UC Berkeley
Research Reports

Title

A Token-Ring Medium-Access-Control Protocol with Quality of Service Guarantees for Wireless Ad-hoc Networks

Permalink
https://escholarship.org/uc/item/22c3f97s

Authors
Attias, Roberto
Lee, Duke
Puri, Anju
et al.

Publication Date
2001-03-01

eScholarship.org Powered by the California Digital Library
University of California

This paper has been mechanically scanned. Some errors may have been inadvertently introduced.

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

A Token-Ring Medium-Access-Control Protocol with Quality of Service Guarantees for Wireless Ad-hoc Networks

Roberto Attias, Duke Lee, Anju Puri, Starvros Tripakis, Raja Sengupta, Pravin Varaiya
University of California, Berkeley

California PATH Research Report
UCB-ITS-PRR-2001-7

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.

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

Report for MOU 329

March 2001

ISSN 1055-1425

A Token-Ring Medium-Acces-Control Protocol with Quality of Service Guarantees for Wireless Ad-hoc Networks*

Roberto Attias, Duke Lee, Anuj Puri, Stavros Tripakis,
Raja Sengupta and Pravin Varaiya
WOW! Group/PATH LAB
University of California at Berkeley

November 10, 2000

*This research is funded by PATH-CALTRAN MOU329, ONR-GRANT 23083 (Distributed Autonomous Agent Networks), and ONR-GRANT N00014-99-1-0695

1 Executive Summary

This report describes the design and implementation of a wireless token bus protocol for local area networks. This is the second wireless token passing protocol that has been jointly designed by the PATH program and the faculty and students of the EECS department at UC Berkeley.

The first wireless token bus protocol, designed by Duke Lee and Professor Pravin Varaiya, was successfully implemented to provide the wireless network required by an automated vehicle platoon. The token passing mode of wireless medium access control was chosen to provide the delivery time guarantees required by the safety critical control data transport required by the platoon. This protocol is currently also being used for safe automated vehicle merging maneuvers.

The token bus protocol described in this report represents a significant advance over the first version. The earlier protocol did not permit the wireless radios to dynamically join and leave the network while still maintaining the quality of service for those remaining in the network. This version does. The earlier version also worked only when each wireless radio in the network was within range of every other radio in the network at all times. This version relaxes this requirement. Since mobile wireless LAN's frequently encounter hidden or exposed terminal configurations, these advances greatly enhance the usefulness of the protocol.

We intend that this wireless token bus protocol support the Vehicle Automation Demonstration to be held in 2002 in California and the autonomous agent networking needs of the Berkeley BEAR UAV-UGV (Unmanned Aerial Vehicle, Unmanned Ground Vehicle) testbed. We also hope that this protocol will be picked up by ITS (Intelligent Transportation System) network builders to build wireless networks for ITS Dedicated Short Range Communications (DSRC). The DSRC community has recently been seeking protocols to coordinate multiple access points, possibly operated by different jurisdictions, within the same frequency channel. This protocol is well suited for such coordination. We also believe that this protocol will be useful for home or enterprise networking.

This report provides a full specification of the wireless token bus protocol. It specifies procedures for radios joining the network, leaving the network, detection and removal of multiple tokens, recovery from node or link failure, and the generation of unique ring identifiers. A formal proof is provided for the correctness of these procedures. The report includes a formal specification in the Teja design environment. The Teja tools also generate real-time code from this specification. Thus the specification in this report corresponds to an implementation that we plan to release shortly. The report also presents a generalized wireless token bus protocol that should be a seen as a blueprint for a future, third version of this protocol.

We are grateful to our sponsors, CALTRANS and the Office of Naval Research, for their support of this two year effort.

Contents

1 Executive Summary **2**

2 Introduction **5**
2.1 Ad-hoc Networks 5
2.2 Applications 5
2.3 Physical Layer Model 5
2.4 Quality of Service Guarantees 5
2.5 Motivation 6
2.6 The Wireless Token Bus Protocol 6

3 Overall Design and Architecture **6**

4 A High-Level View of the Wireless Token Bus Protocol **8**
4.1 Normal Operation 8
4.2 Abnormal Conditions 10
4.3 Other features of WTBP 12

5 Detailed Operation of the Wireless Token Bus Protocol **13**
5.1 WTBP station information 13
5.2 Timers 14
5.3 WTBP packet formats 14
5.4 Finite-state-machine specification 16
5.4.1 In Ring macro-state 17
5.4.2 Waiting to Join macro-state 20
5.4.3 Idle macro-state 22
5.5 Connectivity Caches 22
5.5.1 Need for topology knowledge 22
5.5.2 The approach taken in WTBP 23

6 Proof of Stabilization **24**

7 The Generalized Wireless Token Bus Protocol **27**
7.1 High-level description of the generalized protocol 27
7.1.1 No requirements on topology 28
7.1.2 Format of solicit-successor token 28
7.1.3 Per-hop implicit acknowledgments 30
7.1.4 Close-route and close-ring operations 30
7.1.5 Other details 31
7.2 Properties of the generalized protocol 31
7.2.1 Stability 31
7.2.2 Connectivity 33
7.2.3 Hop-count of token routes 33

8 Support for Data Forwarding **34**

A Application Domain: Vehicle Communication **35**

B CSMA based schemes **36**

C Comparison with the IEEE 802.4 Token Bus Protocol 37
C.1 The Token Bus MAC protocol 37
C.2 Comparison with IEEE 802.4 38
C.2.1 Joining a Ring 38
C.2.2 Claiming the Token 39
C.2.3 Passing the Token 40

D Detailed Description of the Protocol 40
D.1 A Hybrid Automaton Model 41
D.2 Hierarchical specification 42
D.3 High-level automaton 44
D.4 Low-level automata 46
D.4.1 Offring Automaton 46
D.4.2 Inring Automaton 47
D.4.3 Enter Automaton 49
D.4.4 Pass-Token Automaton 51
D.4.5 Close-Ring Automaton 52
D.4.6 Claim-Token Automaton 53
D.4.7 Have_ Token Automaton 54

2 Introduction

We are interested in a medium-access-control (MAC) protocol for applications running on top of wireless ad-hoc networks and requiring some type of quality-of-service guarantees.

2.1 Ad-hoc Networks

Ad-hoc networks are networks where participating stations can join or leave the network at any moment in time. Also, stations are allowed to move. The ad-hoc nature of these networks implies the following:

- The physical layer of the network must be wireless (due to mobility).
- The topology of the network is changing dynamically: nodes (representing stations) and links (representing the fact that two stations are within range) are added or removed as stations join, leave or move.

2.2 Applications

Ad-hoc networks are needed to provide the communication infrastructure for applications involving the distributed coordination of autonomous agents. Examples of such applications are the automated highway project [6] (see appendix A for more details) and the Berkeley Aerobot project [4]. Apart from autonomous-agent systems, it can be expected that ad-hoc networks will play an important role in design of wireless networks for mobile internet access [5].

