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

WTRP-Wireless Token Ring Protocol

Mustafa Ergen
University of California, Berkeley

California PATH Research Report
UCB-ITS-PRR-2002-29

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.

Report for TO 4224

September 2002

ISSN 1055-1425

WTRP-Wireless Token Ring Protocol

by

Mustafa Ergen

ergen@eecs.berkeley.edu

University of California Berkeley

May 2002

Abstract

WTRP (Wireless Token Ring Protocol) is a medium access control (MAC) protocol for wireless networks. The MAC protocol through which mobile stations can share a common broadcast channel is essential in wireless networks. In a IEEE 802.11 network, the contention among stations is not homogeneous due to the existence of hidden terminals, partially connected network topology, and random access. Consequently, quality of service (QoS) is not provided. WTRP supports guaranteed QoS in terms of bounded latency and reserved bandwidth which are crucial real time constraints of the applications. 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 use the channel for equal amount of time. The stations take turn to transmit and are 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. WTRP is robust against single node failure. WTRP recovers gracefully from multiple simultaneous faults. WTRP has applications to inter-access point coordination in ITS DSRC, safety-critical vehicle-to-vehicle networking, home networking and provides extensions to sensor networks and Mobile IP.

Contents

Chapter 1

Overview

1.1 Introduction

WTRP (Wireless Token Ring Protocol) is a medium-access-control (MAC) protocol for applications running on wireless ad-hoc networks that provide quality of service. In ad hoc networks, participating stations can join or leave at any moment in time. This implies a dynamic topology. The MAC protocol through which mobile stations can share a common broadcast channel is essential in an ad-hoc network. Due to the existence of hidden terminals and partially connected network topology, contention among stations in an ad-hoc network is not homogeneous. Some stations can suffer severe throughput degradation in access to the shared channel when load of the channel is high, which also results in unbounded medium access time for the stations. This challenge is addressed as quality of service (QoS) in a communication network.

In networks, QoS efforts have focused on network layer queuing and routing techniques [5],[6]. In an unreliable medium such as wireless, providing 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 [7] in PCF (Point Coordination Function) mode, the HiperLAN [18], and Bluetooth [19]

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.

The Wireless Token Ring Protocol (WTRP) discussed in this paper is a distributed medium access control protocol for ad-hoc networks. Its advantages are robustness against single node failure, and support for flexible topologies, in which nodes can be partially connected and not all nodes need to have a connection with a master. Current wireless distributed MAC protocols such as the IEEE 802.11 (Distributed Coordination Function (DCF) mode) and the ETSI HIPERLAN do not provide QoS guarantees that are required by some applications. In particular, medium is not shared fairly among stations and medium-access time can be arbitrarily long [12], [13].

As in the IEEE 802.4 [4] 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 WTRP. 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 priority 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.

WTRP has other desirable properties. It achieves high medium utilization since the collision probability is reduced by scheduling the transmission with token reception. WTRP distributes throughput in a flexible and fair manner among stations because each station in the ring takes a turn to transmit and forced to give up the right to transmit after a fixed time that results in bounds on medium-access time.

1.2 Applications

The wireless ad hoc network has many applications: Military, rescue missions, national security, commercial use, education, sensor networks, in which there is a need for rapid establishment of a communication infrastructure. Usefulness of the WTRP protocol is applicable in all these areas and some of applications are described here and in Chapter 12.1.

1.2.1 Unmanned Vehicles

One of the trends in transportation studies is the application of inter-vehicles communication. WTRP is to be deployed initially for University of California at Berkeley PATH Advanced Vehicle Safety Systems Program [10], the CALTRANS-PATH Demonstration 2002, and the Berkeley Aerobot project [15]. These applications impose stringent bandwidth, latency, and speed of failure recovery requirements on the medium access protocol. The platoon mode of the automated highway project involves up to 20 nodes in each platoon, and requires that information (approximately 100 bytes per vehicle for speed, acceleration, and coordination maneuvers) be transmitted periodically. WTRP meets the application requirements in terms of bounded delay and share of bandwidth to all stations in the network and fast failure recovery if there is a failure in a station.

1.2.2 Home Networking

Home networks link[1] many different electronic devices in a household through a connected local area network (LAN). Home networking allows all users in the household to access the Internet and applications at the same time. In addition data, audio, and video files can be swapped, and peripherals such as printers and scanners can be shared. There is no longer the need to have more than one Internet access point, printer, scanner, or in many cases, software packages.

In the future, the broadband "pipe" coming into a home will carry more than the Internet; it will also carry new telephone and entertainment services in the form of streaming video and audio and interactive networked games. Consequently, more devices that have real time constraints are expected to compete for the frequency band of wireless LAN as both number of users and type of wireless LAN devices increase in number. One observes that total aggregate throughput diminishes dramatically in IEEE 802.11 (DCF mode) with increase in number of concurrent transmissions due to collisions as shown in [2]. Apart from WTRP, other protocols take the centralized approach in home networking. Distributed property of WTRP provides partial connectivity in which all nodes need not to be connected with the access point. Partial connectivity, reduce transmission power and with packet forwarding of WTRP, connection is guaranteed for a node away from the range of the access point.

1.3 History of WTRP

WTRP was first designed for Automated highway project [10]. The first version was designed by Duke Lee in 1998. The second version was implemented with Teja software [23], released by Duke

[1]According to the Yankee group [27], 29 percent of U.S. households have multiple PCs. Many of these households also have high-speed (Broadband) Internet access (either cable modem or DSL). According to a recent report by ARS of La Jolla [28], there are now 5.1 million cable modem subscribers and 3.3 million DSL subscribers today. The total number is expected is grow to 28 million by 2004.

Lee, Roberto Attias , Dr.Stavros Tripakis, Dr.Anuj Puri, Dr.Raja Sengupta, and Prof.Pravin Varaiya in 2000 and reported in [2]. The last version is implemented in Linux Operating System with three deliverables by Duke Lee, Ruchira Datta, Jeff Ko, Dr.Anuj Puri, Dr.Raja Sengupta, Prof.Pravin Varaiya and by the author in August 2001 and published online[2] as an open source software.

In this version, compared to the second version, we simplified the Finite State Machine of the protocol. We introduced a distributed admission control procedure. Unlike the second version in which the module is implemented in application layer and hooked to the kernel, a kernel implementation is built as a Linux link layer module.

Besides the kernel implementation, we built a user-space implementation that works solely in the application layer as a platform-independent scheduler and a simulator that enables testing the protocol in a wireless environment.

1.4 Summary

In wireless medium, network layer modifications are insufficient to provide quality of service (QoS). QoS must also be addressed in link layer. Wireless Token Ring Protocol is a medium access protocol that provides QoS in terms of bounded latency and reserved bandwidth. WTRP overcomes the challenges introduced by ad hoc wireless medium through procedures for joining, leaving and failure recovery.

Applications of Intelligent Transportation Systems require QoS in terms of delay-bounds or fair share of the spectrum. This quality of service guarantee is critical in mesh stability of the formation of the vehicles. The communication of the speed and the velocity of the lead vehicle to all other vehicles in the formation had been shown to be sufficient for mesh stability of the system. In the future in a home, more devices are expected to compete for the frequency band of Wireless LAN.

[2] `http://wow.eecs.berkeley.edu/WTRP`

They require QoS in order to reduce collision and create connectivity.

WTRP has been developed by Berkeley Web over Wireless Group [1] since 1998 and the last version was distributed in 2001.

Chapter 2

Network Architecture

2.1 Introduction

Inspired by the IEEE 802.4 [4] standards, 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. When adapting a MAC protocol designed for wireline networks to the wireless ad hoc case, additional challenges are encountered in the wireless environment. The stations in the network are organized into multiple rings that can exists in proximity. Stations may not be fully connected that means not all nodes in the ring are directly connected. Radio range can be asymmetrical. In this chapter, we describe the design of WTRP to cope with these issues and outline the functions of each module, and discuss the context in which these modules are designed.

2.2 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 2.1. In addition to the communication stack including the Datalink Layer where WTRP will be located, we need Mobility Manager, Channel Allocator,

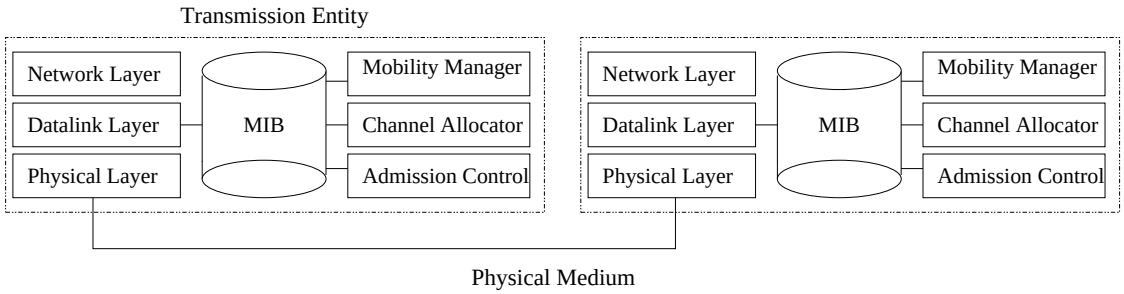

Figure 2.1: System Architecture

Management Information Base (MIB), and Admission Control Manager. We assume that multiple channels are available, and that different rings are on different channels. Different rings are assigned to different channels by a channel allocator (Section 2.2.2).

2.2.1 Medium Access Control

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

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

1. Ensuring that each ring has a 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 3 and Section 4.

2.2.2 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 a core idea of wireless cellular telephony. 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 (by reducing the transmission power) increases the capacity and decrease equipment costs. However, dividing the nodes into multiple rings will reduce the number of nodes in a ring, and thereby increase routing overhead between rings.

Finding the globally optimal solution for channel allocation, an allocation that maximizes the capacity of the network, 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. 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. And this is the method that is being considered 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 distributed manner using the information collected from

the token structure, which contains the number of nodes (NoN) in the ring. If NoN reaches the maximum value, this is an indication for the nodes out of the ring to shift to the next channel and search for another ring.

2.2.3 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 3.

2.2.4 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. MAX_NoN is the maximum number of node (NoN) that is allowed in the ring. The Admission Control Manager has to ensure the inequality: $RESV\_MTRT < MAX\_MTRT$ and $NoN < MAX\_NoN$. Only if these inequalities are satisfied, 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 available in the ring may join.

2.2.5 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.

2.2.6 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.

2.3 Summary

In order to manage the wireless medium, WTRP operates in conjunction with several modules. Medium Access Control is where WTRP is located, Channel Allocator deals with channel assignment of rings in order to avoid interference, Mobility Manager performs hand over in case of movement. Admission Control manages the ring size and invites if there is room. Policer monitors the traffic and perform congestion control, Management Information Base is the place where the ring parameters is stored.

Chapter 3

Protocol Overview

3.1 Definitions

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

2. The term “frame” refers to the sequence of bits that is passed to the physical layer for one packet. A “frame” does not include the preamble, 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 successor and predecessor fields of a node are correct. It is more precisely defined in Section 10.1.

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

9. The Channel Allocator, Mobility Manager, and Admission Control Manager introduced in Section 2.2 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.

11. NoN refers to the Number of Nodes in the ring.

3.2 Observations

1. Not all stations need to be involved in token passing. Only those stations which desire to initiate 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.

3.3 Description

In 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.

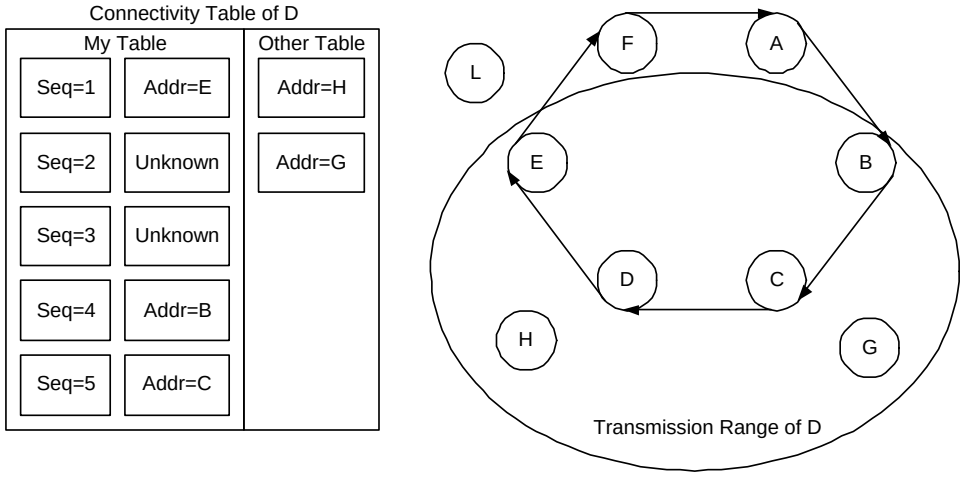

Figure 3.1: Connectivity Table

FC	RA	DA	SA	NoN	GenSeq	Seq
1	6	6	6	2	4	4 bytes

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 number of nodes (NoN) in the ring is represented in the token frame and calculated by taking the difference of sequence numbers in one rotation.

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 local list of stations in its own ring and an unordered global list of stations outside its ring (See Figure 3.1). In Figure 3.1, station D monitors the successive

token transmission from E to F before the token comes back to D. At time 0, D transmits the token with sequence number 0, at time 1, E transmits the token with the sequence number 1, and so on. D will not hear the transmission from F and A, but when it hears transmission from B, D 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 E and B.

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 node 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 successing 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

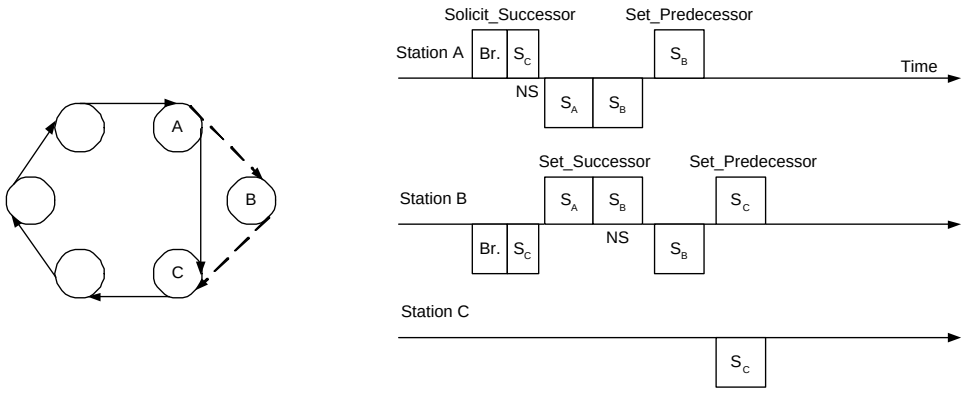

Figure 3.2: Joining

token rotation provided no more tokens are being generated (See Section 10.1).

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 forming the ring again. The strategy taken by the WTRP is to try to reform the ring by excluding as few 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 3.2, suppose station B wants to join the ring. Let us also say that the admission control manager on station A broadcasts (Br.) other nodes to join the ring by sending out a SOLICIT_SUCCESSOR that includes successor(C) of A. 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, such as B that wants to join the ring, hears a SOLICIT_SUCCESSOR token, it

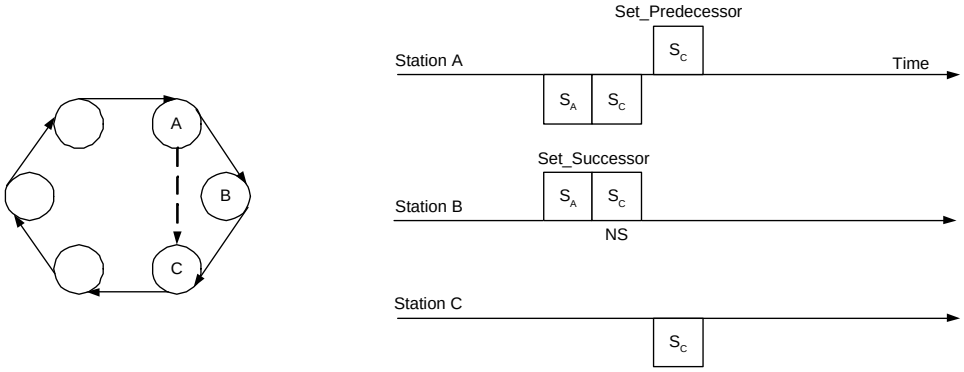

Figure 3.3: Exiting

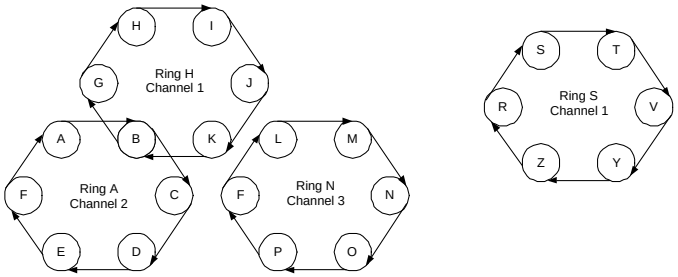

Figure 3.4: Multiple Rings

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

As shown in Figure 3.3, suppose station B wants to leave the ring. First, B waits for the right to transmit. Upon receipt of the right to transmit, B sends the SET_SUCCESSOR packet to its predecessor A with the MAC address of its successor, C. If A can hear C, A tries to connect with C by sending a SET_PREDECESSOR token. If A cannot hear C, A will find the next connected node, in the transmission order, and send it the SET_PREDECESSOR token.

Interference is eliminated by including NoN in the token packet. When a station detects a ring, it

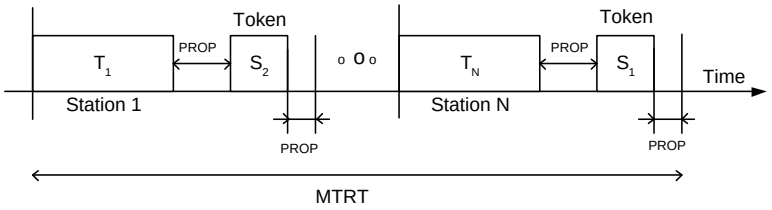

Figure 3.5: Timing diagram for WTRP

examines the NoN value in the token. If NoN is set to maximum, the station changes its channel and searches for another ring. Otherwise, the station either waits to become a ring member or changes its channel to search for another ring. If the station waits, it suspends transmission and waits for a SOLICIT_SUCCESSOR token. As a result, a newcomer station never interferes with the ring.

In Figure 3.4, we can see that the ring address of a ring is the address of one of the stations in the ring, which 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. Multiple ring management is left open and cited briefly in Chapter 12.1. There are possible schemes where a station can belong to more than one ring or a station may listen to more than one ring.

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.

The transmission proceeds in one direction along the ring. We use Figure 3.5 to analyze the protocol. Assume that there are N nodes on a token ring. We define T_n to be the time during which

node n transmits when it gets the token, before it releases the token. Thus, T_n can range from 0 to THT. A station first sends its data during T and if there is enough time left, the station decides to send SOLICIT_SUCCESSOR. PROP stands for propagation time of a signal in the medium.

