring, since the token ring only requires that a node is connected with its predecessor and successor.

In some cases, one can even achieve higher spatial reuse by utilizing the token ring, as shown in Figure 7.3, the vehicle platooning example[11]. In the platooning application, a network connects the platoon of vehicles. In our ring topology, one does not need to increase the transmission range even if the number of nodes in the ring increases[21]. However, in the centralized approach, one would be forced either to increase the transmission range of each radio to maintain the connection between master and all slaves, or to create multiple

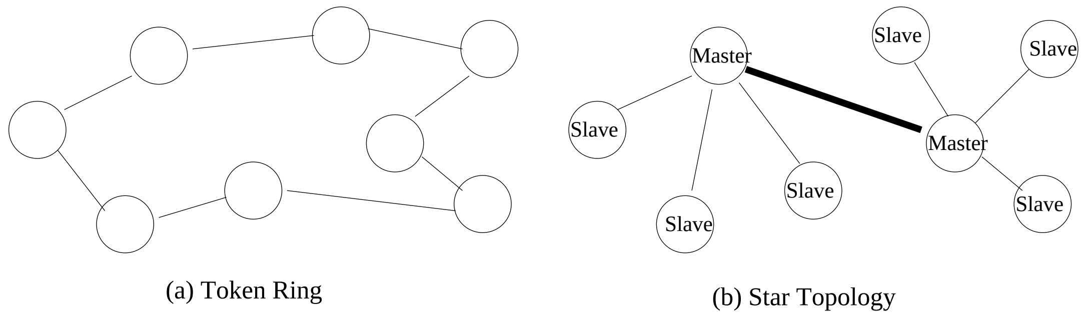

Figure 7.2: Topological Comparison

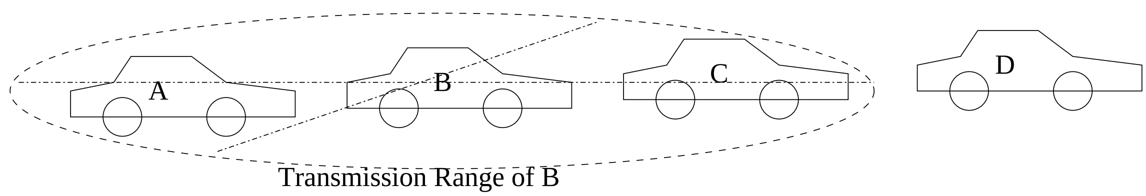

Figure 7.3: Vehicular Platoon

subnets and connect them together.

7.3 Working Together with Centralized Approach

At a particular level in the hierarchy of communication network, the centralized approach may make more sense than the distributed approach. The centralized and the distributed approach can be mixed in a hierarchy as shown in Figure 7.4.

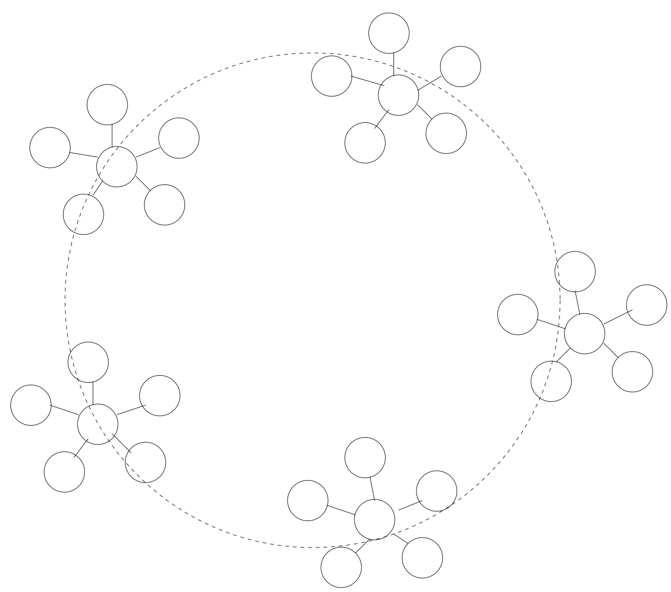

Figure 7.4: Hierarchically Mixed Structure

Chapter 8

Conclusion

The performance results from the Linux implementation of the token ring protocol is good. Even though the token ring protocol was implemented on top of the IEEE802.11 in DCF mode, incurring all the overhead that is associated with IEEE802.11 plus overhead introduced by the token ring protocol, we found that the token ring performs well or even better under heavy load. We expect that the advantage of WTRP over the IEEE802.11 in DCF mode would increase as the number of interfering nodes increases. A larger testing platform is currently being built to test this hypothesis. This shows that we have designed a protocol that is fast in terms of recovery (since there were several tokens lost during the tests) and efficient in terms of header size. One reason for the fast recovery is the use of a connectivity cache in each station. The performance results also show that we have a software implementation that is useful under a controlled application environment when utilized on top of an arbitrary network interface card.