2.3 Physical Layer Model

As we said, the physical layer must be wireless, to support mobility of nodes. This implies the following:

- Bandwidth is limited.
- The channel is shared among many stations

In this paper, we also make the following assumption: a packet is either lost or delivered intact, that is, if the packet is corrupted, the error detection mechanism is adequate for detecting this.

2.4 Quality of Service Guarantees

For the applications we are interested in, the network is required to deliver certain types of data in real-time: for instance, in the context of the automated highway project, every vehicle periodically send its speed to its successor vehicle. Therefore, *under normal operating conditions,* the MAC protocol must provide the following guarantees:

1. a minimum throughput must be guaranteed for each station;
2. the medium-access time for each station must be bounded.

The medium-access time is the delay from the time a station wishes to transmit data until the time it actually manages to transmit the data successfully. By "normal operating conditions", we mean "fewa" packets are lost due to noise or other phenomena such as multi-path.

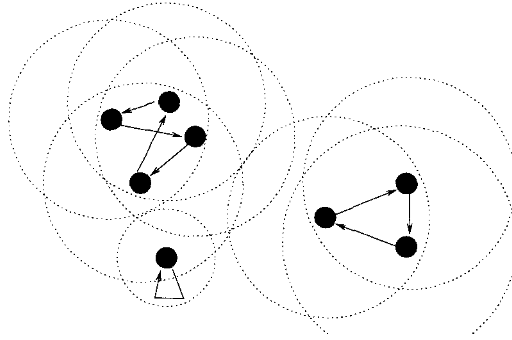


Figure 1: Stations arranged into multiple rings

2.5 Motivation

Current wireless MAC protocols such as the IEEE 802.11 (ad-hoc mode) and the ETSI Hiperlan do not provide the 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. Some more considerations regarding the inadequacy of CSMA-based schemes are given in section B.

Other architectures such as the base-station mode of 802.11 or the master-slave scheme of Bluetooth have disadvantages like restrictions on topology (all stations connected to the central point), single point of failure, or limited efficiency (going through the central point).

2.6 The Wireless Token Bus Protocol

We call our protocol the *Wireless Token Bus Protocol* (WTBP). WTBP is inspired from the IEEE 802.4 Token-Bus protocol. Token-ring protocols have many desirable properties:

1. they achieve high medium utilization under high load,
2. they distribute throughput in a flexible and fair manner among stations,
3. they provide bounds on medium-access time.

Still, there are problems to be solved when adapting a MAC protocol designed for wirelined networks to the wireless ad-hoc case. In this paper, we describe the design of our protocol to cope with these issues. In appendix C, we discuss more specifically the differences of our protocol with respect to IEEE 802.4 Token-Bus, our extensions to the later, and the motivations for these extensions.

3 Overall Design and Architecture

The stations in the network are organized into multiple *logical rings* as shown in Figure 1. By "logical" we mean that the structure of the ring is not directly related to the *physical connectivity* of the stations (which depends on their transmission ranges). The arrows in the figure represent the (logical) successor of each station. The dotted circles represent the transmission range of each station.

We assume there are multiple channels available for transmission so that transmissions from neighboring rings do not conflict with each other. Within each ring, there is a *token*. The station in the ring that has the token transmits. After transmitting, the station passes the token to its successor in the ring. To move from one ring to another ring, a station *leaves* one ring and *joins* another ring.

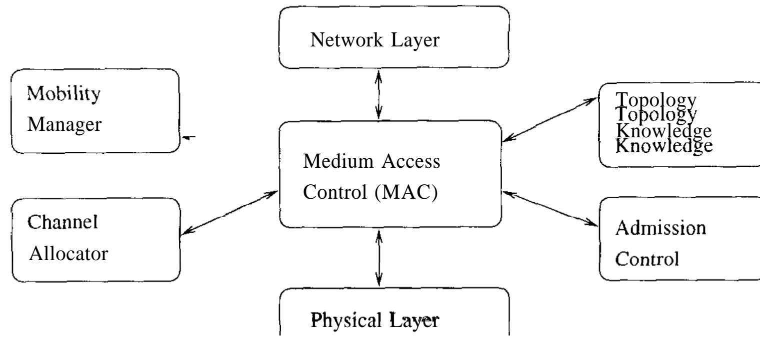


Figure 2: Organization of the protocol

We also assume that each station has a unique MAC address.

The overall architecture is shown in Figure 2. The main components are the MAC component, the topology manager, the channel allocator, the topology knowledge component and the admission control component. Although the main focus of this paper is the MAC component, we briefly describe the functions of the other components and how they may interact with the MAC component.

MAC

The MAC protocol is responsible for organizing a single ring. In doing so it performs the following functions:

1. It ensures that each ring has a unique *ring address.*
2. It makes sure that there is a single token in the ring.
3. It allows a station to join or leave the ring.
4. It is responsible for reconstituting a ring if the ring breaks because a station moved out of range.

To perform these functions, the MAC protocol needs to interact with the channel allocator, the mobility manager and the admission control manager.

Channel Allocator

We assume that there are multiple channels available for transmission. Each ring transmits on a channel that does not conflict with the neighboring rings. The channel allocator is responsible for choosing the channel that a station transmits on. To do this, it may need to coordinate with the channel allocators in the other stations. It may also use the information from the MAC component about neigboring rings (based on the packets the MAC component hears). The design of the channel allocator may also depend on the specific application that the network is supporting. In this paper, we will not be concerned with the design of the channel allocator. We will assume that transmissions in neighboring rings do not conflict with each other.

Mobility Manager

The mobility manager decides when a station should leave a ring and join another ring. It sends the commands {Join(RA), Leave, MakeRing} to the MAC protocol. The command "Join(RA)" causes a

station to join the ring with ring address RA, "Leave" causes the station to leave the current ring it belongs to, and "MakeRing" causes a station to form a ring consisting of only itself.

The decision of the mobility manager to leave one ring and join another ring may be based on the application being supported. For example, in the design of the automated highway system [6], vehicles are organized into platoons. In this application, vehicles that are part of the same platoon can be part of the same ring. The application requires vehicles to sometimes leave one platoon and join another. This could then naturally be mapped by the mobility manager to a vehicle leaving one ring and joining another ring.

It is also possible to design a more general purpose mobility manager for general adhoc networks. Such a mobility manager may use the information from the MAC component (for example, information about neighboring rings) to decide that it is better to leave one ring and join a different ring.

In this paper we will not be concerned with the design of the mobility manager.

Topology knowledge

In a non-fully-connected network, it is important for stations to know their connectivity, e.g., when a station A wants to send a packet to a station B, it has to know whether a direct connection exists or whether routing has to take place (we discuss support for routing in section 8). In WTBP, we use *connectivity caches* for maintaining a local view of the network topology at each station. We discuss this in more detail in section 5.5.