3.4 Summary

Wireless Token Ring Protocol manages the medium by creating multiple rings. Rings are identified by an ID that is the MAC address of a node in the ring. Each station has a successor and predecessor. When a station gets the token from its predecessor, it transmits for a fixed time and passes the token to its successor. After transmitting, the station looks for an implicit acknowledgement that is a transition from its successor to the successor of its successor. Stations monitor token transitions and create a connectivity table, an ordered list of stations in their ring. A token has a priority which increases each time it is transmitted. If a station gets two different tokens, it chooses the higher priority one and notifies its predecessor.

Each station monitors the token rotation time and the number of nodes in the ring and send the invitation to the nodes outside if there is room. The joining process is a handshake mechanism between the inviting station, joining station and the successor of the inviting station. When a station leaves the ring willingly, it notifies its successor to its predecessor and predecessor tries to connect to the successor of the station and if it is unsuccessful, the predecessor closes the ring with the next station in its connectivity table. If there is a failure in a station, its predecessor detects it when passing the token and tries to connect with the next station in its connectivity table.

Chapter 4

Specification

4.1 Introduction

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 3. The frame formats are also defined in detail.

4.2 MIB Parameters

Processing_Time (PT) This time represents the time it takes for a station to process a token. More precisely, it is the delay between the end of reception of a token to beginning of data or token transmission in reaction to the token reception.

Max_Token_Holding_Time (MTHT) This is the maximum amount of time one station can hold the token before passing it. A station can only transmit data packets when it is holding the token. After this amount of time has passed, it must yield the token even if it has not transmitted all its data, so other stations also have a chance to transmit. It must be set appropriately to achieve

good bandwidth and latency: each station should be able to transmit a significant amount of data while it has the token, and no station should have to wait too long before receiving the token.

Max_Token_Rotation_Time (MTRT) This is the maximum amount of time the token should take to travel around the ring and return to the station. This determines the latency of the data transmission. This time is enforced by keeping too many nodes from joining the ring.

Max_Num_Token_Pass_Try (MTPT) This is the maximum number of times the station should try to pass the token before concluding that it is badly connected to the rest of the ring and leaving the ring.

Transmission_Rate (TR) This is the data transmission rate of the connection (measured in bytes per jiffy in kernel implementation (Chapter 9)).

Solicit_Successor_Prob (SSP) This is the probability in percent that the station will solicit a successor (i.e., look for new nodes to join the ring) at any given opportunity. This is only relevant if there is room in the ring, i.e., neither the maximum number of nodes nor the maximum token rotation time has been reached.

4.3 Timers

4.3.1 Definitions

1. Each timer value is randomized before it is assigned. $R(.)$ represents the randomize function.
2. TICKS_PER_SEC represents ticks per second. In the kernel implementation, *HZ* value is considered (Chapter 9).
3. BANDWIDTH(BW) defined as $TR/TICKS\_PER\_SEC$.

4. MAX_TOKEN_PASS_TIME (MTPT) defined as $R(2(MTHT) + 2(PT))$.

5. TOKEN_ROTATION_TIME (TRT) defined as $max((NUM\_NODE * MTPT), MTRT)$.

6. MAX_JOINING_TIME[1] defined as the maximum time that takes for a joining process.

7. MAX_RESPONSE_TIME[2] defined as maximum time to get a response from the soliciting station in the joining process.

4.3.2 Timer Types

Idle_Timer is set to the $R(TRT)$ and starts to count down. It is reset whenever the station receives an implicit acknowledgement. When the *idle_timer* expires, the station calls *idle_timer_handler*.

Inring_Timer is set to the $R(TRT + IDLE\_TIME)$ and starts to count down whenever the station receives a token. When the *inring_timer* expires, the station assumes that it has been kicked out of the ring, and calls *inring_timer_handler* to exit the ring.

Token_Pass_Timer is set to the $R(2(MTHT) + 2(PT))$ whenever a station sends a token and starts to count down. When the timer expires without the station receiving an implicit acknowledgement of the transmission, it assumes that the transmission was unsuccessful and calls *token_pass_timer_handler* to retransmit the *token*.

Token_Holding_Timer is set to the $MTHT$ and the station can transmit data until the timer expires and *token_holding_timer_handler* is called to pass the token.

Offline_Timer is set to the $R(TRT + IDLE\_TIME)$ and starts to count down when the station goes to the *offline* state. *Offline_timer_handler* is called when the timer expires.

---

[1] $R(3 * PT + \frac{SOLICIT\_SUCCESSOR\_SIZE + SET\_SUCCESSOR\_SIZE + SET\_PREDECESSOR\_SIZE + 3 * PHY\_HEADER\_SIZE}{BW})$

[2] $R(PT + \frac{SET\_SUCCESSOR\_SIZE + SET\_PREDECESSOR\_SIZE + 2 * PHY\_HEADER\_SIZE}{BW})$

Claim_Token_Timer is set to the $R(TRT + TOKEN\_PASS\_TIME)$ and counts down at the *floating* state. When expired, the station calls *claim_token_timer_handler* to make a *self_ring*.

Solicit_Successor_Timer is set to MAX_JOINING_TIME and when expired, the station calls *solicit_successor_handler* to send *solicit_successor* token.

Solicit_Wait_Timer is set to $R(TOKEN\_PASS\_TIME)$ and when expired, the station calls *solicit_wait_handler* to pass the *token*.

Contention_Timer is set to the MAX_RESPONSE_TIME. When the invited station does not receive *set_predecessor* from the soliciting node for CONTENTION_TIME, it assumes that it has lost the contention process and calls *contention_timer_handler* to go back to *floating* state.

The timer values are arranged so that following relationship holds:

1. TOKEN_HOLDING_TIME $<$ IDLE_TIME $<$ INRING_TIME
2. MTRT[3] $\leq$ IDLE_TIME

4.3.3 Timer Handlers

Idle_Timer_Handler Station sends *claim_token* and *token* and goes to *monitoring* state.

Inring_Timer_Handler Station goes to *floating* state.

Offline_Timer_Handler Station goes to *floating* state.

Claim_Token_Handler Station makes *self_ring*, sends *claim_token* and goes to *idle* state.

Solicit_Successor_Handler Station sends *solicit_successor_token* and goes to *soliciting* state.

[3]MTRT is the Maximum Token Rotation Time defined more precisely in Section 4.2 & 10.1

Token_Holding_Timer_Handler Station passes the *token* to its successor and goes to *monitoring* state.

Solicit_Wait_Handler Station goes to the *idle* state if it is *self_ring*, otherwise, sends *token* and goes to *monitoring* state.

Contention_Timer_Handler Station goes to *floating* state.

Token_Pass_Timer_Handler Station retransmits token for *num_token_pass_try* times, if it is unsuccessful, the station sends *set_predecessor_token* to close the ring and goes *offline* state.

4.4 Frame Formats

4.4.1 Frame Control Field

1. Control Frame

0	0	C	C	C	C	C	C

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
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. Transparent Data Frame

   This is the 802.11 DCF Packet Format. This is an option in the kernel implementation. With this option WTRP and 802.11 network can communicate with each other.

4.4.2 Sequence Control Fields

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}.

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) passes the token, it 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.

4.4.3 Address Fields

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

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

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

4.4.4 Admission Control Field

Number of Nodes (NoN) When the token rotates the owner calculates the number of nodes in the ring by taking the difference between the *Seq* of the token and *Seq* of the node.

4.4.5 Queues

Data Packet Queue Data packets coming from the upper layer are stored and transmitted when the station gets the token.

Control Packet Queue Control packets if there is a failure in its transmission are stored and re-transmitted.

4.4.6 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.

4.4.7 Frame Types

1. Token

0 0 0 0 0 0 0 0	RA	DA	SA	NoN	GenSeq	Seq
1	6	6	6	2	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	Br.	SA
1	6	6	6 bytes

The Claim Token is broadcast(Br.) 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	Br.	SA	NoN	NS
1	6	6	6	6	6 bytes

The *solicit_successor* token updates the successor field of a station. It is broadcast for inviting another nodes to join the ring. NS field is to inform the joining node about its prospective successor.

4. Set Predecessor Token

0 0 0 0 0 0 1 1	RA	DA	SA	NoN	GenSeq	Seq
1	6	6	6	2	4	4 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

8. Transparent Data

FC	Duration	DA	SA	BSSID	SC	N/A	Data	CRC
2	2	6	6	6	2	6	0-2312	4 bytes

- FC - Frame Control:protocol version and frame type
- Duration - Time reserved for packet transmission
- BSSID - Basic Service Set ID

- SC - Sequence Control
- CRC -Cyclic Redundancy Check

4.5 Summary

WTRP is implemented as a Finite State Machine and each state has one or more timers. Timer values are calculated from the MIB parameters. When it passes to a new state, a station initializes the timers of the state and the handler function of the timer is called, when a timer expires. Tokens have a three-dimensional priority; sequence number that is incremented in each passing, generation sequence number that is incremented in each rotation by the owner, and ring address. In addition to the token signal, WTRP introduces additional control signals involved in joining, leaving, and failure recovery processes.

Chapter 5

Finite State Machine

5.1 Introduction

In this chapter, we describe the finite state machine in detail. Figure 8.1 shows the finite state machine (FSM) of WTRP. The states are { *Beginning, Floating, Offline, Joining, Soliciting, Idle, Monitoring, Have Token*}. To cope with "mobility", FSM has *joining* and *soliciting* state where inviting and joining processes are handled, "Interference avoidance" to other rings is done by introducing *floating* and *offline* states, whereby a station suspends transmission in these states and waits to join a ring. "Collision avoidance" in the same ring is eliminated by the *idle* state, whereby a station suspends transmission until it gets the *token*. "Equal bandwidth share" is controlled by *have token* state whereby the station transmits packets as long as it is allowed. *Monitoring* state is for "guaranteed transmission", whereby a station checks its transmission and retransmits in case of a failure. In the rest of the chapter, we describe the states and states transitions in detail.

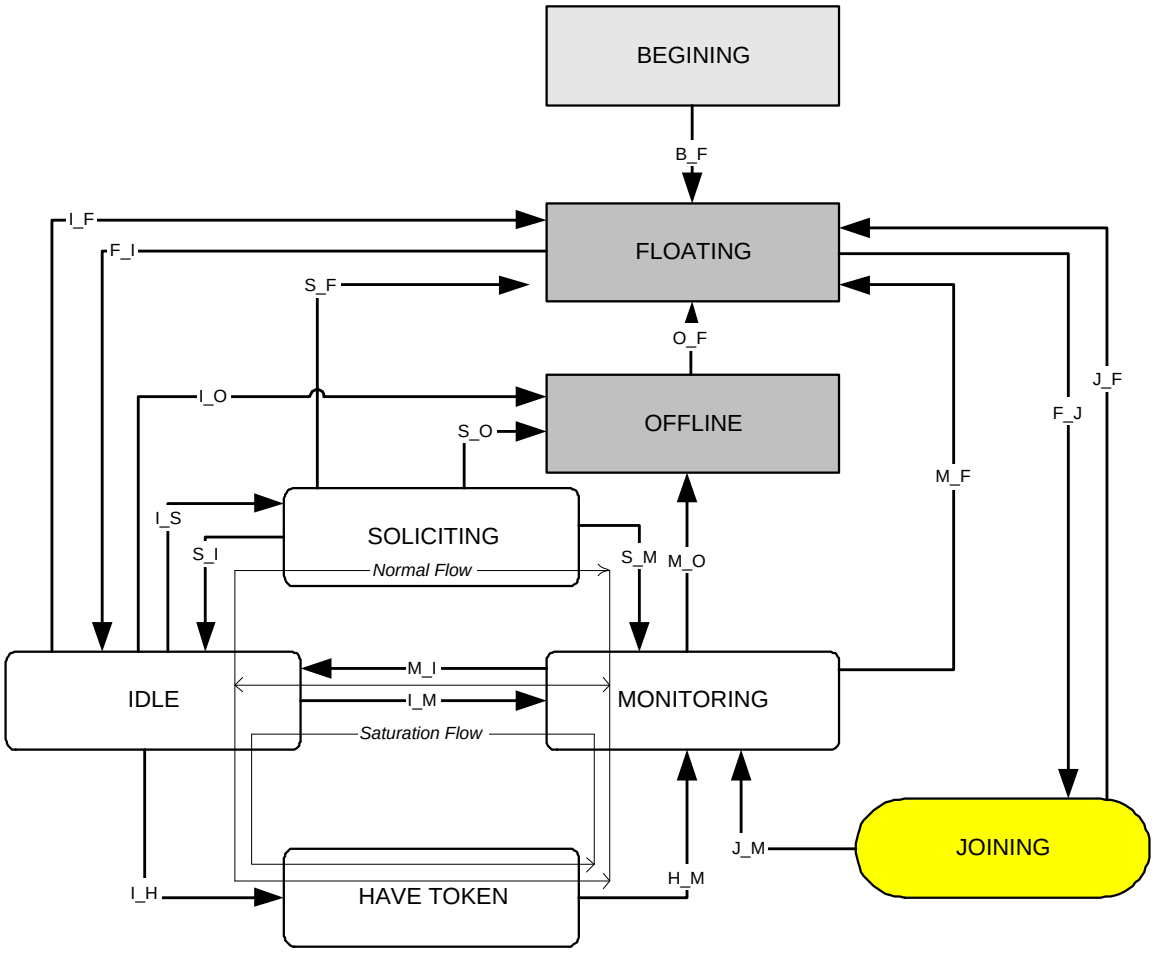

Figure 5.1: Main Flow

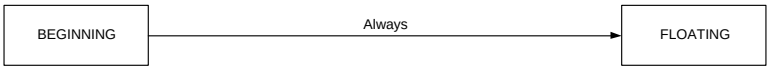

Figure 5.2: Transition From/To Beginning State

5.2 States

5.2.1 *Beginning State*

Beginning state is a virtual state that represents the starting of the protocol. There is only one transition as seen in Figure 5.2 and the station directly goes to the *floating* state.

5.2.2 *Floating State*

Floating state is the state where a station resets its parameters and waits to join a ring. When a station passes to the *floating* state, it resets its station parameters, cleans up its packet queues, and initializes the *claim_token_timer*.

A station passes to the floating state at the beginning and when there is a failure in the ring. When the station is *self_ring* (When the successor and predecessor of the station is itself and station is in *idle* state.) and detects another ring, it goes to *floating* state. If the station does not get the token in the *idle* state or can not join to a ring, *idle_timer* expires and it goes to *floating* state. If the station detects a ring and is in the *floating* state, it waits to be invited from the ring and suspends its transmission in order not to interfere the ring transmission. If the station does not detect a ring , *claim_token_timer* expires and it goes to the *idle* state and creates a *self_ring*. If the station gets a *solicit_successor* token, and if it wants to join a ring, the station sends a *set_successor* token and goes to *joining* state. The state transitions from or to the *floating* state can be seen in Figure 5.3.

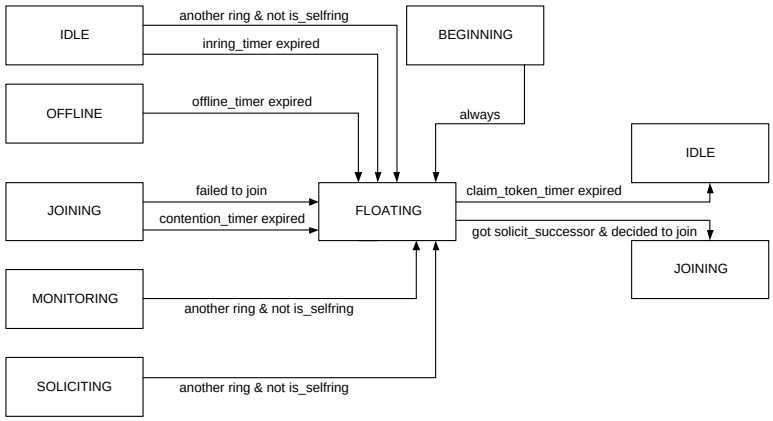

Figure 5.3: Transition From/To Floating State

5.2.3 *Offline State*

When a station goes to the *offline* state, it initializes the station, clears the packet queues, and adds *offline_timer* to the scheduler. Since *offline_timer* is twice the *max_token_rotation_time*, the wait period in the *offline* state gives sufficient time to the former ring members to realize that the station is out of the ring. This prevents the station from joining a ring before the ring is closed. A station goes to the *offline* state if it belongs to a ring other than *self_ring* when it detects another ring or when a station joins a ring but fails to pass the token to its successor. In the *offline* state, a station waits and does nothing until the *offline_timer* expires. From the *offline* state, a station only goes to *floating* state. The state transitions from or to the *floating* state can be seen in Figure 5.4.

5.2.4 *Joining State*

When a station goes to the *joining* state, it initializes the *contention_timer*. A station goes to *joining* state only from *floating* state. When a station receives *solicit_successor* and decides to join the ring, the station sends *set_successor* and goes to the *joining* state. If the station receives *set_predecessor*, this means that joining is successful. Therefore, the station sends *set_predecessor* and passes to the *monitoring* state. If the station does not get *set_predecessor*, *contention_timer* expires and this

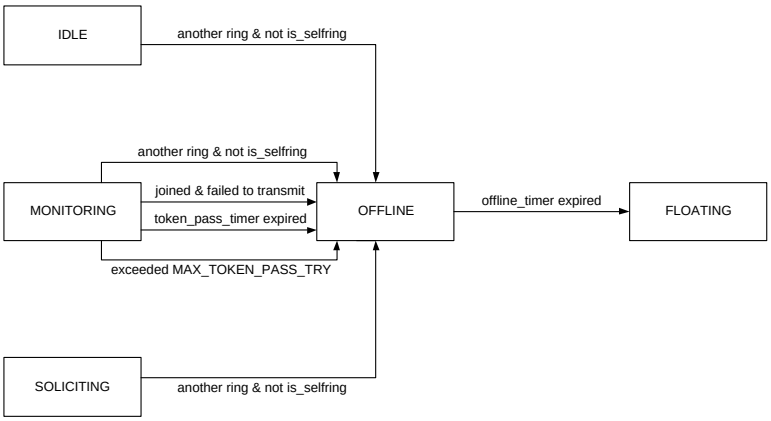

Figure 5.4: Transition From/To Offline State

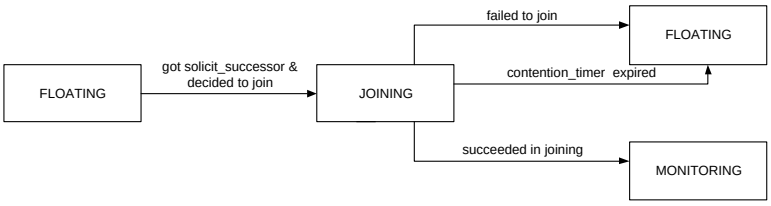

Figure 5.5: Transition From/To Joining State

means a failure in joining and the station goes back to the *floating* state as seen in Figure 5.5.

5.2.5 *Soliciting State*

When a station goes to the *soliciting* state, the station initializes *solicit_wait_timer* and sets the *station->num_node*[1] to *MAX_NoN+1* in order to suspend transmission of other stations if it is not a *self_ring*. When the station is in the *idle* state and receives the *token*, it checks its queues and if they are empty, it decides to send solicit successor. If the decision is positive, then it sends *solicit_successor* token and goes to the *soliciting* state. If the station is *self_ring*, it adds *solicit_successor_timer* and when it expires, it sends *solicit_successor* token and goes to the soliciting state.