The consistency of the token rotation time, regardless of the number of simultaneous transmissions is key to bounding the medium access latency. This perhaps is the most valuable feature of the wireless token ring protocol, since this is necessary in real time applications like intersection decision support, emergency management, and automated highway systems. In fact, the development of IEEE802.4 was initiated by people from General Motors and other companies interested in factory automation[22]. It has been established as a “mandatory protocol within the General Motor’s manufacturing automation protocol (MAP)”[23]. Features such as bounded latency and robustness against multiple node failures are some of the reasons for this choice. Our design bring the same bounded latency and robustness features to the wireless medium. Moreover, WTRP accomodates ad-hoc topologies.

Bibliography

[1] *http://wow.eecs.berkeley.edu*

[2] *International Standard ISO IEC8802-4:1990* — ANSI/IEEE Std. 802.4-1990

[3] K. Nichols, S. Blake, F. Baker, D. Black. *RFC2474 Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers.* December 1998.

[4] S. Herzog. *RFC2750 RSVP Extensions for Policy Control.* January 2000.

[5] *Draft International Standard ISO IEC 8802-11* — IEEE P802.11/D10, 14 January 1999

[6] *High Performance Radio Local Area Network (HIPERLAN), Type 1; Functional specification*, ETS 300 652, Radio Equipment and systems (RES) October 1996

[7] *http://www.bluetooth.com*

[8] Hannikainen, M.; Knuutila, J.; Letonsaari, A.; Hamalainen, T.; Jokela, J.; Ala-Laurila, J.; Saarinen, J. *TUTMAC: a medium access control protocol for a new multimedia wireless local area network.*, Ninth IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, New York, NY, USA: IEEE, 1998. p.592-6 vol.2. 3 vol. 1574 pp.

[9] Akyildiz, I.F.; McNair, J.; Martorell, L.C.; Puigjaner, R.; Yesha, Y. *Medium access control protocols for multimedia traffic in wireless networks.* IEEE Network, vol.13, (no.4), IEEE, July-Aug. 1999. p.39-47.

[10] Salles, R.; Gondim, P. *A new multiple access protocol for multimedia wireless networks.* ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, Monterey, CA, USA, 31 May-3 June 1998. p.538-41 vol.4.

[11] P.Varaiya. *Smart Cars on Smart Roads: Problems of Control. IEEE Transactions on Automatic Control*, 38(2):195-207, February 1993.

[12] H. Shim, T. J. Koo, F. Hoffmann, S. Sastry *A Comprehensive Study on Control Design of Autonomous Helicopter* In Proceedings of IEEE Conference on Decision and Control, Florida, December 1998

[13] Haas, Z.J. *On the performance of a medium access control scheme for the reconfigurable wireless networks. MILCOM 97. MILCOM 97 Proceedings, (vol.3), Monterey, CA, USA,* p.1558-64, 2-5 Nov. 1997

[14] Rubini, Alessandro. *Linux Device Drivers* O'Reilly & Associates, Inc. Sebastopol, CA. 1998.

[15] Pomerantz, Ori. *Linux Kernel Module Programming Guide*

[16] http://www.wavelan.com

[17] *The Teja Technical Reference* — Teja Technologies

[18] *WaveLAN/IEEE PCMCIA device driver for Linux (WVLAN49)* Lucent Technologies Inc, 1998-1999.

[19] J. Postel *Internet Control Message Protocol* RFC-792

[20] *IBM Travelstar 32GH, 30GT, and 20GN 2.5-inch hard disk drives* IBM TECHFAX #7012

[21] Chen, Chao *Leapfrog token ring : vehicle-to-vehicle communication protocol for the platoon environment* Master's Thesus at Berkeley, 1996

[22] A. Tnanebaum "Computer Networks" Prentice Hall, New Jersey, 1988

[23] W. McCoy "RFC1008: Implementation Guide for the ISO Transport Protocol" 1987