Admission Control

There is an Admission Control Manager in each ring. The Admission Control Manager moves from one station to next only when the station has the right to transmit. The Admission Control Manager periodically solicits other stations to join if there are "resources" are 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. And the RESV_MTRT is the sum of THT of the station. Now the Admission Control Manager has to ensure the following inequality: $RESV_MTRT < MAX\text{-}MTRT$. Only if there are enough resources left, the Admission Control Manager may solicit another station to join. At the time of solicitation, the Admission Control Manager also advertises the available resources. Only the stations that require less resource than what is left in the ring may join.

4 A High-Level View of the Wireless Token Bus Protocol

In this section we give a basic overview of WTBP. First we describe the normal operation of the protocol. Then we describe how the protocol behaves under various abnormal conditions. A more detailed description of the protocol is provided in Section 5. For simplicity, this section assumes that the connectivity of stations is such that each station is connected to both its successor and predecessor in the ring. We remove this assumption in Section 7, where a generalized version of the protocol is presented.

4.1 Normal Operation

The stations are organized into multiple rings as shown in Figure **3.** There is a token in each ring. Each station has a next station or a successor (called NS) and a previous station or a predecessor (called PS). One-station rings (where PS and NS are same as the station itself) are allowed. The station

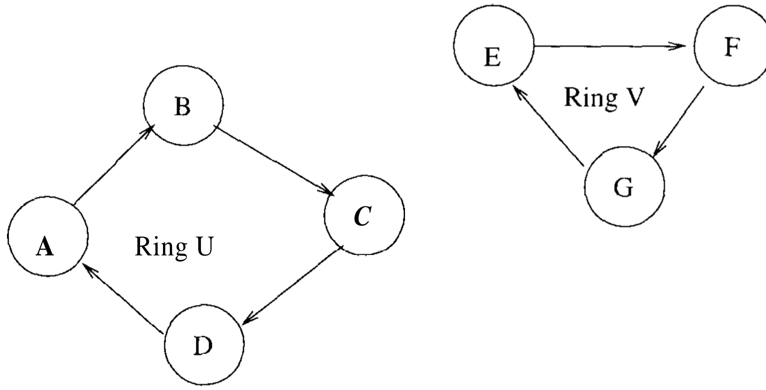


Figure 3: Stations arranged into multiple rings

with the token transmits data. After it has finished transmitting data, the station forwards the token to its successor.

For example, in Figure 3, stations {A, ..., G} are organized into two rings. In ring U, station A has the token. Its PS (denoted PS(A)) is **D** and its NS (denoted NS(A)) is B. After transmitting data, A forwards the token to B.

There are two basic operations supported on a ring. A station may leave the ring, or a single station may join a ring. Using these basic operations, it is possible for a station to move between rings.

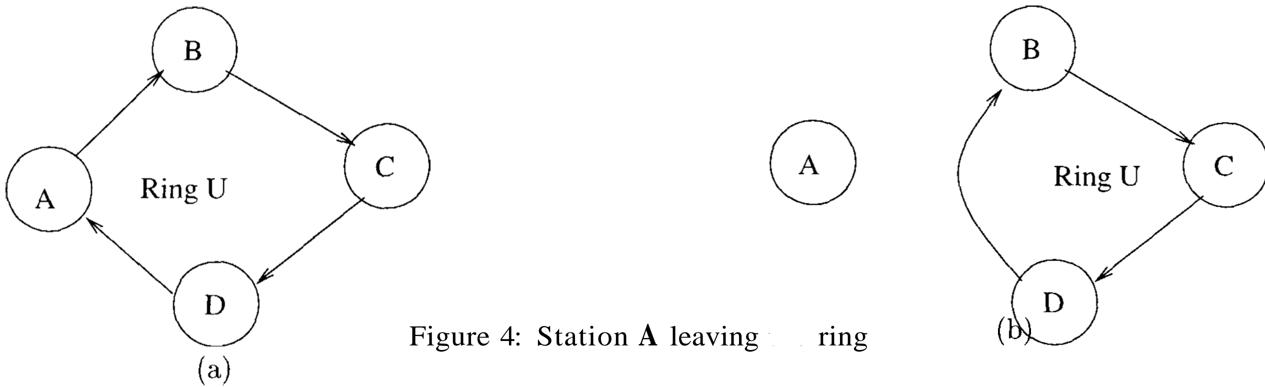


Figure 4: Station **A** leaving ring

Leaving a Ring When a station wants to leave the ring, it waits until it gets the token. It then informs its predecessor that it is leaving. After this the station is free to leave the ring. The predecessor then sends a special token (called the **set_predecessor(X)** token) to the next node in the ring that it knows about to reconstitute the ring. The **set_predecessor(X)** token tells the receiver to set its PS to **X.**

For example, suppose station A decides to leave the ring in Figure 4(a). A then informs PS(A) (i.e., station **D)** that it is leaving the ring. Station D then sends **set_predecessor(D)** token to B. After this B sets PS(B) to D, and **D** sets **NS(D)** to B. Figure 4(b) shows the ring after A has left the ring.

Joining a Ring Each station **X** in the ring periodically sends a special invitation token (called the **solicit-successor** token) inviting other nodes to join the ring. In particular, **X** sends **solicit_successor(X,Y),**

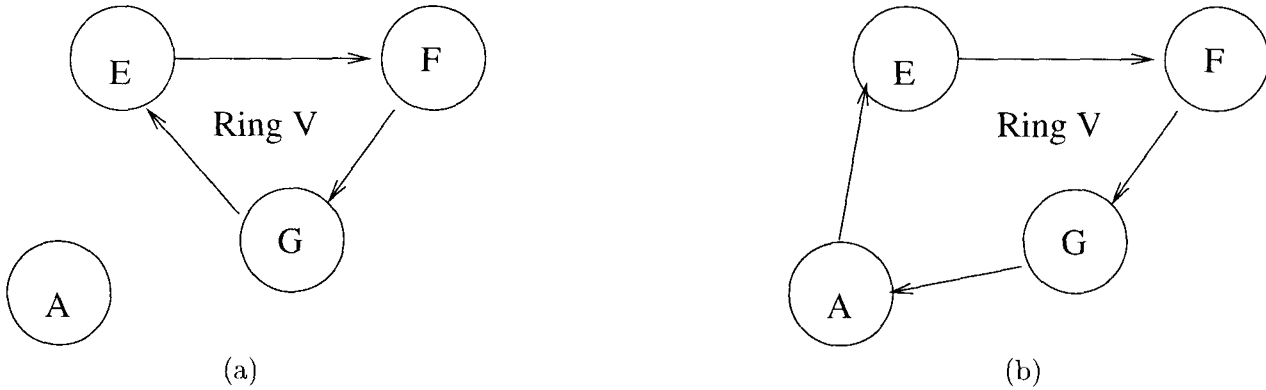


Figure 5: Joining a ring

where Y = NS(X), inviting a station to join "between" stations X and Y. When a station wishing to join hears the solicit_successor(X,Y) token, it checks whether it is within range of Y (it will be within range of X, since it heard the token and we assume symmetric connectivity; in section 7, we remove the restriction of being in range of Y). If it is, it replies back to X. To avoid collisions in case many stations wish to join, each joining station waits a random amount of time before replying back to X. Station X decides to admit one of the replying stations in the ring, and sends the token to that station.