[1] *station* structure is defined more precisely in Section 6.4

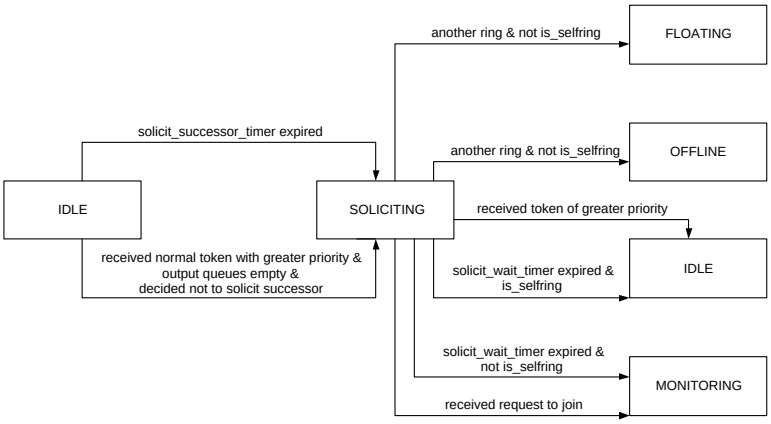

Figure 5.6: Transition From/To Soliciting State

If a station receives *set_successor* in the *soliciting* state, this means that there is a station responding to its *solicit_successor*, then the station sends *set_predecessor* and goes to the *monitoring* state. If there is no response to the invitation, *solicit_wait_timer* expires and the station goes to *idle* state if it is *self_ring* and *monitoring* state if it is not *self_ring* as seen in Figure 5.6.

5.2.6 *Idle State*

When the station goes to *idle_state*, it adds *idle_timer* and *inring_timer* if it is not a *self_ring* , otherwise it adds only *solicit_successor_timer*. When the station passes the *token* to its *successor*, it goes to the *monitoring* state to listen for *implicit_ack* that is a transmission from its successor to the successor of its successor. After hearing the *implicit_ack*, the station goes back to the *idle* state. When the station gets the *token* with greater priority (lower priority tokens are deleted by sending *token_deleted* token) and packet queues are nonempty, station goes to *have_token* state. If the packet queues are empty, then it decides whether to send *solicit_successor* or not; if the decision is positive, it goes to *soliciting* state, otherwise, the station passes the *token* and goes to *monitoring* state. If the station receives *set_successor* token that means a station is leaving , the station sends *set_predecessor* and goes to the *monitoring* state. If the station wants to leave, it sends *set_successor*

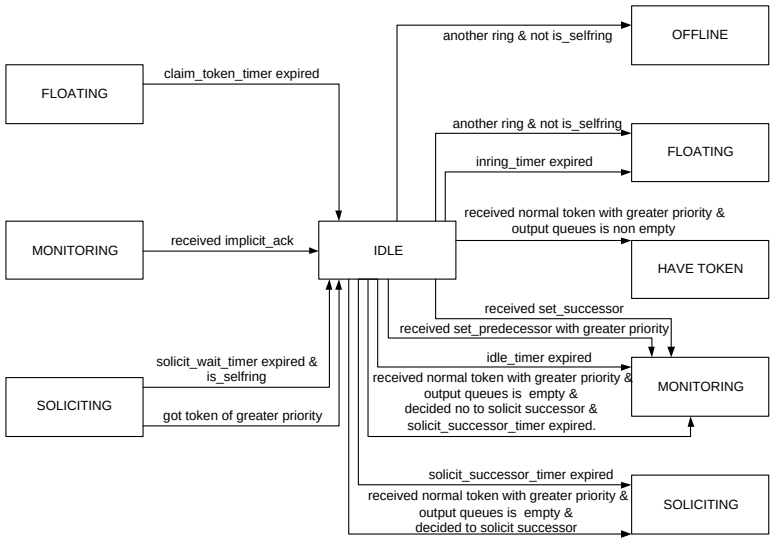

Figure 5.7: Transition From/To Idle State

and goes to *offline* state. The detailed state transition of *idle* state can be seen in Figure 5.7.

5.2.7 *Monitoring State*

When the station goes to the *monitoring* state, it resets *num_token_pass_try* and adds *token_pass_timer*. *Monitoring* state is to monitor the medium in order to make sure that the transmission is successful. *Implicit_ack* is used to decide whether a transmission is successful or not. If there is a retransmission of the same token, the station sends *token_deleted* token. Figure 5.8 shows the transition to and from *monitoring* state. If the transmission is successful , the station goes to *idle* state.

5.2.8 *Have Token State*

When the station goes to the *have_token* state, it initializes *token_holding_timer*. A station passes to *have_token* state only from the *idle* state if it has packet to send. When a packet is transmitted *tx_done_handler* is called. *Tx_done_handler* checks for the packet queues and transmits if those are nonempty. When the queues are empty or the *token_holding_timer* is expired, the station passes the

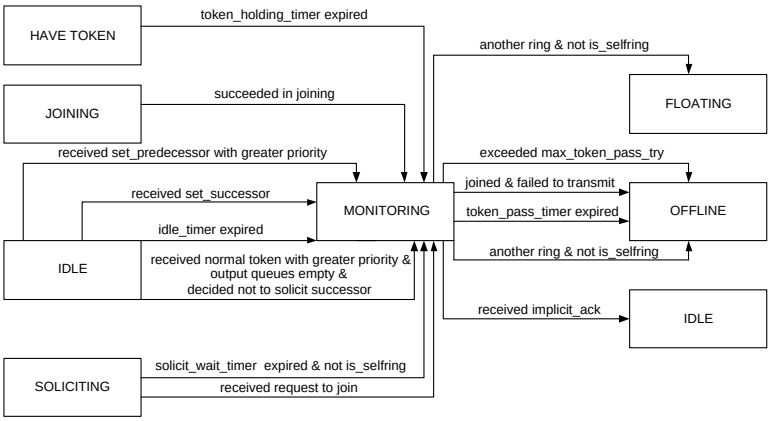

Figure 5.8: Transition From/To Monitoring State

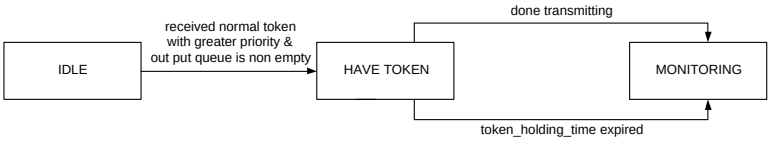

Figure 5.9: Transition From/To Have Token State

token to its successor and goes to *monitoring* state as seen in Figure 5.9.

5.3 Operating Types

5.3.1 Normal Operating Flow

The station does certain state changes when it is operating in *normal operating flow*. *Normal operating flow* is the flow when there is no joining process that means either the ring is full or there is no station outside of the ring. When the station gets the *token* in the *idle* state, it goes to *have token*, *soliciting*, and *monitoring* state, when there is a packet to send, when it decides to send *solicit_successor* or when the packet queues are empty and *solicit_successor* decision is negative, respectively. The station goes back to *idle* state from *monitoring* state when it gets the *implicit_ack*.

5.3.2 Saturation Operating Flow

The station operates in saturation condition when there are always packets to send. In this case, the station passes to the *have_token* state from the *idle* state and keeps sending packets until the *token_holding_timer* is expired. The station passes to the *monitoring* state after passing the *token* to its successor, and goes back to *idle* state after receiving *implicit_ack* as seen in Figure 8.1. The difference between *normal* and *saturation* operating flow is that the latter does not send *solicit_successor* and state transitions always follow the *idle, have token* and *monitoring* order, whereas the former does not follow an ordered state transition.

5.4 Summary

In WTRP, a station changes its state upon packet reception, packet transmission, or timer expiration. *Floating* and *offline* states are the states where the station only listens to the medium. In these states, the station does not belong to any ring. When a station belongs to a ring, it waits for *token* in the *idle* state and passes to *have_token* state if it has data to send. If the station has enough time, it may send *solicit_successor* token and pass to *soliciting* state. When a station wants to join a ring, it responds to a *solicit_successor* token and passes to *joining* state. The station confirms its transmissions in *monitoring* state. A station traverse around *idle, have token , soliciting* and *monitoring* states when it operates in *normal* operating conditions. If it operates in *saturation* condition, the station follows *idle, have token, monitoring* states in order.

Chapter 6

Implementation Overview

6.1 Introduction

Wireless Token Ring Protocol is deployed in three modes: {Simulator, User-Space, and Kernel Implementations}. We create a unified codebase that is shared by all the implementations. The main core of the protocol is the same for all three implementation and each implementation introduces certain hooks to the main core as seen in Figure 6.1. This unified codebase is made possible by the implementation of Linux kernel libraries for packet manipulations and event scheduling in user space.

Ad hoc networks are now being designed that can scale to very large configurations. Design and development costs for such systems could be significantly reduced if there were efficient techniques for evaluating design alternatives and predicting their impact on overall system performance metrics. Simulation is the most common performance evaluation technique for such systems. The simulator we developed attempts to address design and development cost reduction by providing an easy path for migration of simulation models to operational software prototypes and support for visual and hierarchical model design. The Wireless Token Ring Simulator (See Figure 6.1) allows debugging

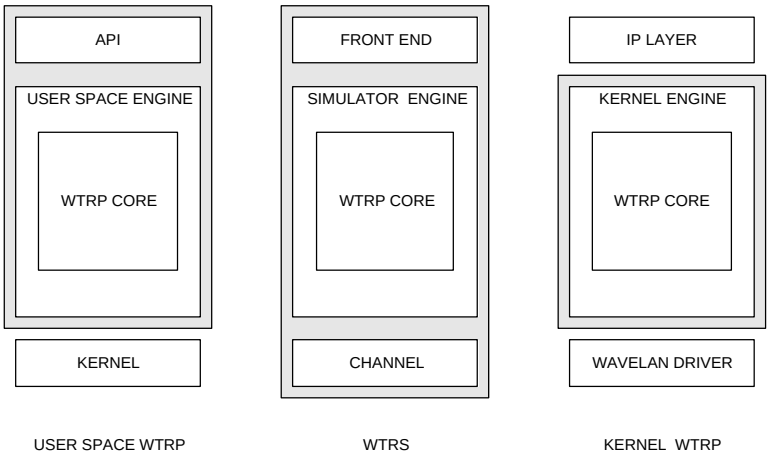

Figure 6.1: Implementation Overview

and simulating the WTRP core shared by all implementations.

The importance of the user-space implementation (Figure 6.1) is that it is platform independent. In a managed network, applications can reduce the frequency of packet collisions by sending the UDP packet using the UDP interface. In this case, WTRP is used as a transmission scheduling method rather than a medium access control.

The Linux kernel module is developed for kernel version 2.2.19, and it is inserted between the IP and WaveLAN libraries as seen in Figure 6.1.

6.2 State Transitions

State transitions in WTRP are defined in timer handler functions (See Section 4.3.3) and in *process_packet* function. These functions are from the unified codebase. The transitions that these functions introduce are explained in detail in Section 5. When the timer is expired, appropriate handler of the timer is called. When there is a packet reception, after removing the appropriate header of the packet according to each implementation, the protocol calls *process_packet* function.

6.3 Data Flow

Data packets coming from the medium are processed first in the *process_packet* and then *app_rx* is called. Data coming from the upper layer is first queued and then transmitted to the medium when the station gets the *token*. Successful transmission is notified to the protocol by *tx_done_handler* that re-checks the queues. If the queue is empty, the protocol passes the *token*, otherwise, sends another packet.

6.4 Data Structures

Station and *device* structure are the main structures that preserve the station information and implementation specific information respectively. *Station* structure contains the protocol state and is encapsulated in the *device* structure that contains the implementation parameters.

6.4.1 *Station* Structure

In the *station* Structure, protocol parameters of the station are kept which include {Identification, MIB, Ring Statistic, Joining and Monitoring Parameters, Timers, Packet Queues, Connectivity Table}.

Identification Parameters

A node is identified by MAC address of the node (TS), MAC address of the previous node (PS), MAC address of the next node (NS), MAC address of the ring owner (RA), Sequence Number of the node (seq), Generation Sequence Number of the node (genseq).

MIB Parameters

User specified parameters (See Section 4.2) are also pointed in the *station* structure.

Ring Statistic Parameters

The station calculates the number of nodes in the ring (*num_node*) by subtracting the *seq* of the token from the *seq* of station since in the rotation, *seq* number of the token is incremented by each station. Token rotation time (last_accepted) is calculated by taking the difference of two successive *token* arrival times.

Joining Parameters

When a station wants to join a ring, the station sets *wants_to_join* to true. After getting an invitation from a station, it sets the *soliciting_station* to the address of inviting station and *soliciting_station_successor* as the successor of the inviting station. This information is sent with *solicit_successor_token* (See Section 4.4.7).

Monitoring Parameters

The station tries up to *num_token_pass_try* before giving up passing token and exiting.

Timers

Station structure contains the timers. A timer is initialized by *init_timer* function which creates the timer structure by assigning the appropriate handler to the *handler* pointer, *device* structure to the *data* pointer and timer value to the *expiration* pointer of the timer structure.

Packet Queues

Station structure contains pointers to two queues; *out_buffered_queue* which is the queue for data packets that are buffered for later transmission and *out_tx_queue* which is the queue for the packets that are ready to be transmitted, but blocked due to hardware being busy (packet can be blocked due

to carrier sensing).

Connectivity Tables

Pointer to connectivity tables of the station is contained in the *station* structure. The connectivity table (*my_ctable*) holds information about local transmissions that are from the station‘s ring. The other connectivity table (*other_ctable*) holds information about all transmissions in the reception range.

Initialize station

Station is initialized by *initialize_station* function, When the function is called, all the parameters are set to default values.

6.4.2 *Device* Structure

Device structure contains the protocol information and implementation specific parameters. In the simulator implementation, position and velocity information, seed of the random variable, movement pattern, application list is contained here. In the user-space implementation, socket information is kept here. *Station* information is casted out from the *device* structure when it is passed to a function as an input.

6.5 Summary

In this chapter, we presented the approach that we take when designing the WTRP Implementations. WTRP has three deliverables; User-Space, Simulator, Kernel implementations. We created a unified codebase for each implementation. Common structures are presented. *Station* structure keeps protocol parameters and *device* structure contains *station* structure and implementation parameters.

User-Space implementation is a platform independent scheduler, Simulator is a tool to simulate the WTRP in wireless environment and do some performance analysis, and Kernel Implementation is a Linux link layer module.

Chapter 7

Wireless Token Ring Simulator

7.1 Introduction

WTRS (Wireless Token Ring Simulator) is a simulation library for Wireless Token Ring protocol. It is built on top of the WTRP Core that provides the finite state machine of the WTRP. It has been designed and built with the primary goal of simulating the actual implementation code in large network models that can scale up to a hundred or thousand nodes. This design of simulator is advantageous since it eliminates the need to develop separate program for simulation and for deployment. The separated development is not only expensive in terms of added developmental costs, but also increases the chance of failure to identify potential problems of the protocol since the protocols that are tested in simulations and deployed are often different.

As most network systems adopt a layered architecture, WTRS is being designed using a layered approach similar to the OSI seven-layer network architecture. Simple APIs are defined between different simulation layers. This allows the rapid integration of models developed at different layers by different people. Actual implementation code can be easily integrated into WTRS with this layered design. Now, WTRP was implemented in MAC layer.

Layer	Models
Physical	Free Space
Data Link	WTRP
Transport	UDP
Application	CBR

Table 7.1: Models Currently in WTRS Library

Table 7.1 lists the WTRS models currently available. WTRS supports three different mobility models. Nodes can move according to a model that is generally referred to as the “brownian” model. A node periodically moves to a position with randomly chosen x and y velocities. The second mobility model is called “bounce” in which a node moves linearly to a location with fixed x and y velocities specified in the configuration file. In these two mobility models, the initial position and initial velocities can be specified in the configuration file and when the node reaches the boundary of the canvas, it bounces back and continues. The third mobility model is called “manual”. In this mobility pattern, any movement pattern can be specified by chopping it into linear segments that is defined by initial and final positions, velocities and starting time in the configuration file. The node position is updated periodically by a parameter specified in the configuration file. Before describing each interface, one more point worth mentioning is the use of the same random function in the configuration. This ensures that the same output will be generated with the same seed. This helps to debug code easily and results of parameter modification can be tested in the same environment as before. The random function is *net_random*, identical to Linux. The seed can be specified in the configuration file.

7.2 Architecture

Overall design of the simulator can be divided into four parts, ordered from input to output;{animator, simulator core, analyzer and visual tracer} as seen in Figure 7.1. Animator creates the simulator

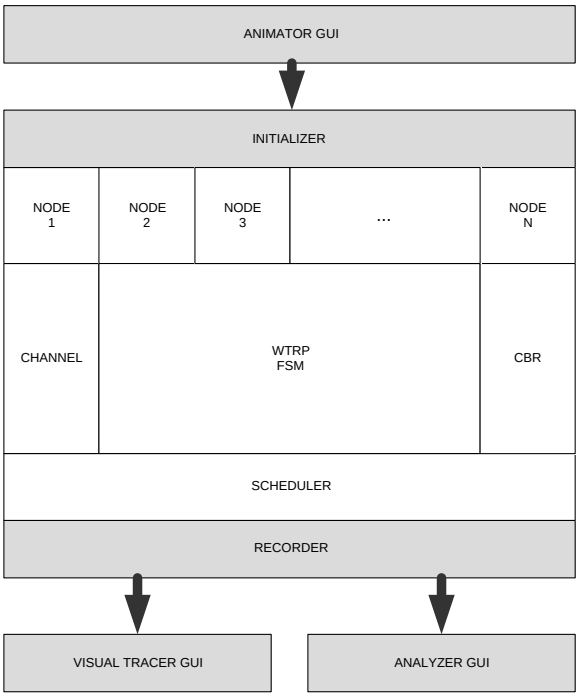

Figure 7.1: Overall Design

configuration information. The simulator core is the main part where the simulator engine, channel model, data application and WTRP module are located. Visual Tracer and Analyzer process the output files generated by the simulator. Currently, NAM [20] is used for visualization and analyzer file format is the same as that of NS [21] trace file.

7.2.1 Animator

As we mentioned before, simulation parameters are defined in animator. A snapshot of the GUI can be seen in Figure 7.2. Parameters of the simulation are shown in Table 7.2. These parameters are used as an input to the Simulator Engine (See Figure 7.1) and used at the initialization.

Parameter	Definition
Canvas size	Determines the size of the topology
Visual tracer output	File Name for NAM
Analyzer output	File Name for log info
Simulation time	Determines the execution time
Random Seed	Controls the outcome of the simulation
Record Function Frequency	Controls the output file size
Nodes & Nodes Address	Node Identification Parameters
Movement Patterns	Defines the movement patterns of the nodes
Nodes Topology Information	Defines the velocity and initial and final position information
Constant Bit Rate (CBR) Applications	Defines the application information

Table 7.2: Configuration Parameters

Figure 7.2: Animator GUI

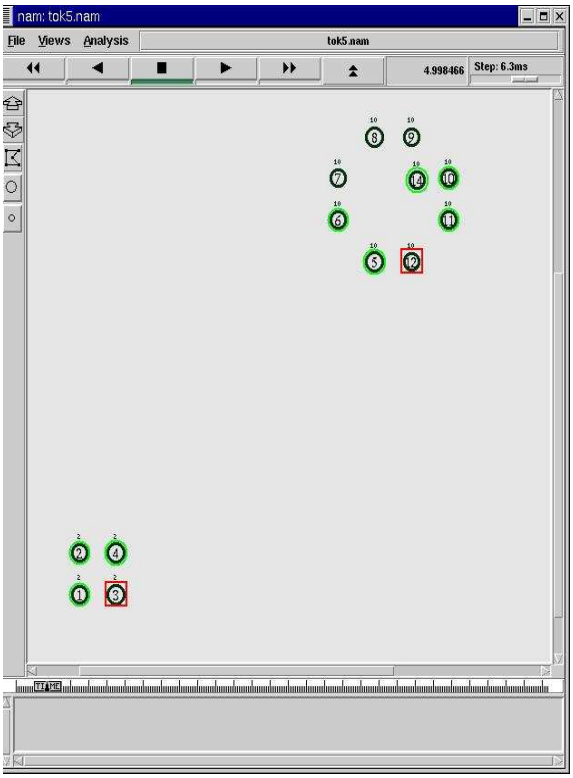

Figure 7.3: Visual Tracer

7.2.2 Visual Tracer & Analyzer

For visualization, a topological snapshot of the simulation is taken periodically and recorded in a trace file. The trace file is in NAM format and is viewed with NAM [20]. A screen shot of the visualization is shown in Figure 7.3. Movement, transmission, reception and ring address can be observed from the Visual Tracer.

We created a set of libraries that can be used to record performance-related events for Analyzer. We have implemented *record_packet_transmission* and *record_packet_reception* functions that write detailed information about that packet and station to the analyzer output file.

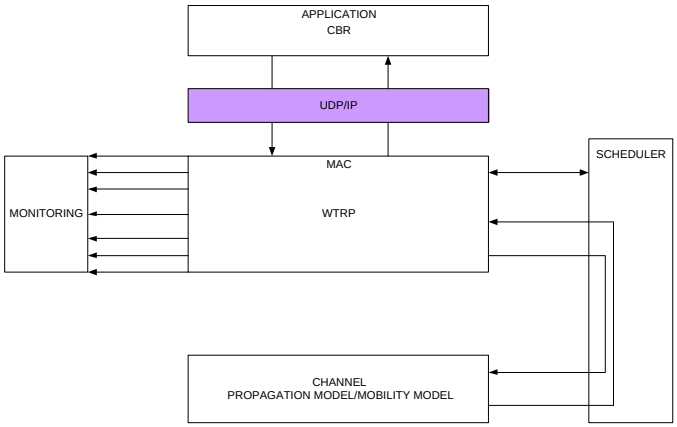

Figure 7.4: WTRS Architecture

7.2.3 Simulator Engine

Simulator Core consists of two basic parts: WTRP module and simulator module. The simulator provides interface for scheduling and packet manipulation code to the WTRP module. The simulator module also contains a simulation engine which allows simulation of multiple nodes.

Node data structure, Data agent (CBR), Scheduler, Channel module, Monitoring functions constitute the simulator module as can be seen in the Figure 7.1.

Node Data Structure

Instance of a node is represented by a data structure that holds state information. All the variables are node specific and are stored inside the *device* structure (See Section 6.4.2).

CBR Agent

Constant Bit Rate (CBR) is a basic data application. It sends periodic packets to the module. CBR module uses *tx_handler* function of the unified codebase to transmit. Being consistent with Linux implementation, packet information is encoded into *sk_buff* data structure. CBR structure is a linked list and initialized for each instance of CBR agent during the initialization state of simulator. From

```
struct timer_list {
struct timer_list *next;       //Pointer to the next event
struct timer_list *prev;       //Pointer to the previous event
unsigned long expires;         //Expiration Time
unsigned long data;            //Device structure
void(*function)(unsigned long);//Handler function
};
```

Figure 7.5: Event Structure

network module to the CBR as an acknowledgement *app_rx* can be used from the unified codebase.

Scheduler

Scheduler implements an event queue, add event and delete event functions. We used the same data structure and function calls as Linux. *Event* is a linked list, which has a specific handler function. Figure 7.5 shows the event structure. The scheduler functions *add_timer* and *delete_timer* are for adding and deleting event to the scheduler respectively. *Mod_timer* updates the expiration value of the event.

In simulator, both transmission and reception are scheduled as an event. Packet transmission to the medium is an event done by *transmit*. Making packet transmission an event is to introduce transmission delay. This improves concurrency among transmission events scheduled for exactly the same time. Function *_transmit* is called later in order to pass the transmission to the channel module.

Channel module decides on the success of packet transmission from the transmitter to every other node in the network pairwise. Channel module schedules packet reception event for each node by calling *add_timer* with handler function *rx_handler* from the unified codebase.

Channel Module

In terms of the channel model, the simulator models a simple Direct Sequence Spread Spectrum (DSSS) capturing effect based on signal to interference ratio, free space propagation, and underlying CSMA radio. Multipath or fading affect is not modelled. Packet Loss is modelled based on distance and power and packet corruption is modelled based on Signal to Interference Ratio.

Monitoring

Topology monitoring functions record NAM position information of the nodes by a periodic event called *record_function*. Periodicity of the event is set by a parameter in the configuration file. As a result, we can adjust recording period to reduce the size of output file at the expense of brittle movements of nodes. Monitoring functions mark the node with different color, shape or character for the duration of reception. Currently *start_transmission* highlights transmitting station red and *start_reception* marks the receiving station green and ring address of the node is marked on top of it. A snap shot of the NAM can be seen in Figure 7.3.

The *record_transmission* and *record_reception* functions are used for the analyzer output file in order to keep a log of the packet events. These functions create a log of information of the packet header with a time stamp.

After discussing basic hookups of the simulator, We can investigate the simulator run in more detail.

7.2.4 Simulator Run

Simulator starts with initialization process. The initialization steps are as follows:

1. *Initialize_simulation*

```
while(clock< simulation_time) {
tmp=take_event();   // Takes the event from the queue
update_clock ;      // Sets the time to the time of the event
tmp.handler();      // Calls the handler function of the event
}
```

Figure 7.6: Main Loop

Initialize simulation is where the node structures are created, the memory is allocated, and the events are initialized.

2. *Initialize_topology*

   Initialize topology is used for initializing nodes for the topology.

3. *Initialize_traffic*

   Initialize traffic creates the linked list of CBR structures and attaches them to the nodes.

4. *Initialize_record_function*

   As mentioned above, *record_function* is used for recording the monitoring information periodically in order to reduce the output files and it is scheduled here. The handler records the positions and schedules itself as an event again.

5. *Main Loop*

   After initialization, simulator goes into the main loop and runs for *simulation_time*, a parameter of the configuration file. The Main loop implements an event driven simulation. Depending on the next event, the clock jumps to that time and calls its handler function as seen in Figure 7.6.

7.3 Summary

Simulator is designed for the purpose of simulating the actual code in a wireless environment with multiple nodes. The initial aim of the simulator is to facilitate the implementation code and do rudimentary performance analysis. An instance of a node is represented by a data structure that holds state information, and the code base of the WTRP module provides rules for state transitions. Delay in events and timer functions are implemented as scheduling of events to be executed later. The scheduler takes the event, runs the code, and then adds the event. This cycle runs the simulation.

Chapter 8

User Space Implementation

8.1 Introduction

WTRP manages the medium by scheduling the transmission. Stations start transmitting upon reception of a token and are forced to stop transmitting after a fixed time. This idea can be considered as a scheduler and extended to be used by the applications to reduce the frequency of packet collisions among the application that runs on top of WTRP. User-Space Implementation deploys this scheduler idea and is designed for applications using the UDP socket. The implementation is designed in the application layer in order to be platform independent. Applications use the provided WoW API [3] (Web Over Wireless Application Program Interface) and WoW API re-routes the UDP packets to the WTRP module that sits in the application layer and WTRP module does the scheduling.

8.2 Implementation

8.2.1 Initialization

User Space implementation uses IP address instead of MAC address to distinguish a node. An node reads the IP address from *(/etc/hosts)* in Linux . In the initialization state, *init_station* calls

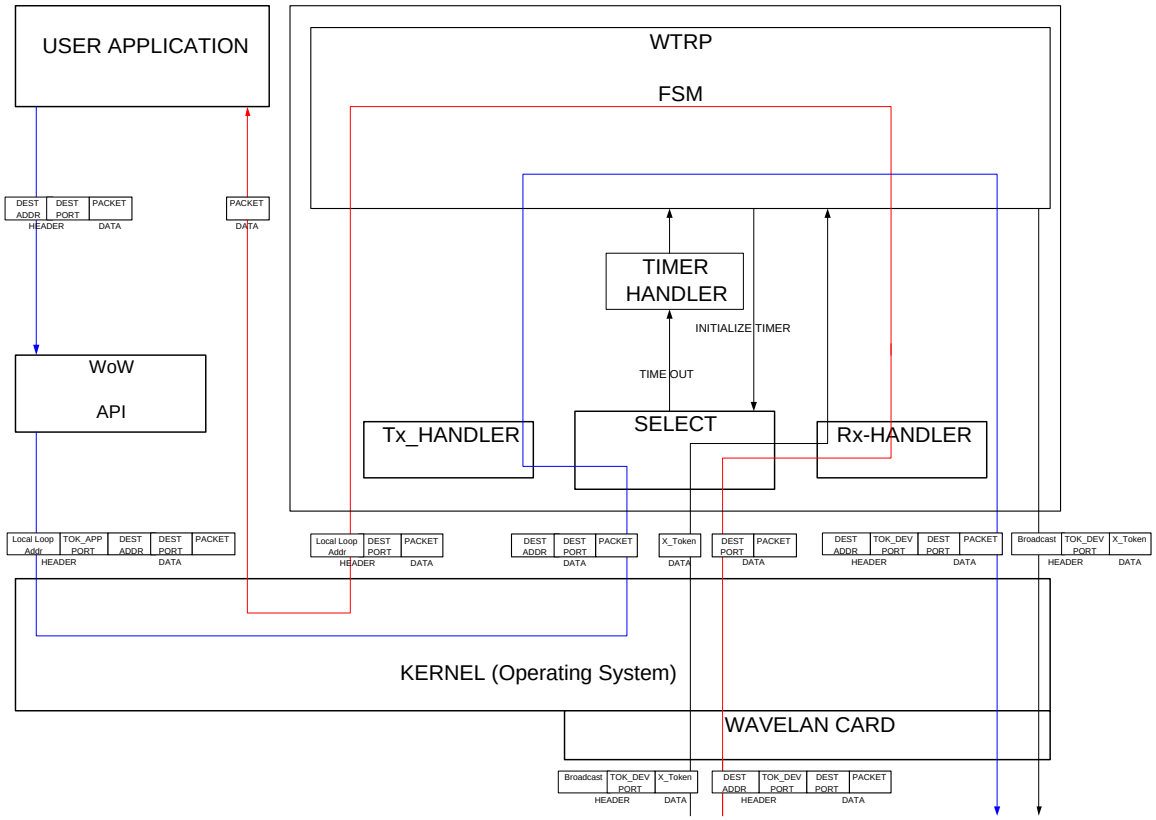

Figure 8.1: User Space Engine

initialize_station from unified codebase to create the *station* parameters and *init_timer* for all WTRP timers (See Section 4.3). At the end, *init_station* starts the station in *floating* state if the *IOCTL* option is not set. When *IOCTL* option is set, station waits to join a ring until it gets "request medium" system call from an application.

UDP sockets are used for packet transmission and reception. UDP ports are open for applications and the medium by *initialize_socket*. Applications use *TOK_APP_PORT* to send packet to WTRP Module or receive packet from WTRP module via local loop. WTRP module uses *TOK_DEV_PORT* when sending packet to the medium or receiving from the medium as seen in Figure 8.1.

8.2.2 System Call (IOCTL)

System Call (IOCTL) is implemented to control the WTRP module from the application. When *IOCTL* option is set, the station waits for a request from an application. Request is sent through the local loop to the WTRP module and if the request is *REGISTER*, the application that is registering is added to the application list of the station. If the request is *JOIN*, station goes to *floating* state to join a ring. If the request is *LEAVE*, station leaves the ring after getting the token. The request *GET_INFO* sends the *station* parameters to the application.

8.2.3 Application Interface (WoW API)

Application Interface (*WoW API*) is a set of functions that can be used by datagram applications to use the protocol developed in user space. The idea in the API is that an application uses WOW socket functions instead of the Linux socket functions. The Linux socket functions {*socket, bind, recvfrom, sendto*} are replaced by {*my_socket, my_bind, my_recvfrom, my_sendto*} respectively. *my_ioctl* is introduced for *IOCTL* option similar to the *ioctl* system call in Linux.

```
while(1) {
update timer;                    // Update Clock.
retval=select();                 // Listen for a fixed time.

     if retval=0                 // No packet detected.
            call timer_handler; // Handler of the timer.
else if from TOK_APP_PORT        // Packet from application.
            call tx_handler;    // Handler of application.
else if from TOK\_DEV\_PORT      // Packet from medium.
            call rx_hanler;     // Handler of medium.
else ASSERT(not_reached);
}
```

Figure 8.2: Main Loop

my_socket calls *socket* in order to open a UDP socket.

my_bind changes the socket destination address to local loop address ("127.0.0.1") and calls *bind* to get a file descriptor for the socket.

my_recvfrom changes the source address to local loop address and calls *recv_from*.

my_sendto creates a new message by appending the destination address to the original message and sends the packet with the local loop address as the destination to the *TOK_APP_PORT* by calling *sendto*.

my_ioctl sends the *REQUEST* and the *value* to the WTRP Module by *sendto* via the local loop address and *TOK_APP_PORT*.

8.2.4 User Space Engine

After the initialization, a station runs in an infinite loop as seen in Figure 8.2.

Scheduling function, *select*, listens to two ports, *TOK_DEV_PORT* and *TOK_APP_PORT*, for a certain amount of time that is determined by expiration value of the initialized timer. Function *select* returns positive if there is a packet in the ports or returns zero if the timer expires. If there is no packet, the protocol calls the *handler* function of the timer. If the *select* function returns because of a packet in the *TOK_APP_PORT*, the protocol calls the *tx_handler*. If the packet is in the *TOK_DEV_PORT*, the protocol calls the *rx_handler*. As can be seen in Figure 8.1, packets from the application are handled by the *tx_handler* and packets from the medium are handled by the *rx_handler*.

8.2.5 Transmission

When an application sends a packet to the medium, the packet is modified and directed to the WTRP module by WOW API (See figure 8.1) and the protocol calls *tx_handler*. Function *tx_handler* examines the data and the destination, and calls *process_packet* if the destination is local loop; otherwise, it calls *tok_tx_handler*. *Process_packet* is called because if the destination is local loop, this means that two applications at the same station are communicating, therefore the packet should be directly forwarded to the destination. Function *tok_tx_handler* puts the packet to the packet queue to be transmitted after the station gains *token*. When the station gets the *token*, station sends the packet to its destination through the *TOK_DEV_PORT* with destination port appended to the data as seen in Figure 8.1. Control packets are broadcast through the *TOK_DEV_PORT*.

8.2.6 Reception

Similar to the MAC layer, all packets received from the medium are from the *TOK_DEV_PORT* captured by the *rx_handler* that calls *tok_rx_handler*. Function *tok_rx_handler* calls *process_packet* from the unified codebase. *process_packet* examines the packet and if it is a data packet, it calls

app_rx in order to send it to the application via the local loop. If the packet is control packet, *process_packet* processes the packet and executes the corresponding state transitions.

8.3 Summary

User-Space implementation is a platform independent implementation running in the application layer for applications that uses UDP sockets. The implementation uses UDP sockets to communicate with the applications and the medium. A UDP interface is provided for applications in order to re-direct their packets to the WTRP module. If the *IOCTL* option is set, an application can send commands to the ring by the *IOCTL* calls. A station attempts to join or leave a ring after getting the corresponding system signal from the application. Linux scheduling function, *select*, listens to the application and medium sockets for the corresponding expiration value of the state timer. Three events drive the implementation. Timer handler is called when there is no packet. Application handler is called when the packet is from an application. Finally, medium handler is called when the packet is from the medium. Outcomes of these events are inputs to the WTRP module.

Chapter 9

Kernel Implementation

9.1 Introduction

In this chapter, we present the detailed description of the Kernel Implementation. Kernel Implementation is a link-layer module, derived from the unified codebase. Some of the kernel functions and data structures are used in non-kernel code. These functions are scheduling functions and socket buffer data structure. The Linux protocol stack does not maintain layering below the network layer. It performs some rudimentary logical link control functions itself and delegates the rest of these low-level functions to the device where it is integrated. While this may lead to better performance, for research and development of link-layer protocols we prefer to retain the full layering in a completely modular way as in ISO layering model (See Table 9.1).

We first explain the usual path between the network device and the network-layer protocols, and then the path taken by our implementation. Every computer requires peripheral hardware devices to operate. Each device plugs into some type of port or socket. This port or socket may be part of an interface card which plugs into the system bus; or the socket itself may be on the bus, as in the case of PCMCIA sockets [24]. In any case, to the computer, the peripheral device is an entity at a

Layer 7	Application Layer
Layer 6	Presentation Layer
Layer 5	Session Layer
Layer 4	Transport Layer
Layer 3	Network Layer
Layer 2	Data Link LLC (Logical Link Control) Sublayer MAC (Medium Access Control) Sublayer
Layer 1	Physical Layer

Table 9.1: ISO Layering Model

particular address which sends and receives data over the bus. When the device has data to send over the bus, it must interrupt the computer so that the computer can stop whatever it is doing, receive the data, and respond appropriately. Thus, each device has a unique IRQ number. When the operating system, in this case Linux, receives an interrupt, it passes control to the interrupt handler registered for that IRQ, usually in the device driver which is bound to that device. This interrupt handler does the necessary processing of the information received from the device, before returning control to the operating system. It must be fast and efficient, because while it is handling this interrupt, interrupts are disabled: no other device can interrupt the system.

One function of the device driver is to handle interrupts from the device. Another function is to send data to the device, perhaps to control it or perhaps to transmit through the device (such as a networking device). In this way, the kernel does not have to know all the details about the communication through well-defined interfaces, and it is up to the device driver to interpret between such an interface and its particular device or class of devices.

In the early days of Linux, all device drivers were built into the kernel image loaded at boot time. But this quickly became impractical, so now most device drivers are also available as modules which can be loaded or unloaded as needed, while the operating system is running. In any case, the device driver must be bound to the particular device for which it is responsible. That is, it must know the IRQ of the device so it can register to receive interrupts from that device, and it must know

the address at which it can communicate to the device. This may happen through parameters passed to the device driver module when it is loaded, through autoprobing, or through some other means such as the Card Services daemon.