An example of station A joining ring V is shown in figure 5. Station G sends the solicit_successor(G,E) token. Because A wants to join and it is within range of G and E, it replies back to G. Station G then sets NS(G) = A and forwards the token to A. Station A then sets PS(A) = G, sets NS(A) = E, and sends a set_predecessor(A) token to E. This causes E to set PS(E) = A. The new ring is shown in Figure 5(b).

Two things should be noted:

1. A station must observe two successive tokens p and p' with p.RA $= p'$.RA, p.Genseq $= k$ and and p'.GenSeq $= k + 1$, in order to respond to a solicitsuccessor.

2. After a station leaves a ring, it cannot immediately join the same or another ring, but has to wait for at least the maximum token-rotation time, or MTRT (see section 5.2).

4.2 **Abnormal** Conditions

Various kinds of abnormal conditions can arise which must be taken care of by the WTBP. We list some of these abnormal conditions and our protocol's response.

Station moves out of range (or fails) Consider the ring in Figure 6(a). Suppose station A moves out of station D's range. Then station D must realize this when it tries to pass the token to A. In our token ring protocol, a station waits for an "implicit" acknowledgement before it is convinced that its successor has received the token. A transmission by station A is taken by station D to be an implicit acknowledgement that A received the token. Other transmissions in the ring can also be considered as implicit acknowledgments (see detailed description of protocol).

If station D does not receive an implicit acknowledgement, it tries to send the token again to station A. After several failed tries, it gives up. Now the ring is incomplete, *so* station D tries to *close the ring* by sending a set_predecessor(D) token to the next station it knows about in the ring (say station B). Station B then sets PS(B) = D and station D sets NS(D) = B. The reconfigured ring is

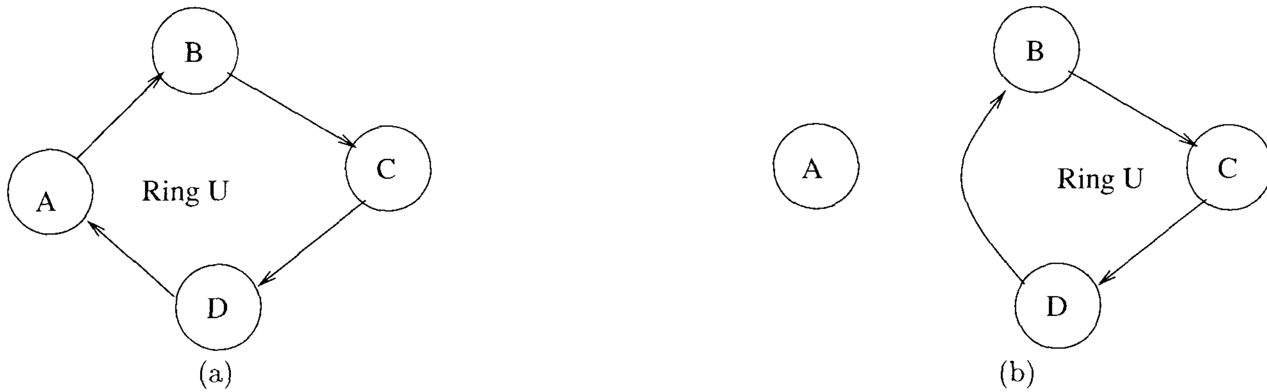


Figure 6: Station A moves out of range

shown in Figure 6(b). Notice that A might still believe it is part of the ring and its successor is B (not shown in the figure).

If the attempt to close the ring fails (because, say, all other nodes have moved out of range) then the node that tried to close the ring will *kick itself out of the ring.* That is, it assumes it is no longer part of the ring. Before kicking itself out, it may decide to notify its predecessor that it is leaving the ring.

The above scenario works even in case A does not move out of range, but fails. In fact, since there is no way for a station to distinguish whether its successor has failed or moved out of range, or if the link between them is too noisy, our protocol will behave the same in all these cases.

No tokens in the ring Suppose station A has the token, but it has moved out of everyone's range as shown in Figure 7(a). Then other nodes will never get the token. So there must be a method to regenerate the token and reconstitute the ring.

In our protocol, if a station has not received implicit acknowledgement for MAX-IDLE-TIME (c.f. idle-timer, section 5.2), it generates a new token and transmits it. [1]

So in Figure 7(b), either station D, B or C will eventually generate a token and try to reconstitute the ring. Now, it is possible for more than one station to generate the token at approximately the same time, but this would be resolved by the part of the protocol which eliminates multiple tokens.

Multiple tokens in the ring Multiple tokens can arise if the token is lost and more than one stations generate new tokens at approximately the same time.

To eliminate multiple tokens in the ring, we use the following technique. Each token carries two fields: a *generation sequence number* (GenSeq) and a *ring address* (RA). The RA is the MAC address of the station which originally generated the token. We call this station the *owner* of the token. The GenSeq field is an integer which is incremented by the owner every time it sees the token. Each station remembers the GenSeq and the RA of the last token they forwarded. A station deletes a token if the token's GenSeq is less than the GenSeq that the station remembers, or the GenSeq are equal but the token's RA is less than or equal to the RA remembered by the station. This protocol ensures that when there are multiple tokens in the ring, only one eventually survives.