In the case of network devices, Linux treats each device as an abstraction defined by a *device* data structure (*net_device* in kernel version 2.4). This structure[1] conventionally referenced through a pointer as *struct device* $*$ *dev*, includes several function pointers as well as some data fields. These define the API through which the kernel interacts with the device. It is the responsibility of the device driver to allocate this structure, fill in the function pointers and the data fields appropriately, and register the device with kernel. A network device usually issues interrupts for two reasons: to notify the system that it has received a packet, or to notify the system that it has finished transmitting and is ready for more packets. The device driver must register itself to receive these interrupts and handle them appropriately. Linux handles all networking packets through an abstraction called a socket buffer, defined by a *sk_buff* data structure. This structure[2] conventionally referenced through a pointer as *struct sk_buff* $*$*skb*, contains many data fields specially adapted for networking. The most important of these, of course, is data, which contains the actual data in the packet. However, this is not as simple as it seems; the *sk_buff* structure is allocated just once and passed up or down through all layers, *skb->data* shrinks as each layer interprets its own header and respective header so it is still accessible. All this is done through pointer manipulations with a minimum of copying or reallocation of memory. In between some of these stages, the *skb* may be placed on various queues to await processing, so it also includes links for this purpose. When a device receives a packet, it issues an interrupt which is serviced by the device driver. The interrupt handler in the device driver allocates a *struct sk_buff* structure and copies the data into this structure. It then reads the link layer header to determine which protocol (usually, a network layer protocol) should handle the packet

[1]The structure is defined in */usr/src/linux/include/linux/netdevice.h*
[2]The structure is defined in */usr/src/linux/include/skbuff.h*

next. It assigns this protocol to the *skb->protocol* field, and then it calls *netif_rx(skb)*. Finally the interrupt handler exits, returning control to the operating system.

9.2 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.

9.3 Real Time Processes in Linux

The Linux kernel runs at 100 Hz by default in x86 compatible hardware and 1024Hz in Sun. The 100 Hz clock translates into the clock granularity of 10ms. The coarse granularity creates problems because WTRP runs much faster than 10ms, and the protocol uses many timers. To combat this problem, one can increase the granularity of the clock by increasing the HZ value in */usr/src/linux/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 P3 500Hz and 700Hz laptops to 2048 Hz. This has resulted in reduced joining and fault recovery time and increased stability of the token ring.

9.4 Old tale of WaveLAN

Old tale of WaveLAN [22] describes the Kernel and WaveLAN interface before the modification. Below are small portions of the code that illustrate how the WaveLAN device driver works. The

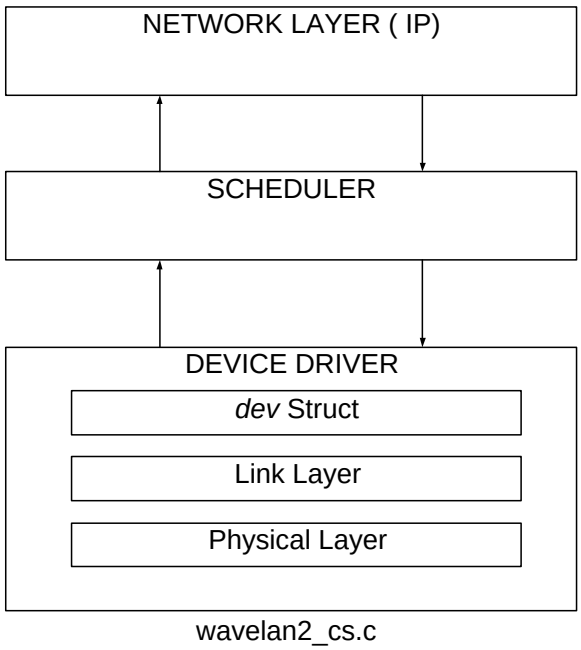

Figure 9.1: Old Tale of WaveLAN

general idea is necessary to understand the implementation. For details on the implementation, please refer to the WTRP code. 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->busy* flag is used by the logical link control to control the flow. The *sk_buff* 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* 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.

9.4.1 Initialization

Insertion

When WaveLAN card is inserted, *PCMCIA Card Service* detects the type. *Card Services* loads the device driver module *wavelan2_cs.o* and calls *adapter_attach*.

Attachment

Function *adapter_attach* is needed to attach PCMCIA driver. The function creates *struct dev_link_t *link* and *struct device *dev*,* sets function pointers, registers *link* with *Card Services*. Function *adapter_attach* registers card event handler *(adapter_event)* with *Card Services*.

Hooks

These are the hooks that bind WaveLAN card with the OS.

Interrupt Service Request *link->irq.handler =wvlan2_isr*

Transmission *dev->hard_start_xmit=wvlan2_tx*

Control *dev->do_ioctl = wvlan2_ioctl*

Via call to *ether_setup*

Transmission Header *dev->hard_header = eth_header*

Ditto,After Address Resolution *dev->rebuild_header=eth_rebuild_header*

Insertion Redux

Card Services notifies adapter of card insertion event. Function *adapter_event* dispatches event to *wvlan_insert* and *wvlan_insert* sets device IRQ and IO port address, marks device as ready to transmit, registers *dev* with kernel and configures WaveLAN card.

9.4.2 Transmission

This part describes the packet transmission from network layer to the wireless medium as seen in Figure 9.1. Packet from the upper layer is sent to the scheduler and then to the device when the device is ready to transmit.

Network Layer

Network layer creates or modifies packet and calls *dev->hard_header* to push link layer header on the packet. Network layer resolves address and calls *dev->rebuild_header* to correct address. Network layer calls *dev_queue_xmit*.

Kernel Scheduler

Function *dev_queue_xmit* puts packet on device queue and calls *qdisc_wakeup* to start queue. Function *qdisc_wakeup* checks if device is ready to transmit and calls *qdisc_restart*. Function *qdisc_restart* calls *dev->hard_start_xmit* that passes the packet to the device driver.

Device Driver

Device Driver controls the *device* structure, link layer and physical layer. Function *wvlan2_tx* does the following:

1. checks if device is ready to transmit
2. notifies if device is busy transmitting
3. disables interrupts
4. gives packet to card for transmission
5. notifies if device is ready to transmit

6. enables again the interrupts

The card interrupts in order to notify done transmitting. Function *wvlan2_isr* notifies that device is ready to transmit, marks *NET_BH* and restores interrupts.

9.4.3 Reception

Packet is detected from the card and forwarded to the upper layers. Device Driver gets the packet from card and cast the packet into *skb* structure and decodes the ethernet header. Kernel scheduler pulls the packet from the device driver and sends it to the upper layer.

Device Driver

Card interrupts to notify that packet is received. Function *wvlan2_isr* calls *wvlan2_rx*. Function *wvlan2_rx* gets packet from card, puts packet into (struct *sk_buff* $*$*skb*) and decodes and "pulls" Ethernet header. Function *wvlan2_rx* calls *netif_rx* and *netif_rx* marks *NET_BH*. After these, *wvlan2_isr* restores interrupts.

Bottom Half

Kernel scheduler runs task queues and runs networking task queue (*net_bh*). Function *net_bh* runs pending transmissions, pulls packet off received queue and checks protocol type. Function *net_bh* sends packet to functions registered to handle this protocol type on all devices. Handlers that are bound to packet socket receive the packet.

Packet distribution to functions registered to handle this type on this device is done by *net_bh*. Network layer handlers such as *ip_rcv* or *arp_rcv* receive packet.

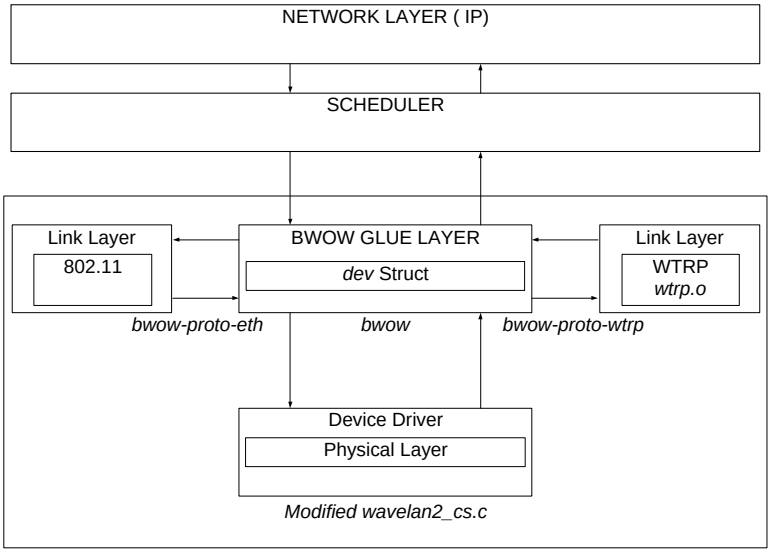

Figure 9.2: WTRP Kernel Architecture

9.5 Installing WTRP to Kernel

In the implementation, *dev* structure is put into *BWOW*[3] Glue Layer. This layer directs the packets to the loaded link layer module. In the kernel implementation, both WTRP module and 802.11 module exist. Modular structure allows user to select the preferred MAC layer.

9.5.1 Initialization

Delegation

We modified the *wavelan2_cs.o* module in the sense that at initialization, it requests *BWOW Glue Manager Module* (*bwow.o*) and registers *BWOW_wavelan2_setup* with *BWOW*. *adapter_attach* now does not create *struct device ∗dev* and does not set hooks in *dev* since *dev* is encapsulated in *BWOW Layer* (See Figure 9.2). Instead of these, *adapter_attach* calls *bwow_attach_dev* that manages the attachment.

[3]BWOW stands for Berkeley Web Over Wireless Group

Attachment

Function *bwow_attach_dev* creates *struct device *dev* and registers *dev->init = bwow_init_dev*. In the kernel implementation, function *wvlan2_insert* does not register *dev* structure with kernel but *bwow_attach_dev* registers *dev* with kernel. After these, kernel calls *bwow_init_dev*.

Other than *dev* structure, *bwow_attach_dev* creates *struct bwow_channel *ch* and */proc* filesystem entries that enable the user to change the WTRP parameters at run time and observe some statistics. After these, *bwow_attach_dev* calls *BWOW_wavelan2_setup* that set up the WaveLAN interface.

Hooks

Hooks are set in *BWOW Glue Layer*. Function *bwow_init_dev* calls *bwow_reset_dev* and function pointers are set as below by *bwow_reset_dev*.

Transmission *dev->hard_start_xmit = bwow_xmit*

Control *dev->do_ioctl = bwow_ioctl*

Transmission Header *dev->hard_header = bwow_header*

Ditto,After Address Resolution *dev->rebuild_header= bwow_rebuild_header*

BWOW_wavelan2_setup sets function pointers as

Transmission *ch->HW_send_packet= wvlan2_tx*

Control *ch->HW_ioctl= wvlan2_ioctl*

9.5.2 Loading the Protocol

User takes some action in order to load the WTRP. User loads WTRP main protocol module *wtrp.o*. *wtrp.o* module initialization registers *wtrp_rcv* with kernel to handle packets of type *ETH_P_WTRP*.

User loads protocol glue module *bwow_proto_wtrp.o* and *bwow_proto_wtrp.o* module initialization registers *bwow_wtrp_init* with *BWOW*. User writes *wtrp* to */proc/bwow/wavelan0/protocol*. *bwow_write_proc* sets *ch->protocol* and calls *bwow_wtrp_init*.

9.5.3 Initializing the Protocol

Function *bwow_wtrp_init* registers *dev*, *bwow_wtrp_up_to_network*, and *bwow_wtrp_data_transmit* with WTRP. Function *wtrp_register* creates *struct station_struct *station* and calls *init_station*, associating with *dev*. *bwow_wtrp_init* sets function pointers and creates */proc* filesystem entries; State, Ring Address, Number of Nodes, Max Token Holding Time, Contention Time, Last Token Rotation Time, Generation Sequence Number, Sequence Number.

9.5.4 Protocol Hooks

The hooks about transmission, reception, and queues are described below.

Reception- *ch-LINK_rx=bwow_wtrp_rx*

Transmission- *ch->LINK_xmit=bwow_wtrp_xmit*

Protocol Busy- *ch->LINK_stop_queue = bwow_wtrp_stop_queue* notifies that protocol is busy.

Protocol Available- *ch->LINK_wake_queue = bwow_wtrp_wake_queue* notifies availability of the protocol.

9.5.5 Headers on Data Packets

WTRP may push just a standard Ethernet header onto data packets when *dev->hard_header* points *eth_header* and after address resolution *dev->rebuild_header* points *eth_rebuild_header*. This is called *transparent_data*

WTRP may push its own header onto data packets when *ch->LINK_header* points wtrp_header and after address resolution when *ch->LINK_rebuild_header* points *wtrp_rebuild_header*.

9.5.6 Transmission

Into WTRP

Packets coming from upper layer are directed from the BWOW layer to WTRP module. If WTRP pushes its own data header network layer calls *dev->hard_header* which points to *bwow_header* and *bwow_header* calls *ch->LINK_header* which points to *wtrp_header*.

Kernel scheduler calls *dev->hard_start_xmit* which points to *bwow_xmit* and *bwow_xmit* calls *ch->LINK_xmit* which points to *bwow_wtrp_xmit*. *bwow_wtrp_xmit* calls *wtrp_data_request* and *wtrp_data_request* calls *tok_tx_handler*. *tok_tx_handler* is one of the function from the common codebase.

Out of WTRP

The created packet inside the WTRP module goes to the hook *transmit*. Function *transmit* checks whether the device is ready or not. If it is, *transmit* calls *bwow_wtrp_data_transmit* , registered previously. *bwow_wtrp_data_transmit* calls *ch->HW_send_packet*, which points to *wvlan2_tx*. When *wvlan2_tx* starts transmitting, it calls *bwow_notify_hw_busy* and *ch->LINK_stop_queue* is called by *bwow_notify_hw_busy* and it points to *bwow_wtrp_stop_queue* and *bwow_wtrp_stop_queue* calls *wtrp_notify_hw_busy*.

Completion

When transmission is done, *bwow_notify_hw_available* is called and it calls *ch->LINK_wake_queue*, which points to *bwow_wtrp_wake_queue*. *bwow_wtrp_wake_queue* calls *wtrp_notify_hw_available*.

wtrp_notify_hw_available queues *wtrp_bh* on the immediate queue and marks *IMMEDIATE_BH*. Kernel scheduler runs task queues and runs immediate task queue: *immediate_bh* that runs *wtrp_bh* and *wtrp_bh* calls *tx_done_handler*, part of the common codebase.

9.5.7 Reception

When the packet is detected at the card. *BWOW* layer gets the packet from the physical layer and directs it into WTRP module. The packet is processed in the WTRP module and sent back to *BWOW* layer. The *BWOW* layer sends it to the upper layers (See Figure 9.2).

Into WTRP

Function *wvlan2_rx* does not decode or pull the Ethernet header, set *skb->protocol*, or call *netif_rx*. Function *wvlan2_rx* calls *ch->LINK_rx*, which points to *bwow_wtrp_rx*. Ethernet header is decoded and pulled by *bwow_wtrp_rx*. If the protocol is *ETH_P_WTRP*, then *bwow_wtrp_rx* calls *wtrp_data_received*, otherwise it sets *skb->protocol* and calls *netif_rx* and *wtrp_data_received* calls *process_packet*, part of the common codebase.

Out of WTRP

The packet goes to the hook *app_rx* from the common codebase and *bwow_wtrp_up_to_network* registered previously is called by *app_rx*. *bwow_wtrp_up_to_network* sets *skb->protocol* and calls *netif_rx* function.

9.5.8 Overhead

We used 802.11 WaveLAN Silver card in our implementation. Although the control packets are broadcast, the data packets are unicast. In the card, when operated in the broadcast mode, RTS/CTS

```
/proc/bwow/wavelan0/processing_time
/proc/bwow/wavelan0/max_num_token_pass_try
/proc/bwow/wavelan0/solicit_successor_prob
/proc/bwow/wavelan0/max_token_holding_time
/proc/bwow/wavelan0/transmission_time
/proc/bwow/wavelan0/max_token_rotation_time
```

Figure 9.3: User Defined Parameters

is disabled but it is still active in unicast mode. The overhead that RTS/CTS introduces is examined closely in Section 10.2.

9.5.9 User Interface

Some of the parameters of WTRP can be seen at run time at *proc* file system. User can also change the MIB parameters of the WTRP at run time by writing to these files. Parameters that user can change are presented in Figure 9.3. User can change these parameters by writing numerical values to the files at run time. We discussed these and all the other parameters in Section 4.2. Many of the parameters used by WTRP describe time intervals. All timer values are measured in jiffies. We explained this unit of time at Section 9.3.

9.6 Summary

Kernel Implementation is a Linux Link Layer module and uses WTRP as the MAC protocol. Linux manages the device as modules and they can be loaded and unloaded at run time. Each device is controlled by the *device* structure. Linux kernel leaves *device* structure, Link Layer and Physical Layer control to the device driver.

In the kernel implementation, we separate the *device* structure and *Link Layer* from the device

driver and WTRP module is implemented into Link Layer as the MAC protocol. A *BWOW Glue Layer* is introduced that encapsulates the *device* structure. Data coming from and going to the device driver first comes to the BWOW Glue layer and is directed to the WTRP module. WTRP module then controls the usage of medium by the station.

Kernel implementation creates logs for the station parameters in the *proc* file system. Each parameter is represented by a file and MIB parameters can be changed at run time by writing into them.

Chapter 10

Analysis

10.1 Proof of Stability

10.1.1 Introduction

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

We first prove that interference is cancelled; after the creation of the first ring only that ring is allowed to transmit and all equivalent tokens are deleted by multiple token resolution protocol in finite time. Then, we show that the ring becomes stable in a finite time and finally, we prove that in one ring the bijection that represents correct relationship between predecessor and successor (see definitions under Section 10.1.3 for the exact definition) increases monotonically to the maximum number of nodes in the graph in finite time. When all bijections converge, we say that all rings, operating in different channels, are correct.

10.1.2 Model

Constants

1. MTRT: Maximum Token Rotation Time. (Defined more precisely in lemma 10.1.6)
2. M: set of all MAC addresses
3. IDLE_TIME: the amount of time that a station waits for token in a ring. IDLE_TIME$\geq$MTRT
4. CLAIM_TOKEN_TIME: the amount of time that a station waits before regenerating a token when the medium is quiescent. CLAIM_TOKEN_TIME$\geq$IDLE_TIME
5. INRING_TIME: the amount of time that a station waits when the station is not receiving any acceptable token, before going into the OFFLINE state. INRING_TIME$\geq$IDLE_TIME$\geq$MTRT, INRING_TIME $<$ 2 IDLE_TIME
6. MAX_NoN: maximum number of nodes that is allowed in a ring.

10.1.3 Graph

1. Definitions
   (a) The adjacency 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)$\in$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; 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.)

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

   $p.non \in Z$ //number of nodes in the ring of 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 priority 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.

Station

1. Attributes

x.genseq(t)$\in$Z // the generation sequence number of the last token that x accepted.

x.ra(t) $\in$M// ring address of x

x.TS $\in$M// the MAC address of x

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

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

x.non $\in$Z // the number of nodes in the ring of station.

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 last accepted.

3. Assumptions

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

10.1.4 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.

10.1.5 Proof

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 OFFLINE state.

Out-Ring Interference Cancellation

Lemma 10.1.1 *When a station hears a transmission from another ring, it suspends its transmission.*

If a station hears a transmission from another ring, it goes to FLOATING or OFFLINE state when it is in a self-ring or normal ring respectively by the implementation. ■

Lemma 10.1.2 *At time $k \leq$ CLAIM_TOKEN_TIME, a station detects a ring.*

For a station, it takes maximum one MTRT to hear a transmission, if there is a ring in the medium. If there is no ring, it takes maximum one CLAIM_TOKEN_TIME to hear a transmission since one of the stations needs to create a self-ring and transmit the token. ■

Lemma 10.1.3 *If there exists a ring operating in a channel, interference at that channel from the stations out of the ring is eliminated within a CLAIM_TOKEN_TIME.*

From lemma 10.1.1 and 10.1.2, we know that a station detects a ring in a finite time and we know that when a station detects a ring, it suspends its transmission. As a result, only the ring operating it that channel is allowed to transmit. ■

In-Ring Interference Cancellation

Lemma 10.1.4 *While a station is not in the OFFLINE 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 10.1.5 *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.*

Assume by contradiction that 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 OFFLINE state at any time after 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 OFFLINE state for one MTRT), by the implementation. Because a station is not allowed to receive a token when it is in the OFFLINE state, it could not have received p before time t_i + MTRT. Thus, x_i could not have been in the OFFLINE state since time t_i. Because of lemma 10.1.4, the priority of a station can only increase while it is not in the OFFLINE state and thus, it does not make sense that token p could have survived station x_j. ■

Lemma 10.1.6 *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, 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 10.1.7 *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 10.1.8 *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 10.1.6. 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 10.1.9 *There exist a time s, such that s $<$ t + MTRT and no multiple equivalent tokens exist any time u $>$ s.*

This directly follows from 10.1.7 and 10.1.8. From 10.1.8, 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 10.1.7, no multiple equivalent tokens exist at u $>$ s. ■

Lemma 10.1.10 *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 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 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. ■

Stability

Lemma 10.1.11 *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 each station in a ring has 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_PREDECESSOR 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 each station in the ring has a 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_PREDECESSOR.

Now we will show after t + MTRT+ INRING_TIME, a station outside a ring cannot possibly send a SET_PREDECESSOR 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 on, no more multiple equivalent tokens exist according to lemma 10.1.9. A station cannot possibly remember 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, it is 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. ■

Ring Enlargement

Lemma 10.1.12 *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 10.1.13 *When a station goes out of ring (into the OFFLINE state), $|S|$, the number of the bijections, remains positive.*

Let us say the predecessor of y is x, and the successor of y is z. When station y goes into the OFFLINE state, it waits for a invitation without changing its channel and changes when it detects a token p.non=MAX_NoN.

We distinguish the two cases where a station can be kicked out. The first case is when the INRING_TIMER expires (Section 4.3). In this case, from the lemma 10.1.8, 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 remain positive, because in the worst case we lose the bijection from x to y, but x creates a bijection with another node or a self-bijection. ■

Lemma 10.1.14 *When the number of bijections is positive, after a station accepts a token some time after t, $|S|$, the number of the bijections, monotonic increases.*

After accepting a token, station y either goes into the OFFLINE state, or attempts to pass the token to its successor, or sends the SOLICIT_SUCCESSOR token. $|S|$ is non-decreasing in the first case

since the station does not go voluntarily into the OFFLINE state according to our assumption. 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 OFFLINE state. And we have already shown in the lemma 10.1.13, that $|S|$ remains positive. 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 10.1.15 *$|S|$ monotonically increases and will converge to $|V|$ in a finite time.*

From lemma 10.1.13 and lemma 10.1.14, we have proven that $|S|$ is non-decreasing. We have also shown in lemma 10.1.11 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 is 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, there exists a station y that is waiting to join at time t. Station y waits to join until it hears a token that p.non=MAX_NoN, then it changes the channel. In this case, we gained a full ring in one channel and start to create another ring in another channel. From lemma 10.1.3, the stations operating in the other channel detects a ring in one CLAIM_TOKEN_TIME. If the station y accepts a token, allowing another station to form the bijection with y, then we gain in the size of $|S|$. 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 10.1.14, we know that there exists one and only one token in all rings. ■

10.1.6 Conclusion

For this proof to be applicable, 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, in the 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 compared to the rate of transmission. The assumption that there will be no transmission errors for the duration in magnitude of MTRT may be valid when considering the fact that wireless link is a type of transmission error-

free. This is especially true when there are multiple tokens in the ring, or when there are multiple rings that run on the different channels.

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 this is to put an upper bound on the number of stations that can be in the graph. One solution we adopted based on this idea is maximum number of nodes and ordered list of channel assignments with the distributed channel assignment protocol where a station shift to next channel when it detects a full ring.

10.2 Saturation Throughput Analysis

10.2.1 Introduction

In this section, we concentrate on the analytical evaluation of the WTRP with the assumption of ideal channel conditions and finite number of non-mobile terminals, and all terminals belonging to a single ring. We propose an extremely simple Markov chain model that allows to compute the saturation throughput performance of WTRP. The key approximation that enables our model is the assumption of formed-ring, which means that every node in the medium belongs to a single ring and WTRP works in saturation operating conditions. We followed the approach that is presented in [11], [14] for saturation throughput analysis for IEEE 802.11.

10.2.2 Model

We present the analytical evaluation of the “Saturation Throughput”. This performance figure is defined as the limit reached by the system throughput as the offered load increases, and represents the maximum load that the system can carry in stable conditions [11]. In the analysis, we operate in

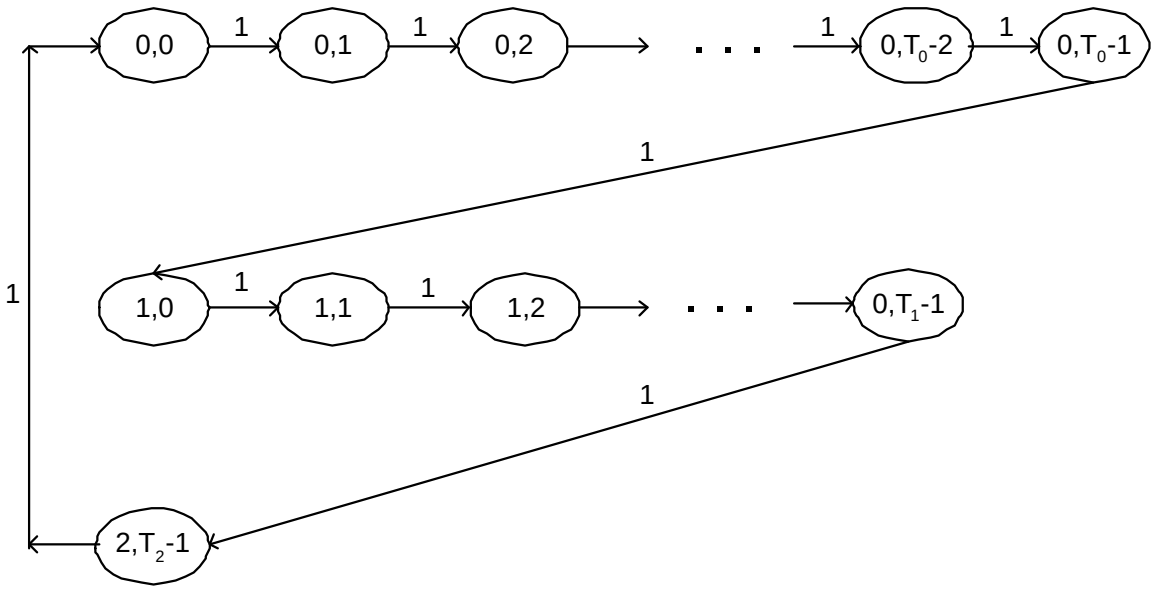

Figure 10.1: Markov Chain model for the Saturation Flow of WTRP.

saturation conditions (See Section 5.3.2), i.e., the transmission queue of each station is assumed to be always nonempty.

The analysis of studying single station with a Markov model and obtaining the stationary probability β that the station transmits a packet in a slot time is enough to express throughput of the ring since when that station has the token the others suspend transmitting (See Figure 3.5).

Consider a fixed number, N, stations in the ring. In saturation conditions, each station has immediately a packet available for transmission, after the completion of each successful transmission. Unlike the 802.11, consecutive packets need not to wait for a random backoff time before transmitting.

Let $c(t)$ be the stochastic process representing the time counter for a given station. A discrete and integer time scale is adopted: t and $t + 1$ correspond to the beginning of two consecutive slot times. Note that this discrete time scale does not directly relate to the system time. In fact, as illustrated in Figure 10.1, one jiffy is adopted as a slot time size j since it may include the token

transmission. Since the value of the token rotation time of the ring depends also on token holding time and token passing time, for convenience let I, R, H be "Idle Time", "Maximum Token Rotation Time", "Token Holding Time" respectively such that $T_0 = I = R - H - 1$, $T_1 = H$, $T_2 = 1$. Token passing time is taken as one slot time. We adopt the notation where $i \in (0, 2)$ is called "state" as *idle* state , *have_token* state and *monitoring* state respectively. Let $s(t)$ be the stochastic process representing the state $(0, 1, 2)$ of the station at time t.

The key approximation in our model is that there is no station outside the ring and only one ring. It is intuitive that this assumption results in more accurate results as long as there is no station waiting to join or leave the ring and as a result, the possible interference from a station is eliminated.

It is possible to model the two-dimensional process $s(t), c(t)$ with the discrete-time Markov chain depicted in Figure 10.1. In this Markov chain, the only non-null one-step transition probabilities are

$$\begin{cases} P\{i, k|i, k-1\} = 1, & k \in (0, T_i) \\ P\{i, 0|i-1, k\} = 1, & k = T_i \\ P\{0, 0|2, 0\} = 1 \end{cases} \tag{10.1}$$

The first equation in (10.1) accounts for the fact that, at the beginning of each slot time, the time counter is incremented. In particular, as considered in the second and third equation of (10.1), when timer expires, the protocol changes its state.

Let $b_{i,k} = lim_{t\to\infty} P\{s(t) = i, c(t) = k\}$, $i \in (0, 2), k \in (0, T_i)$ be the stationary distribution of the chain. We now show that it is easy to obtain the closed form solution for this Markov chain. First note that at

$$b_{i-1,0} = b_{i,0} = b_{0,0} \qquad i \in (0, 2) \tag{10.2}$$

Owing to the chain regularities, for each $k \in (0, T_i)$,it is

$$b_{i,k} = \frac{1}{(I + H + 1)} \qquad i \in (0, 2), k \in (0, T_i - 1) \tag{10.3}$$

Thus, by relation (10.2), all the values $b_{i,k}$ are expressed as functions of the value $b_{0,0}$ which is

$$b_{0,0} = \frac{1}{M} \tag{10.4}$$

We can now express the probability β that a station transmits data in a randomly chosen slot time. As any transmission occurs when the $s(t) = 1$, regardless of the counter.

$$\beta = \sum_{k=1}^{H} b_{1,k} = H \times b_{0,0} = \frac{H}{M} \tag{10.5}$$

However, in general, β depends on the packet size which sometimes affects unused slots in its have token state because the remaining time is not enough to send a new packet. The assumption we adopt is that packet transmission time is a multiple of the token holding time, H. And β probability only involved in fraction of time that the station is transmitting data bits. The packet firing probability is the probability that the station is in state $(1, k)$ $k \in (0, K)$ where K is the number of packets that can be transmitted in one token holding time.

10.2.3 Throughput

Let S be the normalized system throughput, defined as the fraction of time the channel is used to the number of successfully transmitted payload bits. To compute S, let us analyze what can happen in a randomly chosen slot time. Let P_{tr} be the probability that there is at least one transmission in the considered slot time. Since N stations are in the ring, and only one station has right to transmit, the station transmits data with probability

$$P_{tr} = \frac{\beta}{\beta + \frac{1}{M}} \tag{10.6}$$

Let P_f is the probability that station starts firing a packet in the chosen slot time.

$$P_f = \frac{\frac{K}{M}}{\beta + \frac{1}{M}} \tag{10.7}$$

The probability P_s that a transmission occurring on the channel is successful is given by the probability that exactly one station transmits on the channel, conditioned on the fact that at least one station transmits, i.e.,

$$P_s = 1 \tag{10.8}$$

We are now able to express S as the ratio

$$S = \frac{E[\text{payload information transmitted when holding the token}]}{E[\text{length of time}]} \tag{10.9}$$

Since $E[P]$ is the average packet payload size, the average amount of payload information successfully transmitted in a slot time is $P_f P_s E[P]$, since a successful transmission occurs in a slot time with probability $P_f P_s$. The average length of a slot time is readily obtained considering that, with probability $1 - P_{tr}$, the slot time is for token transmission; with probability $P_f P_s$ it contains a successful transmission, and with probability $P_f(1-P_s)$ it contains a collision. Hence, (8) becomes

$$S = \frac{P_s P_f E[P]}{(1 - P_{tr})j + P_f P_s T_s + P_f(1 - P_s)T_c} \tag{10.10}$$

Here, T_s is the average time the channel is sensed busy because of a successful transmission, and T_c is the average time the channel is sensed busy by each station during a collision. j is the

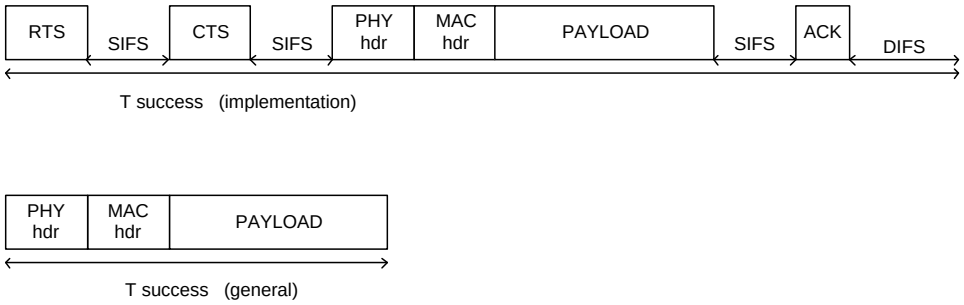

Figure 10.2: T_s for implementation and general

duration of an empty slot time and K is the number of packets a station can transmit during a token holding time.

Let us first consider a system completely that is managed via the basic access mechanism. Let $H = PHY_{adr} + MAC_{hdr}$ be the packet header, and δ be the propagation delay. As shown in Figure 10.2, in the implementation (since we implemented on top of 802.11 card and when broadcasted WaveLAN card operates in basic access mode by disabling RTS/CTS mechanism) and general WTRP case we obtain

$$\begin{cases} T_s^{imp} = & RTS + SIFS + \delta + CTS + \delta + H + E[P] + SIFS + \delta + ACK + DIFS + \delta \\ T_s^{gen} = & H + E[P] \\ T_c^{imp} = & 0 = T_c^{gen} \end{cases} \tag{10.11}$$

The values of the parameters used to obtain numerical results are summarized in Table I. The system values are those specified for the frequency hopping spread spectrum (FHSS) PHY layer [26]. The channel bit rate is assumed to be equal to 1 Mbits. This rate is lower than that of wireless card we used for simulation. This is to compare our result with that of 802.11 presented in [11]. Packet payload size of 8184 bits is considered, which takes one token holding time and that means

WTRP works in saturation state and when it gets a token if has only one packet to transmit ($K = 1$).

packet payload	8184bits
MAC header	272 bits
PHY header	128 bits
ACK	112 bits + PHY header
RTS	160 bits + PHY header
CTS	112 bits + PHY header
Channel Bit Rate	1 Mbit/s
Propagation Delay	1 μs
WTRP Slot Time	488 μs
802.11 Slot Time	50 μs
SIFS	28 μs
DIFS	128 μs
H	17
M	$N * (H + 1)$

Table 10.1: FHSS System Parameters and Additional Parameters Used to Obtain Numerical Results

Figure 10.3[1] shows that the analytical results practically coincide with the intuitive results. WTRP performance is analytically unaffected because unlike the 802.11, P_{tr} does not depend N. This is the property of WTRP that synchronizes the stations and allow only one station to transmit one at a time. On the other hand, 802.11 has collision probability. The figure 802.11 basic represents the situation when the RTS/CTS mechanism is off. When there is a collision in the basic access, the whole packet is lost and more time for collision is wasted compared to the RTS/CTS mechanism where only RTS packet is collided. At a first glance, it might seem that the throughput performance of the WTRP does not depend on the number of stations. In fact, there is a tradeoff between token holding time and maximum rotation time. In the analysis, we keep the token holding time constant and increased the maximum token rotation time that enable a station to transmit the packet even if the number of stations increase.

Theoretically, this does not affect the performance, but in reality, if there is a failure to transmit at the first time, or if it exits the ring, the station waits longer and this decreases the performance.

[1] 802.11 values taken from the proof presented in [11], The details of the proof can be found there.

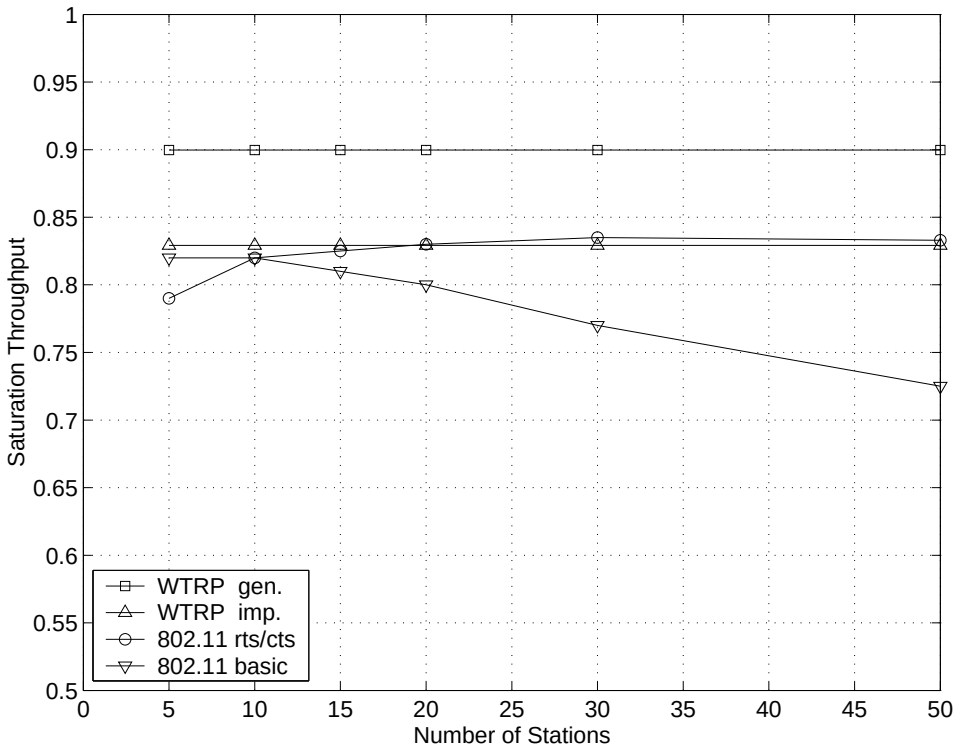

Figure 10.3: Saturation Throughput

On the other hand, if we keep the maximum token rotation time constant and decrease the token holding time, the ring will be as responsive as before while number of stations increases but the station capacity decreases because ratio of payload bits over transmitted bits will decrease.