[1] Another way to reset the idle-timer is to reset whenever a station transmits a token. This approach was one of the competing ideas during the design of the protocol and the section 6 is based on it. One of the drawbacks of this approach is that if a station has been kicked out without the knowledgement of the station, the idle-timer will expire before the inring-timer, and the station will try to connect to the ring. This can lead to multiple tokens in the ring. More studies needs to be made on this point. The actual implementation resets its inring-timer whenever a station

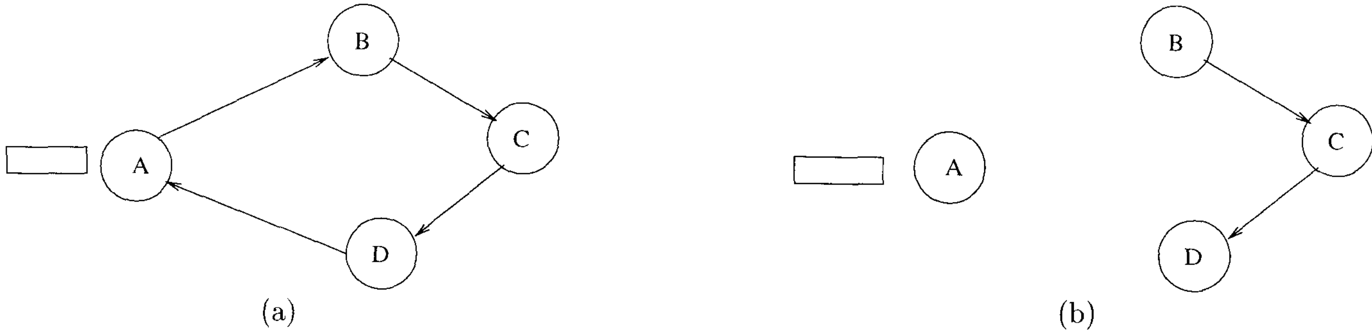


Figure 7: No tokens in the ring

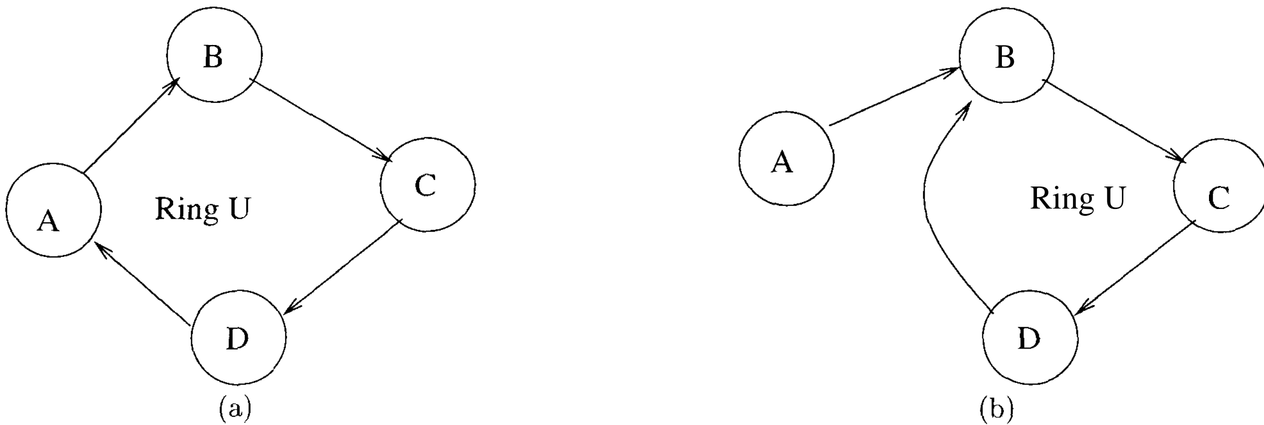


Figure 8: Malformed Rings

Malformed rings Consider the ring of Figure 8(a). Suppose D cannot forward the token to A, so it closes the ring with B. But A stills thinks its successor is B. In the case that **A** is disconnected with the rest of the ring, eventually A will assume the token lost, generate a new token and try to pass it to B. In our protocol, B accepts *normal* tokens (i.e., not set-predecessor or other control packets) only from its PS, so it refuses A's token. Station **A** eventually realizes it is not part of the ring (c.f. inring-timer, section 5.2), and kicks itself out of the ring.

Various other types of abnormal conditions can also arise. WTBP attempts to take care of these abnormal conditions so that the stations eventually settle down into rings each with a single token.

4.3 Other features of WTBP

Implicit acknowledgments Suppose a station A forwards the token to its successor B. Due to errors such as noise, the packet might be corrupted and lost. In such a case (which might occur often due to the nature of the wireless channel) it would not be a good idea to declare the token lost and wait until some node regenerates the token: this might take up to MAX-IDLE-TIME time, which is in the order of token-rotation time, i.e., too long.

Instead, when A sends the token, it sets its token_pass_timer (c.f. section *5.2)* and listens for packets coming from its ring. Any such packet is taken to be an *implicit acknowledgement.* We do not use explicit acknowledgments (e.g., B sending an ACK to A that the token was received correctly), both in order to save bandwidth (a precious resource in wireless networks), and also minimize retransmissions

receives implicit acknowledgement.

(in case B's explicit ACK gets lost).

If an implicit acknowledgment is not received for some time, the token-pass-timer expires (c.f., section 5.2) and the token is retransmitted. After a number of retries (currently, two) the station engages in the close-ring operation.

More specifically, in our implementation, packets that met one of the two following conditions were considered implicit acknowledgement

1. has the same ring address as the station.
2. from any successive nodes observed from the previous token rotation.[2]

Contention resolution in joining When a station X sends a solicitsuccessor token, it is possibly heard by more than one stations who wish to join. To avoid collisions when these stations reply back to X, each of them picks a random slots to send the set-successor as the reply. The size of the slot is based on the transmission time of set-successor and the Contention window is a multiple of slots.

Unique address of each ring The WTBP ensures that each ring has a unique address. This property is important, because it allows stations to distinguish between messages coming from different rings. It also permits to the mobility manager to specify which ring should the station join, upon giving the command "Join".

In WTBP, the ring address is the MAC address of the owner of the token. Since no two stations have the same MAC address, uniqueness is ensured as long as a station cannot be owner of the tokens in two rings at the same time (if, for example, the owner leaves its ring, joins another ring, and creates a new token in the new ring). To ensure that the owner of the token is present in the ring, it is required that the owner "refreshes" the token every time it receives it (by incrementing the GenSeq number). If a station **A** other than the owner receives a token which has not been refreshed, **A** realizes that the owner must have left (or failed) and resumes the role of the owner, by setting the RA to its own MAC address.

5 Detailed Operation of the Wireless Token Bus Protocol

Let us now describe the details of our MAC protocol. A complete specification is given in Appendix D.

5.1 WTBP station information

Each station maintains the following information:

- TS (*This* Station): the MAC address of this station.
- PS (*Previous* Station): 0 if the station does not belong in any ring, otherwise, the MAC address of the previous station (predecessor) in the ring.
- NS (Next Station): 0 if the station does not belong in any ring, otherwise, the MAC address of the next station (successor) in the ring.
- RA (Ring *Address):* an address uniquely identifying the ring. In the WTBP, it is maintained as the MAC address of the owner of the token.

[2]This is because successive nodes can change the ring address

- Seq and GenSeq: copies of the two corresponding fields of the normal token (see section *5.3*). They are updated whenever a token is received by the station and not deleted.
- MRcache and NIMRcaches: the data structures implementing the connectivity cache (see section 5.5 for details).

5.2 Timers

In addition to this state variables, every station mantains a few timers used in different phases of the protocol. All these timers are decremented with time.