10.2.4 Conclusion

We presented a simple analytical model to compute the saturation throughput performance of the WTRP. Our model assumes a constant Token Holding Time even if the number of stations increases. This results in a constant throughput for WTRP even as the number of stations increases in the medium. The better performance is achieved because WTRP eliminates the collision probability by sharing the slots among the stations and collision probability increases in IEEE 802.11 when the number of active stations increases.

10.3 Summary

We proved that when transmission losses and topological changes of the graph stop at time t, and stations do not go into the *OFFLINE* state voluntarily, then the algorithm will come to a stable state in which all stations will belong to a ring at time $s > t$. We also proved using a simple Markov model that in a stable environment, when the station operates in the *saturation* conditions, increase in the network size does not degrade WTRP performance since spectrum usage is fixed and collision probability is zero.

Chapter 11

Performance Analysis

11.1 Performance Analysis

We aim to show the effectiveness of WTRP in terms of “bound on latency”, where each station gets right to transmit in a fixed time, “fairness”, where each station consumes equal amount of bandwidth, “robustness”, where ring recovers from node failures without collapsing, “responsiveness”, where ring responses fast enough to medium changes and “medium utilization” where ring utilize the capacity of the channel at maximum. Performance of WTRP is analyzed for Constant Bit Rate (CBR) and FTP traffic.

11.1.1 Scenario

The testbed is constructed by the laptop computers (2 Dell Inspiron 5000, 1 Dell Inspiron 8000, 6 Dell Latitude C600). Network structure is shown in Figure 11.1. S_i , $i \in (0, 7)$ use Lucent WaveLAN Silver Cards (2Mbit/s), and L_1, L_2 have Lucent Orinoco Gold Cards(11 Mbit/s). In the scenario, S_i and L_i are called “Sender” and “Listener” respectively. Listeners always use 802.11 as the MAC protocol and senders use WTRP or 802.11. Letter N represent senders and one listener

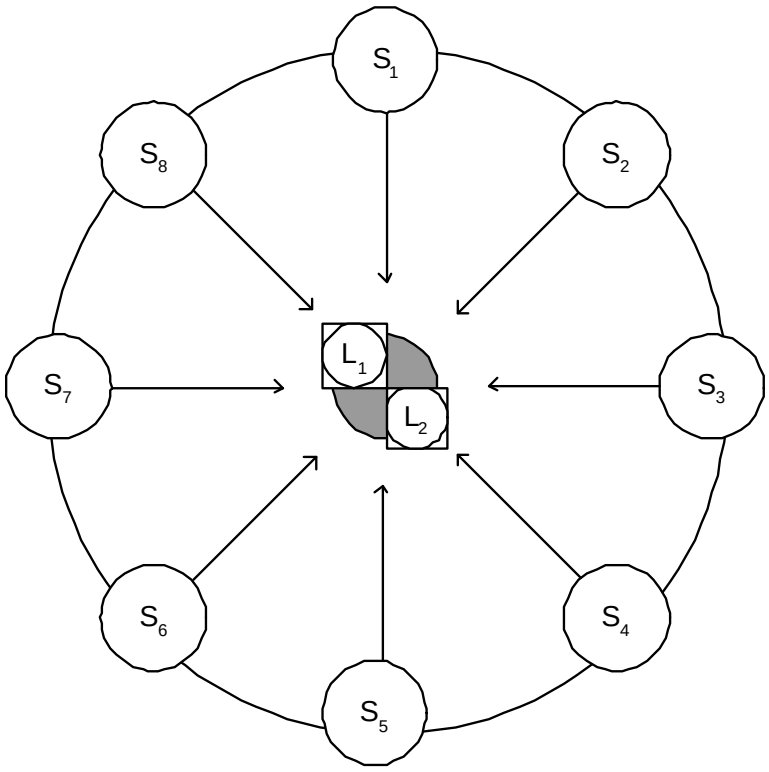

Figure 11.1: Simulation Scenario

is assigned for each 3 sender. We choose this scenario in order to eliminate the overhead that is introduced because of 802.11 protocol embedded to the WaveLAN card (See Section 9.5.8).

11.1.2 Optimum Operating Frequency

WTRP performs better in certain situations because of its deterministic property. We analyzed the performance of WTRP under variable packet rate and packet size. For Figures 11.2 and 11.3, under a constant packet size (100bytes) and we changed the packet generation rate. One can see from Figures 11.2 and 11.3 that WTRP performs better than the 802.11 when the packet inter arrival time is around MTRT. In the high loaded region where packet generating rate is less than MTRT, WTRP shows fluctuations because of its fixed THT and packet queue size. In the scenario, THT only allows a node to transmit one or two packets. This fills the buffer and consequently, stops the upper layer

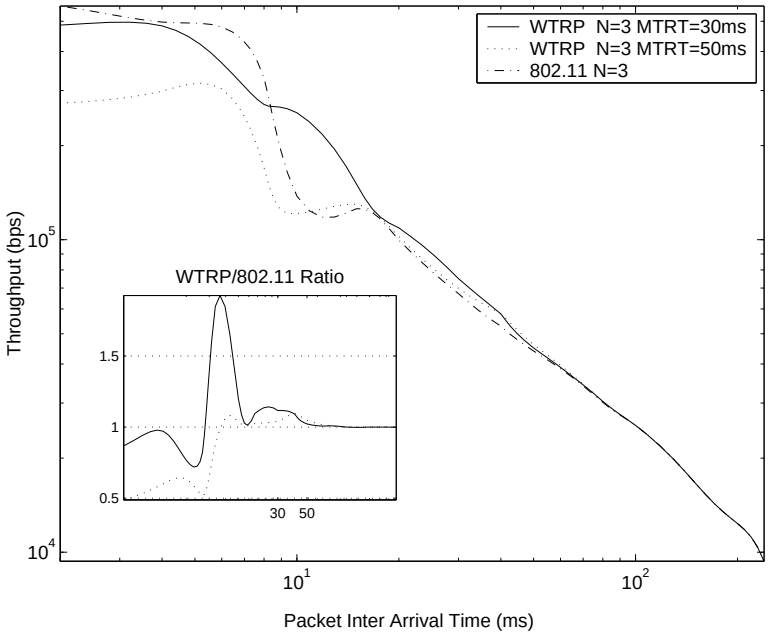

Figure 11.2: Variable Packet Generating Rate - 3 Senders

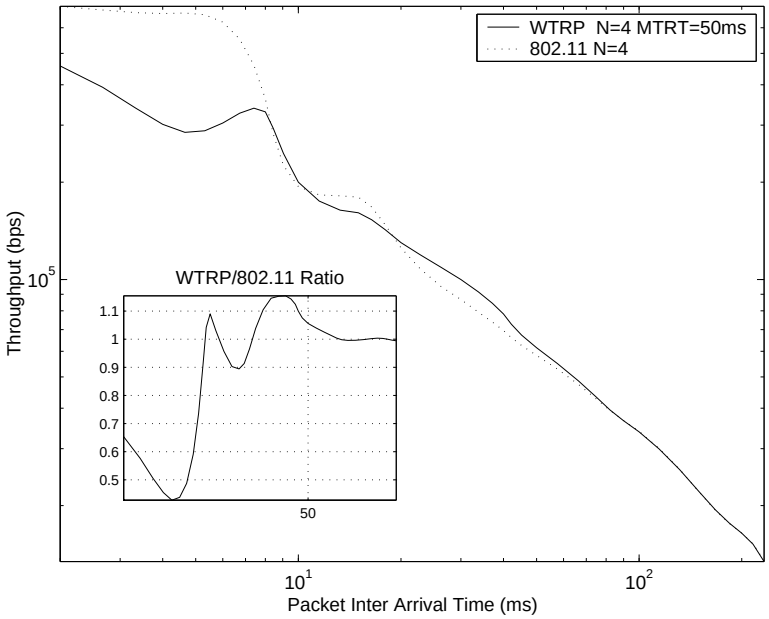

Figure 11.3: Variable Packet Generating Rate - 4 Senders

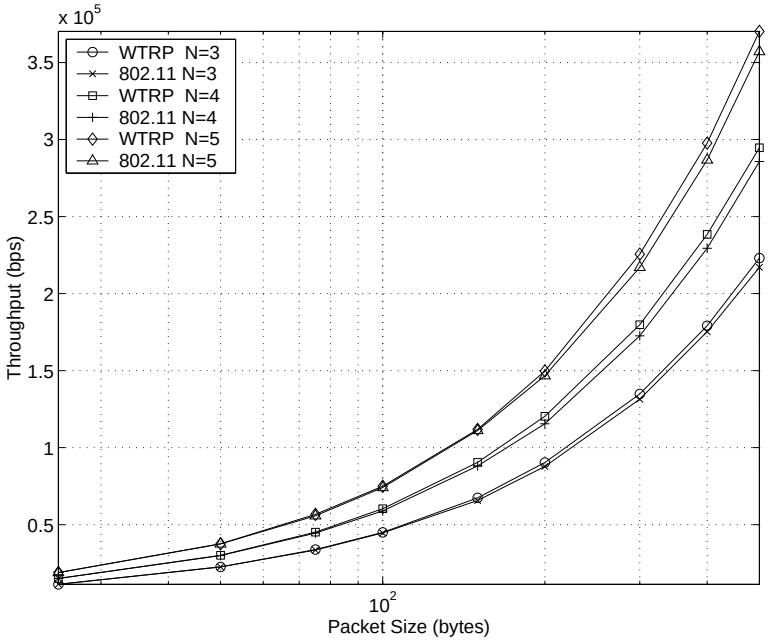

Figure 11.4: Variable Packet Size - MTRT = 30ms

with logical link control. These cause delay and degrades the performance. Peaks in the figures indicate that node transmits more packets than normal in a short period.

Around MTRT, WTRP only sends one packet and one token and performance improvement compared to 802.11 comes from zero collision probability. On the other hand, 802.11 suffers from collision. When the 802.11 node detects a collision, it backoffs and waits for the backoff time. In the low overloaded region, when the packet generating time is bigger than 75ms, WTRP and 802.11 performance are almost exactly the same because the collision probability is reduced with the rare medium access.

We also tested the protocol under variable packet size as seen in Figures 11.4 and 11.5. Under constant packet generation rate which is equal to one MTRT, we increased the packet size. WTRP performance stayed constant as expected but 802.11 showed slightly unstable behavior with the increase in packet size. This is because, packets large in size occupy the channel for longer time, and the station in backoff stage samples and senses the channel. In these sampling times, station

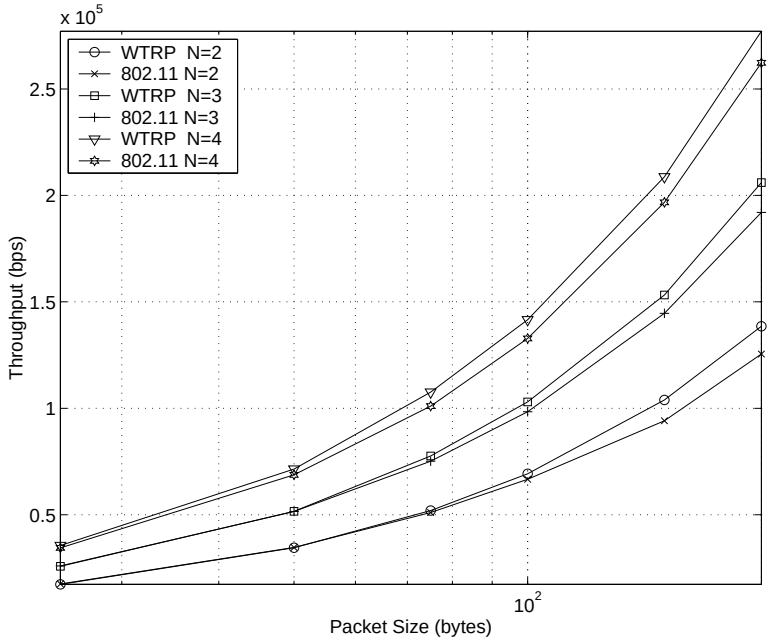

Figure 11.5: Variable Packet Size - MTRT = 50ms

may have chance to detect empty channel even if the channel is not empty since packet inter arrival time is longer than the sensing time.

11.1.3 Bound on Latency

From Figures 11.6 and 11.7, one can see that token rotation time distribution shows almost zero variance when THT is set to 1.5ms. This means that every station gets chance to transmit in a specific bounden time. Token rotation time increases less than one token holding time when a new node is added and variance still remains close to zero that means that increase in network size does not cause instability to WTRP.

Latency measurement can also be observed by examining the jitter of packet arrivals. Figure 11.8 shows the distribution of packet inter-arrivals of the listener when there are 3 senders. Each sender sends 100bytes in every 50ms and MTRT is set to 50ms. Distribution shows that distribution is dense at 50ms even if there is overhead of 802.11 Card in terms of introducing backoff interval.

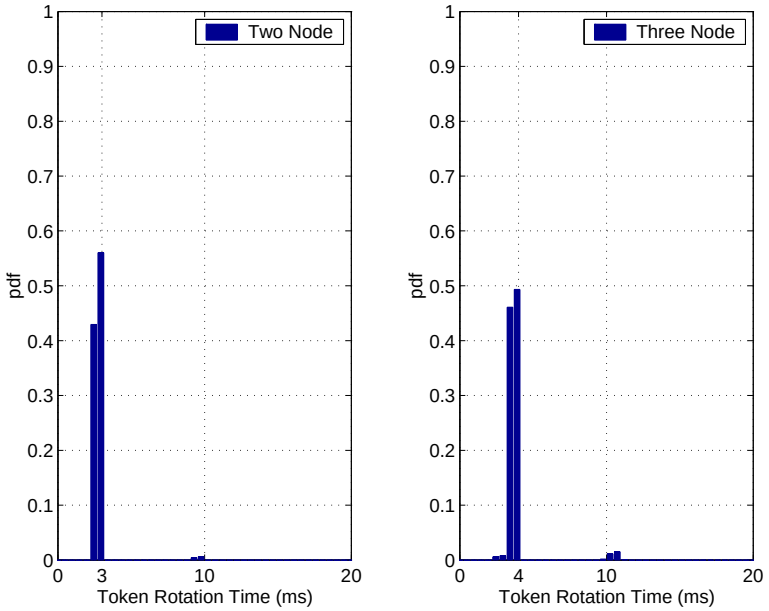

Figure 11.6: Token Rotation Time Distribution - 2 and 3 Senders

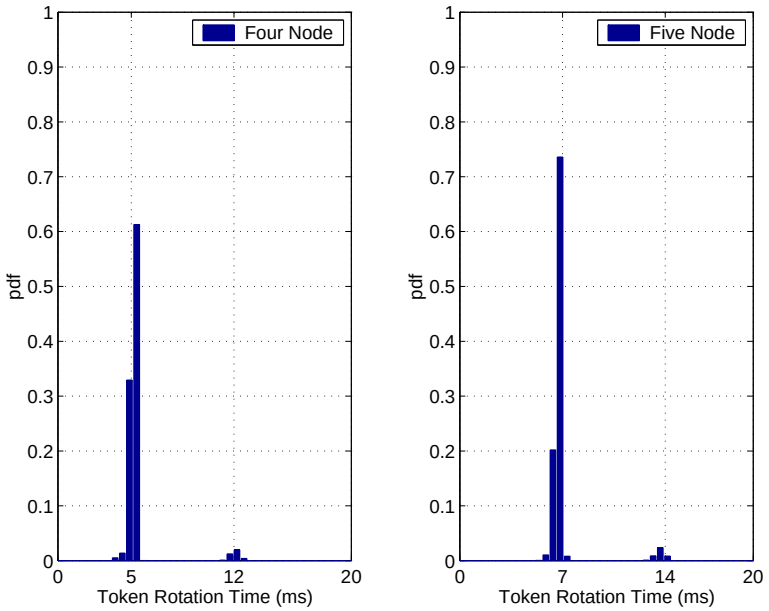

Figure 11.7: Token Rotation Time Distribution - 4 and 5 Senders

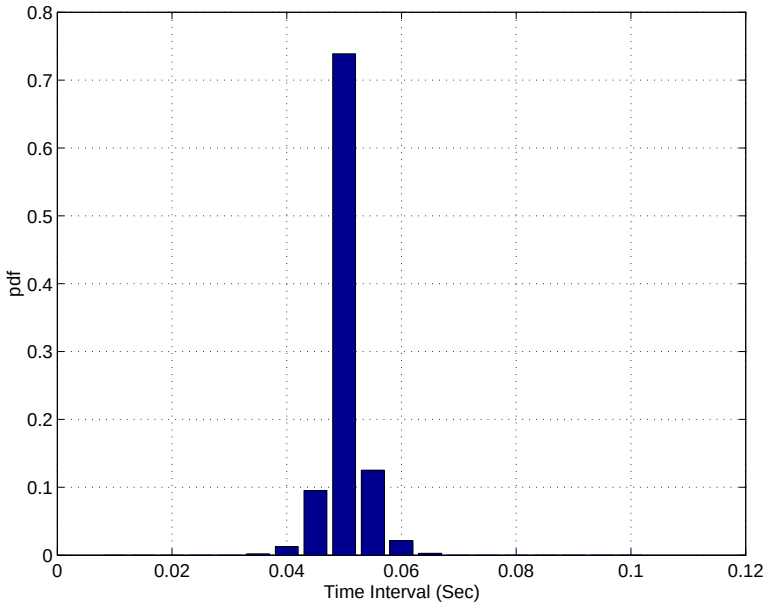

Figure 11.8: PDF of Jitter

11.1.4 Robustness and Responsiveness

Recovery from a node failure is our robustness measure. Ring should not collapse when a node leaves or stops. In the same way, ring should add a joining node in a stable way. When there are 5 nodes in the medium and one of them turns on and off WTRP every second, from upper graph of Figure 11.9 where " . " represents *solicit_successor* token, one can see that WTRP handles leaving and joining in a robust way since number of nodes in the ring never drops below 4. Lower graph of Figure 11.9 is a trace of ring formation when all 5 nodes turn on at the same time. Number of nodes in the ring is monotonically increasing and reaches maximum.

Responsiveness of the system can also be observed in Figure 11.9. WTRP is responsive to single node failures and recovers quickly. The responsiveness of the system could be increased by reducing the latency caused by the token ring protocol computation or using a wireless network interface card with a higher bandwidth.

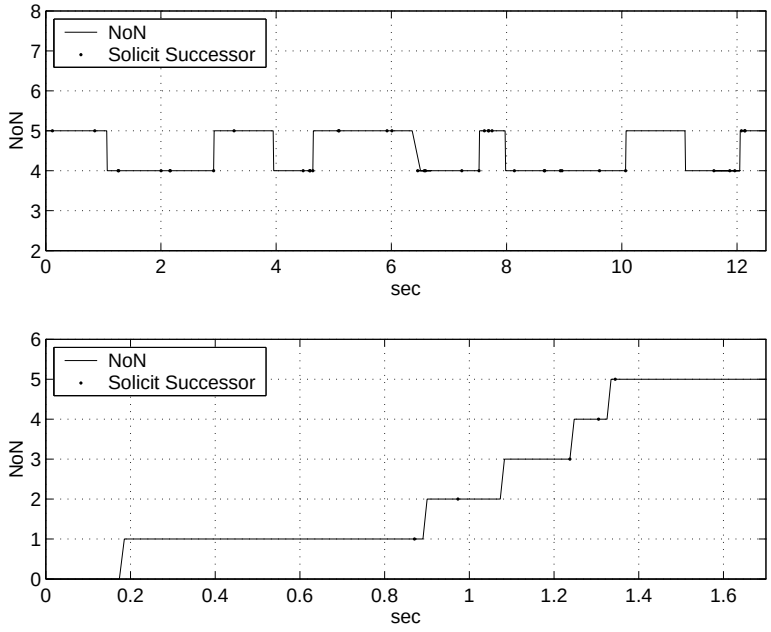

Figure 11.9: Robustness and Responsiveness

11.1.5 Fairness

WTRP provides fairness in terms of equal amount of channel usage. On the other hand, 802.11, suffers from fairness since the station that made the last successful transition is favored [12]. We observed the instantaneous throughput when there are 3 senders and 1 listener, and packet size and generation rate is 100bytes and 50ms respectively. First graph of Figure 11.10 is the WTRP trace. One can see that each of three stations almost has the same throughput, which is constant throughout the observation. One of the stations is suspended by the others in 802.11 where it is represented in the second graph of Figure 11.10. We tested the scenario many times and starting times of transmission is randomly chosen in order to include performance results of movement since nodes may come early or late to the medium. Third graph shows the distribution of the standard deviation of instantaneous throughputs . Increase in the deviation means increase in the unfairness and deviation of WTRP is closer to zero than that of 802.11. Upper graph of Figure 11.9 also points

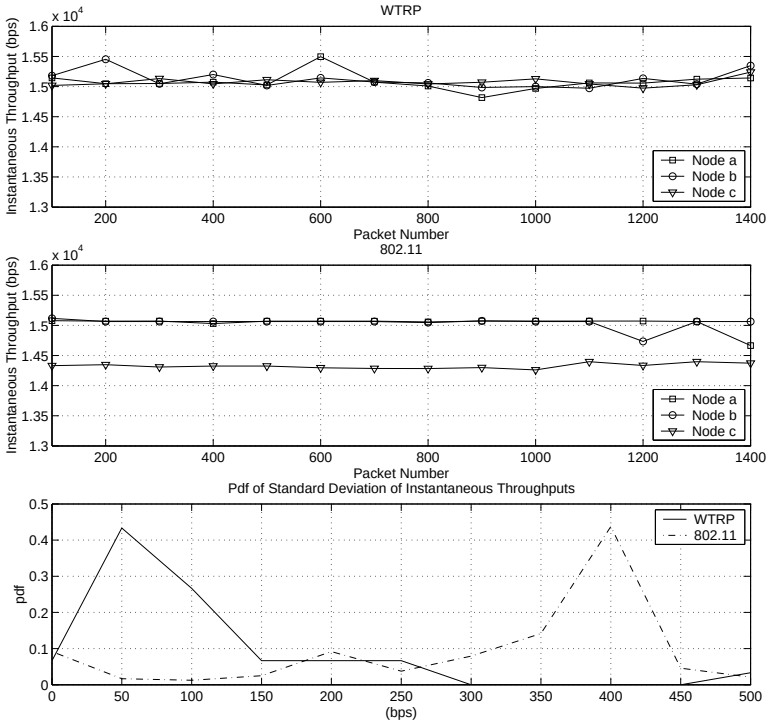

Figure 11.10: Instantaneous Throughput

to the fairness behavior of WTRP since even if a station comes late, station gets immediate right to transmit when it belongs to the ring.

11.1.6 Network Size

In a wireless network without proper organization, each node causes an increase in collision probability. As a result, network size matters to protocols that suffer from collision such as 802.11. We increase the number of nodes in the network and each node sends 100bytes with 50ms packet generation rate. As it can be inferred from the Figure 11.11, WTRP performs better than 802.11 and the performance difference increases as the network grows in size. We expect higher difference, when the WTRP is implemented on top of a card that does not use 802.11 protocol. Test is also performed with Poisson packet generation rate when the parameter is 50ms. Results show that throughput is higher in WTRP than 802.11 but difference of their performance varies. Since Poisson behavior

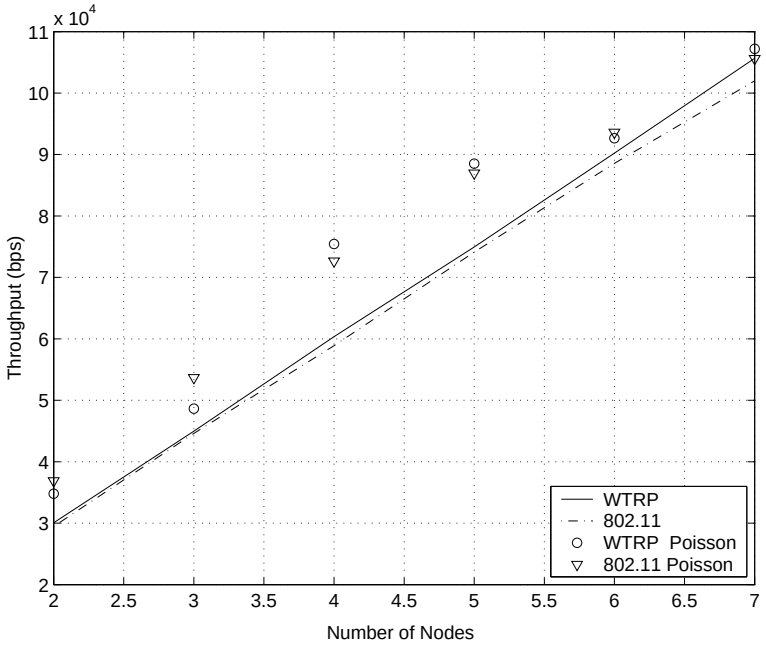

Figure 11.11: Number of Nodes vs Throughput

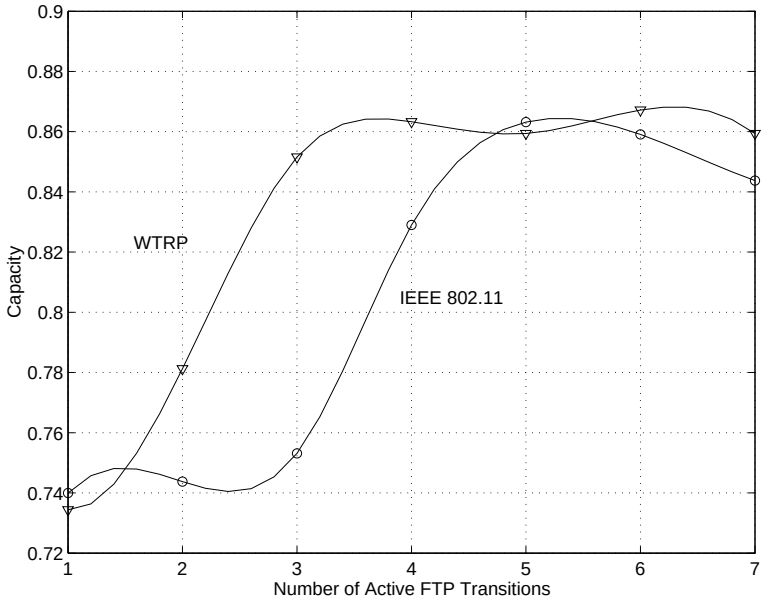

Figure 11.12: Saturation Performance

does not affect WTRP but affects 802.11 positively or negatively.

Under heavy load, the token ring implementation performs better than IEEE 802.11. This is shown in Figure 11.12. In the figure, the aggregate FTP bandwidth is plotted against the number of simultaneous FTP transfers. Both cases involved number of nodes equal to the number of simultaneous FTP transfers. The FTP was done as follows. For the case of two simultaneous transfers, one transfer goes from station 1 to station 2 and the other from station 2 to station 1. For the case of three simultaneous transfers, the transfers are from 1 to 2, from 2 to 3, and from 3 to 1.

In Figure 11.8, we observed a decrease following a increase in IEEE802.11. The decrease in the throughput is expected since the number of collisions increases in a CSMA medium access control and after the saturation point, performance of IEEE802.11 degrades [14].

The performance intuitively should be constant in Wireless Token Ring case but improves in the simulation when going from 1 to 3 simultaneous transfers. This can be explained as follows. Since for all trials in Figure 11.8, the *Maximum Token Rotation Time* varies, the nodes can not operate at saturation point with few nodes in the network. As a result, the nodes can not use all of the capacity. The need for retransmission of the data due to collisions is eliminated since WTRP reduces collisions.

11.2 Summary

WTRP in its current implementation is disadvantaged relative to the original 802.11 driver because WTRP is implemented on top of 802.11 in DCF mode, incurring all the overhead that is associated with 802.11 plus the overhead from the WTRP. The overhead is the increased computation time and packet header size.

We observe that WTRP has optimum operating condition when the packet generation rate is equal to the token rotation time and as long as token holding time is enough , increase in packet size

does not affect its performance. WTRP responses in a quick and robust way to the possible results of mobility. We find that WTRP achieves bounded delay and distributes bandwidth fairly among the stations compared to 802.11. Finally, since collision is affectively reduced, WTRP is not affected from the network size as long as each station use their THT at maximum.

Chapter 12

System Extensions

12.1 Introduction

WTRP is a new research topic and can be extended in various ways. In this chapter, we introduce possible extensions to the WTRP and cite design issues left open.

12.2 Hybrid Schemes

Hybrid Schemes (See Figure 12.1) are the schemes where the centralized approach that uses the star topology and the distributed approach that uses the ring topology are hierarchically mixed together. All slave nodes need to have a connection with the master and master nodes are connected to each other via WTRP. Packets are routed one-hop to the master with centralized protocol, multi-hop with WTRP to the master of the destination and finally, one-hop to the destination with centralized protocol.

12.3 Token Chain

Token chain (See Figure 12.1) creates the ring by bi-directional token passing. Token coming from the predecessor is passed to the successor and token coming from the successor is passed to the predecessor. The nodes at the edges have the same station as the predecessor and the successor. A station gets right to transmit twice in one token rotation and consumes fixed amount of time in total. In these structure, it is advantageous to communicate with only one node for the joining node to become a ring member. However this is disadvantageous for the ring since permission to send invitation belongs to the edge nodes now. The joining advantage leads to a ring that can be formed in a wide area. These kind of network structures is suitable where the node distribution among the area is rare.

12.4 Sensor Networks

Sensor networks is a special kind of network where the destination of all packets is the access point to the backbone network. We can use token ring or token chain in sensor networks where low performance is enough. All the nodes will belong to one ring and only one node will be transmitting at a time. The energy consumption at the nodes can be decreased by putting nodes into sleep while they do not have the token since the nodes predict their possible token reception time by considering the number of nodes in the ring and token holding time of each node. Each node stores the data packets coming from its predecessor and transmits all the packet to its successor. Access point may also involve in multiple rings and may schedule giving right to transmit each ring by transmitting the token in order.

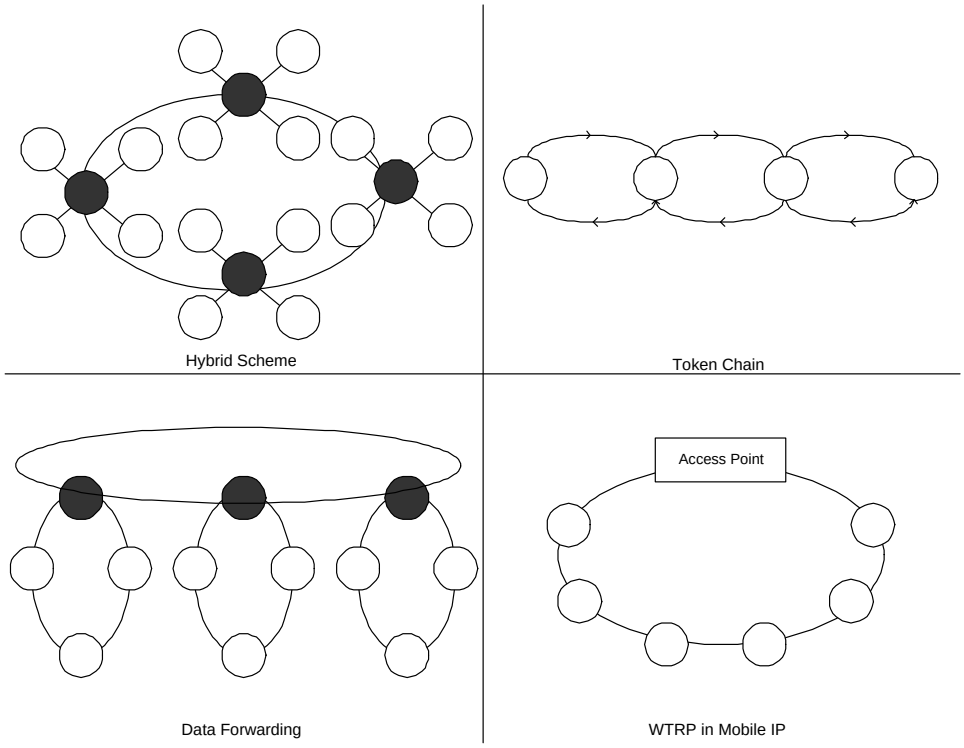

Figure 12.1: System Extensions

12.5 Data Forwarding

WTRP clusters stations into multiple rings. Different channels are assigned to different rings in order to avoid interference. There are several possibilities for the routing scheme. We will introduce one of them in which connected network can be created by "gateway" nodes that belong to two rings, one of them is its own ring and the other is the ring that "gateway" nodes create, named "backbone ring". Routing is divided into intra-ring routing, routing between nodes that belong to the same ring and inter-ring routing, routing between gateway nodes. Gateway node keeps the information of the nodes. If the destination nodes belongs to another ring, packets are forwarded to the gateway node and gateway node forwards it to its successor in the backbone ring (See Figure 12.1). When the gateway node of the destination gets the packet, it sends them to the destination. Backbone ring can also be formed with token chain.

12.6 Extension to Mobile IP

Mobile IP has been designed within the IETF to serve the needs of the increasing population of mobile computer users who wish to connect to the Internet and maintain communications as they move from place to place. WTRP provides efficient bandwidth share to the users and provides solutions to the early detection of attachment points and movement detection of mobile node problems. Both problems brings concerns in Mobile IP research in order to decrease hand-off time.

In the Mobile IP design with WTRP (See Figure 12.1), resource information can be injected to the token signal. This token may also contain the *care_of_address* of the attachment point, which enables the mobile node to detect the attachment point early without waiting for beacon signals. As a result, the time that takes for a node to communicate with the base station is equal to the time to get into the ring. When the mobile node lost the connectivity to the attachment point, attachment point

can easily detect the movement by checking its connectivity table and execute a smooth hand-off algorithm. Papers show that hand-off time in Mobile IP is dominated by the beacon period of the attachment points [17]. Designing Mobile IP with WTRP may eliminate the beacon necessity and inject beacon information to the periodic token signal.

12.7 Multimedia

Nodes in a ring may access different kind of resources. Resources can range from printer to an internet access. The resource information can be conveyed by injecting to the token. If a mobile node requires a resource, it searches for a ring that has the appropriate resource. As an analogy, WTRP and IEEE 802.11 can be thought as ”Guest enters the appropriate house by checking the name tag on the door and finds the right house” and ”Guest enters the appropriate house by entering each house and asking the person to the households” respectively.

12.8 Summary

We presented the possible extensions to the WTRP. WTRP idea is suitable for many wireless networks. Networks can leverage token passing identity to use as an information carrier and WTRP is compatible to work in a hierarchy with other MAC layer protocols.

Chapter 13

Conclusion

WTRP is inspired from IEEE802.4 token bus protocol. The development of IEEE802.4 was initiated by people from General Motors and other companies interested in factory automation [16][25]. Features such as bounded latency and robustness against multiple node failures are some of the reasons for this choice. In addition to bringing the same bounded latency and robustness features to the wireless medium, WTRP also manages ad hoc topologies.

We deployed the WTRP idea in three implementations: Simulator, User-Space and Kernel Implementations. We created a sharable library and functions that enable the implementations use the library. Simulator implementation is a simulation tool for WTRP and extendable to any other network module. It provides interfaces to simulate the implementation code in large wireless networks. User-Space implementation is a platform independent implementation in application layer. This implementation is useful under a controlled application environment when utilized on top of an arbitrary network interface card. Kernel implementation is a Linux link layer module built on top of the IEEE802.11 in DCF mode. It incurs all the overhead that is associated with IEEE802.11. Despite the overhead, we have found that WTRP performs well or even better under heavy load.

We have designed a protocol that is fast in terms of recovery and efficient with zero collision

probability. 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 a valuable support for real time applications, emergency management, and automated highway systems.

Bibliography

[1] Berkeley Web Over Wireless Group, *http://wow.eecs.berkeley.edu*.

[2] D. Lee, *Wireless Token Ring Protocol*, Master's Thesus at Berkeley, 2001.

[3] D. Lee, B. Dundar, M. Ergen, A. Puri, *Socket Interface for User Applications in WOW Project*, http://wow.eecs.berkeley.edu.

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

[5] 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.

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

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

[8] M. Hannikainen, J. Knuutila, A. Letonsaari, T. Hamalainen, J. Jokela, J. Ala-Laurila, J. Saarinen, *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] I. F. Akyildiz, J. McNair, C. L. Martorell, R. Puigjaner, Y. Yesha, *Medium access control protocols for multimedia traffic in wireless networks*, IEEE Network, vol.13, (no.4), IEEE, July-Aug. 1999. p.39-47.

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

[11] G. Bianchi, *Performance Analysis of the IEEE 802.11 Distributed Coordination Function*, IEEE Journal on Selected Areas in Communications, Vol.18, No. 3, MARCH 2000.

[12] Y. Wang, *Achieving Fairness in IEEE 802.11 DFWMAC with Variable Packet Size*, Global Telecommunications Conference, 2001.

[13] T. Pagtzis, P. Kirstein, S. Hailes, *Operational and Fairness Issues with Connection-less Traffic over IEEE802.11b*, ICC 2001.

[14] G. Bianchi, *IEEE 802.11-Saturation throughput analysis*, IEEE Commun. Lett., vol. 2, pp. 318-320, Dec. 1998.

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

[16] A. Tanenbaum, *Computer Networks*, Prentice Hall, New Jersey, 1988.

[17] M. Ergen, S. Coleri, B. Dundar, A. Puri, P. Varaiya, *Fast Handoff with GPS Routing in Mobile IP*, IPCN Paris April, 2002.

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

[19] Bluetooth, *http://www.bluetooth.com*.

[20] Network Animator, *http://www.isi.edu/nsnam/nam*.

[21] Network Simulator, *http://www.isi.edu/nsnam/ns*.

[22] WaveLAN, *http://www.wavelan.com*.

[23] The Teja Technical Reference, *http://www.teja.com*.

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

[25] W. McCoy, *RFC1008: Implementation Guide for the ISO Transport Protocol*, 1987.

[26] *IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Nov.1997. P802.11.

[27] Yankee Group, *http://www.yankeegroup.com*.

[28] ARS, *http://www.ars1.com*.