- idle-timer — it is set to MAX-IDLE-TIME whenever the station transmits a token. If the timer reaches 0 (we say it *expires* or *times-out),* the station assumes that the token has been lost and goes to the "Generate Token" state (see below).
- inring-timer — it is set to MAXNO-TOKENRECEIVED whenever a station transmitts a token, except if the token has been generated by this station because the idle-timer has expired. If the inring-timer expires, then the station assumes it has been kicked out of the ring and goes to the "Idle" state (see below).
- token-pass-timer — it is set to MAX_ACK_TIME whenever the station sends a token If the token_pass_timer expires before the station hears an implicit acknowledgment, then the station assumes that its successor did not receive the token correctly and retransmits it (see "ForwardTokenState" below).
- token_holding_timer — it is set to MAX-TOKENHOLDING-TIME whenever the station goes into the "SendData" state (see below). The station can transmit data as long as this timer is positive. token before the counter gets to zero. This timer is also used to check if there is enough time to let new nodes join the ring.

The following relations between the time parameters are necessary:

$$\text{MAX-TOKENHOLDING-TIME} < \text{MAX-IDLE-TIME} < \text{MAXNO-TOKENRECEIVED} \quad (1)$$

We will also define the constant MTRT (maximum token-rotation time), such that

$$\text{MTRT} > n \cdot (\text{MAX-TOKENHOLDING-TIME} + \text{MAX_PROP} + \text{MAX-TOKEN-TRANSMIT-TIME}) \quad (2)$$

$$\text{MAX-IDLE-TIME} > \text{MTRT} \quad (3)$$

where n is the maximum number of stations in the ring, MAX_PROP is the maximum signal propagation delay, and MAX-TOKEN-TRANSMIT-TIME is the maximum time to transmit a token.

5.3 WTBP packet formats

WTBP uses 7 different types of packets: one type includes all data packets, and the remaining 6 types are control packets, which we call *tokens*. Some fields are common to all packets, while some are specific to each packet type. We first discuss the common fields, and then the individual packet formats.

The fields common to all packets Every packet type contains at least the following fields:

- TT (Type): a unique code indicating the type of the packet, e.g., data, solicit-successor, and so on.
- SA (Source Address): MAC address of the station sending the packet.
- DA (Destination Address): MAC address of the station the packet is addressed to. If this field is 0, the packet was broadcasted by the sender.
- RA (Ring Address): the address of the ring where the sender belongs to, or 0 if the sender does not belong to any ring.

The normal token This token is used during the normal behavior of the ring, when a station transfers the medium control to its successor. It has the following extra fields:

- Seq (Sequence Number): While the token circulates in the ring, this counter is increased at every transmission. If a token is retransmitted however, it's an exact copy of the previous one and so it contains the same sequence number. The counter wraps-up when reaching its maximum value.

  This field is present to support the connectivity information in the caching mechanism explained in section 5.5, page 22.
- GenSeq (Generation Sequence Number): a counter, incremented by one every time the owner of the token (i.e., the station who generated the token last) forwards the token. GenSeq is used when removing duplicate tokens. It is also used by stations to discover whether the owner of the token has left the ring, in which case, a new station A becomes the owner of the token and the ring address is changed to the MAC address of A.

The solicit-successor token This token is sent by a station in a ring to allow other stations to join the ring. In the basic version of the protocol, this token has one extra field:

- NS (Next Station): the MAC address of the successor of the sender of this token (which will become the successor of the joining node).

The DA of the solicitsuccessor packet is always set to 0 because this packet is broadcast.

In the generalized protocol (section 7), the format solicitsuccessor token is extended to allow for nodes to join even if they are not physically connected to their logical successor.

The set-predecessor token There is no additional field in the set-predecessor with respect to the token type. The receiver accepts the token provided its RA matches the RA of the token, and sets its predecessor field to the sender. The token is used by a station to close a ring, and it is also used during the joining phase.

The set-successor-join and set_successor_leave tokens The set-successor-join token is used during the joining phase. Once the stations willing to join have heard a solicitsuccessor, they wait a random amount of time and then they send a set-successor-join to the sender of the solicit-successor. If the latter receives a valid set-successor-join, it uses the information enclosed to set its successor to the sender, which consequently becames part of the ring.

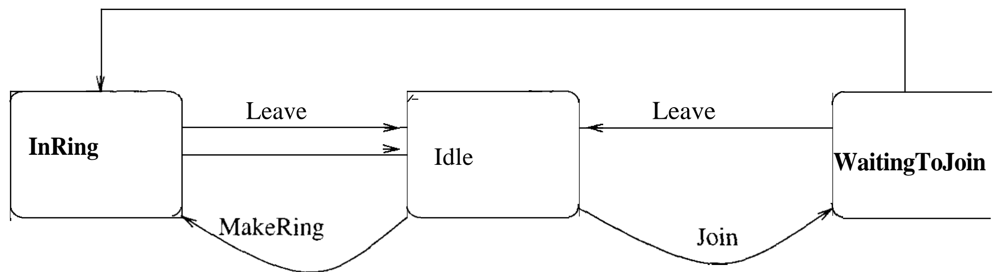


Figure 9: Overall organization of the protocol

The setsuccessorleave token is used when a station decides to leave the ring, either because of a decision taken by the mobility manager, or because the station can't reach its successor and is unable to close the ring with someone else. In any of these cases, the station sends a setsuccessorleave to its predecessor, to let it set its successor to a different station.

The token-deleted **token** When a station receives a token of any type, one of the following reactions may happen:

- The token is accepted — this is the normal operational mode of the ring, when the token travels around the ring without repetitions or duplicates.
- The token is silently deleted — A case when this happens is when a station A has been left out of the ring (perhaps because some other node has closed the ring), but A doesn't know this. It may happen that A generates a new token and tries to send it to the station it still thinks to be its successor (let's say B). When receiving this token, B simply ignores it. Eventually, the inring-timer will expire and A will realize that it is out of the ring.
- The token is deleted with notification to the sender — This happens when a station A has not heard the implicit acknowledgment and retransmits the token to its successor B. B will realize that and send to A a token-deleted token.

The token-deleted doesn't have any additional fields apart from the common fields.

The data **packet** This packet type represent the generic data frame. It is the only variable length packet, and the only packet in which part of the content is determined in a layer above the MAC. It declares the following fields:

- data_size: is the size (in bytes) of the data enclosed in this packet.
- data: variable length field containing the data.

Notation We denote the field A of token T as T.A or A(T). Similarly, we denote the variable or timer V of station S as S.V or V(S). When S is clear from context (e.g., "this" station) we simply write V.

5.4 Finite-state-machine specification

We will describe WTBP using automata extended with timers and C-like code to test and update variables. To ease the task, the specification of WTBP is actually done *hierarchically,* i.e., in multi-level automata. The states in a higher-level automaton are *macro-states* which expand into detailed

lower-level automata. When a high-level automaton enters a macro-state, it yields control to the lower-level automaton, and when the low-level automaton finishes execution, it hands control back to the high-level automaton. Only one automaton is active at any given time, that is, the model is equivalent to a "flat" model of a single automaton.

The highest-level automaton describing WTBP is as shown in Figure 9. The macro states are { InRing, Idle, Join }. In the Idle state, the station is waiting for a command from its topology manager. In the WaitingToJoin state, the station is waiting to join a ring. In the InRing state, the station is part of a ring. We now describe the automata corresponding to these macro-states in detail.

5.4.1 In Ring macro-state

Figure 10 shows the automaton for the InRing macro-state. We describe each of the states below:

- WaitForToken: The station waits for a token. When it receives one, it moves to the state ProcessToken.

- ProcessToken: In ProcessToken, the station looks at the token to see which of the boolean conditions SSL_Valid(token), MT_Valid(token) or CR_Valid(token) is true. These conditions are given below:

  1. $SSL_Valid(token) \equiv (token.TT ==$ set-successor-leave).
  2. $MT_Valid(token) \equiv (token.TT ==$normal && $token.SA ==$PS).
  3. $CR_Valid(token) \equiv (token.TT ==$ set-predecessor && $token.RA ==$RA).

  Upon the SSL_Valid(token) condition being true, the station moves to the SetSuccessorLeave state; when CR_Valid(token) is true, the station moves to the CloseRing state; and when MT_Valid(token) is true, the station moves to the Manage_Token state. If none of these conditions are true, the station deletes the token and moves back to the WaitForToken state. By deleting a token, we mean that an notification is sent to the sender of the token.[3]

- SetSuccessorLeave: In the SetSuccessorLeave state, the station sets NS = NULL. The station maybe asked to leave, in which case it goes to the LeavingRing state. Otherwise, it sets the token.TT = set-predecessor, and goes to the ForwardToken state.

- ManageToken: The ManageToken state eliminates multiple tokens in the ring. It does this by checking whether it is a duplicate token because it matches the GenSeq of a token it has seen before, or whether the token's address is "less" than the address of a token it has previously transmitted. In these cases, it deletes the token. If the station is the owner of the token, it refreshes the token by incrementing token.GenSeq. The manage_token() function is shown in Figures 11 and 12.

- CloseRing: The station sets its PS = token.SA. It then checks whether the owner of the token has left the ring. The close-ring() function is shown in Figure **13.**

- GenerateToken: In the GenerateToken state, the station generates a new token by setting token.GenSeq = GenSeq + 2, and token.RA = TS. So the station becomes the owner of the new token generated.

[3] Another competing ideas for the token deleting policy is to delete duplicate tokens silently by not notifying the sender of the token. If it is highly probable that the token received by the current station is still in the ring and the sender of the token will hear it soon, then silently deleting the token is a better choice. This can be a highly probable case in highly connected network.

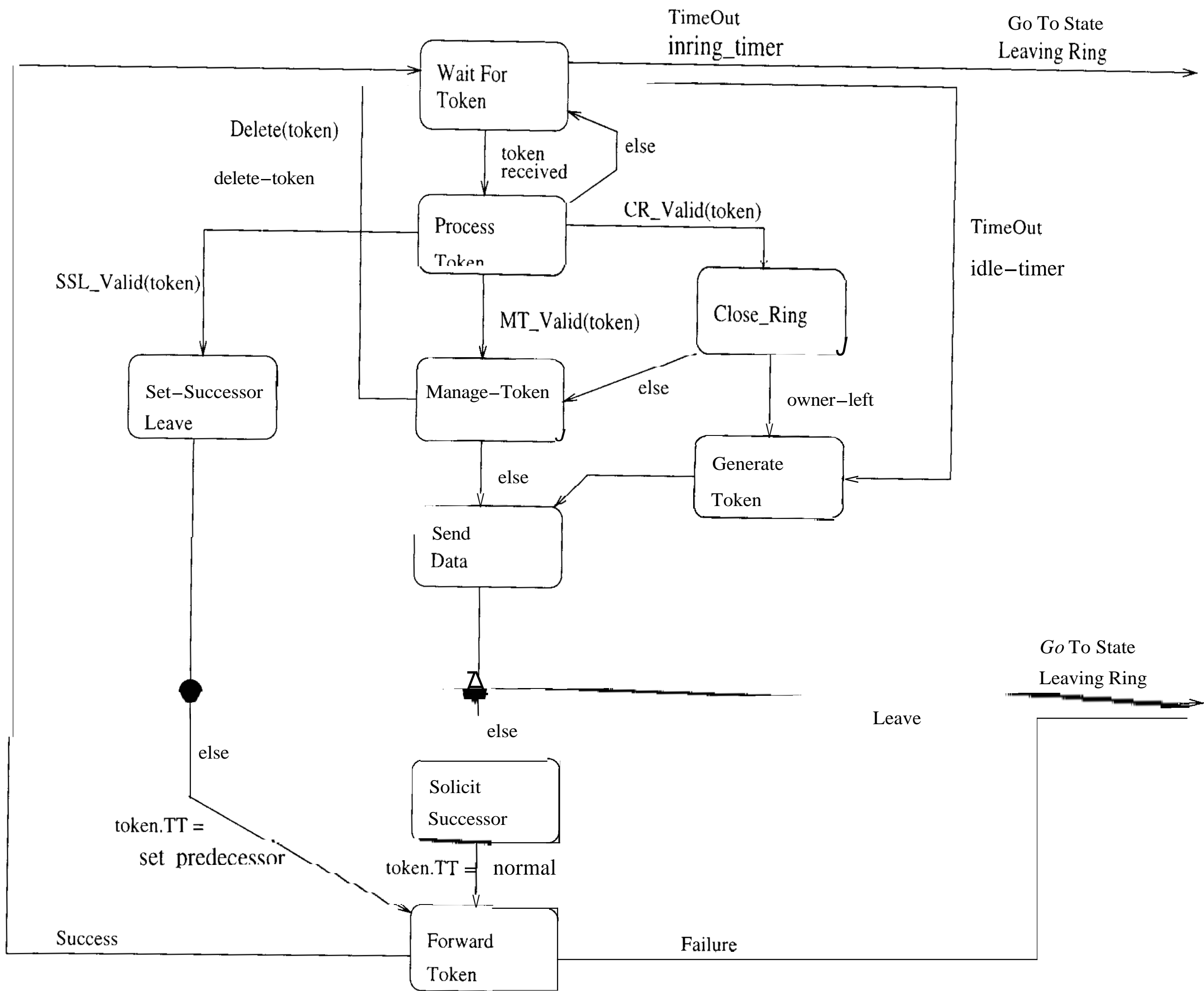


Figure 10: The InRing automaton

```
manage_token(token):
    manage_token(token) {
        delete-token = FALSE;
        if (is_duplicate_token(token) || address-&(token)) {
            delete-token = TRUE;
        } else if (token_owner(token)) {
            refresh_token(token);
        } else {
        }
```

Figure 11: manage-token() function

```
duplicate_token(token) ≡ (token.GenSeq < GenSeq && token.RA ≡ TS)
        || (token.GenSeq ≤ GenSeq && token.RA != TS && token.RA ≡ RA)
address_gt(token) ≡ (GenSeq > token.GenSeq) || ( (token.GenSeq ≡ GenSeq) && (RA > token.RA) ).
token_owner(token) ≡ token.RA ≡ TS
```

Figure 12: Functions called by manage_token()

```
close-ring(token):
    close_ring(token) {
        PS = token.SA;
        if (token.GenSeq == GenSeq && token.RA != TS)
            generate-token = TRUE;
        else
            generate-token = FALSE;
    }
```

Figure **13:** Close-Ring() function

- SendData: In the SendData state, the station sends data. After this, the station maybe asked to leave, in which case it goes to the LeavingRing state. Otherwise, it goes to the SolicitSuccessor state.
- SolicitSuccessor: In the SolicitSuccessor state, a station may invite other stations to join the ring. It does this only if its RA has not changed. It does this by sending a solicit-successor token to which other stations respond. It then accepts one of these stations into the ring. After doing this, it sets the token.TT = **normal** and goes to the ForwardToken state.
- ForwardToken: In the ForwardToken state, the station sends the token to its NS. If it does not receive an acknowledgement, then it sets the token.TT = set-predecessor, and attempts to close the ring with another station. If it is successful in one of these attempts, it goes to the WaitForToken state. Otherwise, it kicks itself out of the ring, and goes to the LeavingRing state.

Solicit Successor State The automaton for the Solicit Successor macro-state is shown in Figure 14. The states of the automaton are as follows:

- CheckToSolicit: The station only solicits a successor with some probability p and when the admission control module allows it to do so. It then checks to see whether its RA has changed. If the RA has not changed, it goes to the SendSolicitSuccessorMessage state. Otherwise, it goes to the Done state.
- SendSolicitSuccessorMessage: In this state, the station sends a solicit-successor token. If it receives a valid response before a timeout, it sets its NS to the sender of the response. Otherwise, it timesout and goes to the Done state.
- Done: Finished with Solicit Successor

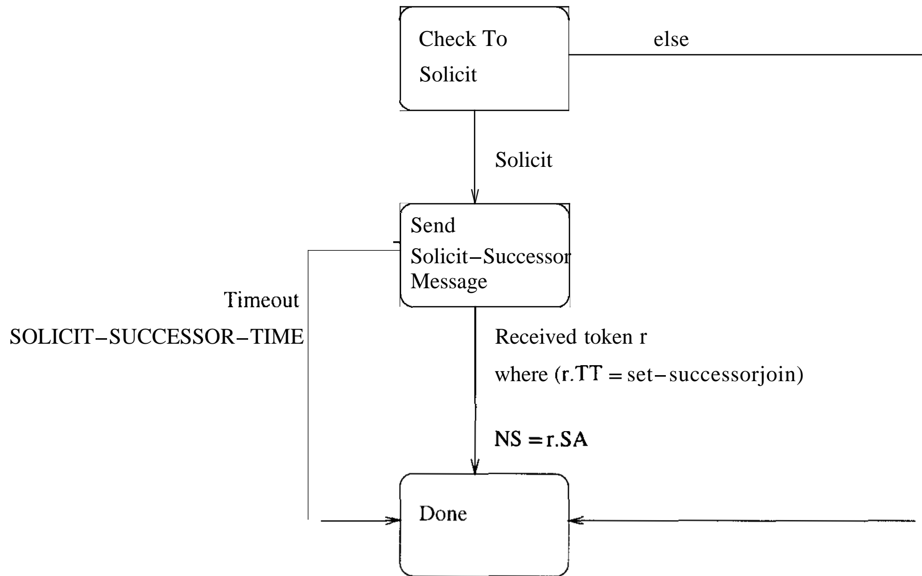


Figure 14: Solicit Successor automaton

Forward Token macro-state The automaton for the Forward Token macro-state is shown in Figure 15.

The states of the automaton are as follows:

- StartForwardToken: The station first checks to see whether the NS field is valid (i.e., it is not set to NULL). If the field is valid, it goes to the SendAckData state, otherwise it goes to the SenderCloseRing state.
- SendAckData: In the SendAckData, the station sends the token to NS and waits for an implicit acknowledgement. If it fails to receive an implicit acknowledgement, it tries again. After two or more failures, it gives up and goes to the SenderCloseRing state. If it succeeds, it goes to the Done state.
- SenderCloseRing: In the SenderCloseRing state, the station tries to close ring with another station in the ring.
- Done: Finished with Forward token.

The idle-timer is reset in the ForwardToken state. The inring-timer is also reset in the Forward-Token state provided the token being sent was not generated because of an idle-timer timeout.

5.4.2 Waiting to Join macro-state

Figure 16 shows the automaton for WaitingToJoin macro-state. We describe each of the states below:

- **WaitForSolicitSuccessorToken:** The station waits for a solicit-successor token. When it receives one, it moves to the next state.
- WaitForContentionTimer: The station sets a contention-timer. If the station hears something from the ring from which it received the solicit-successor token before the timeout, it returns back to the WaitForSolicitSuccessor state. Otherwise on timeout, it goes to the SendSetSuccessorMessage.

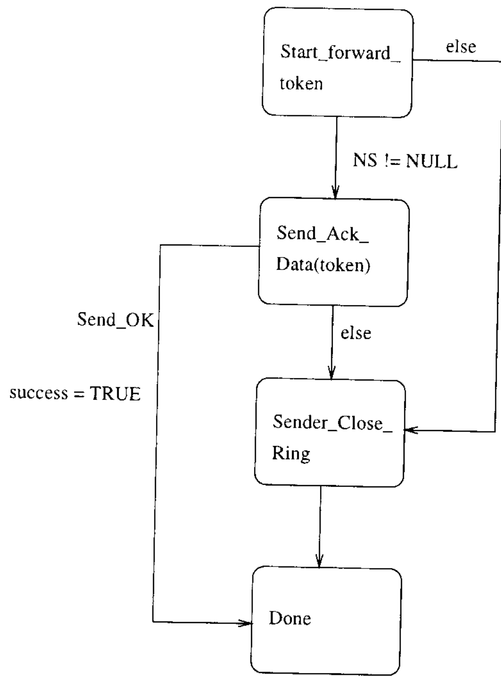


Figure 15: Forward Token automaton

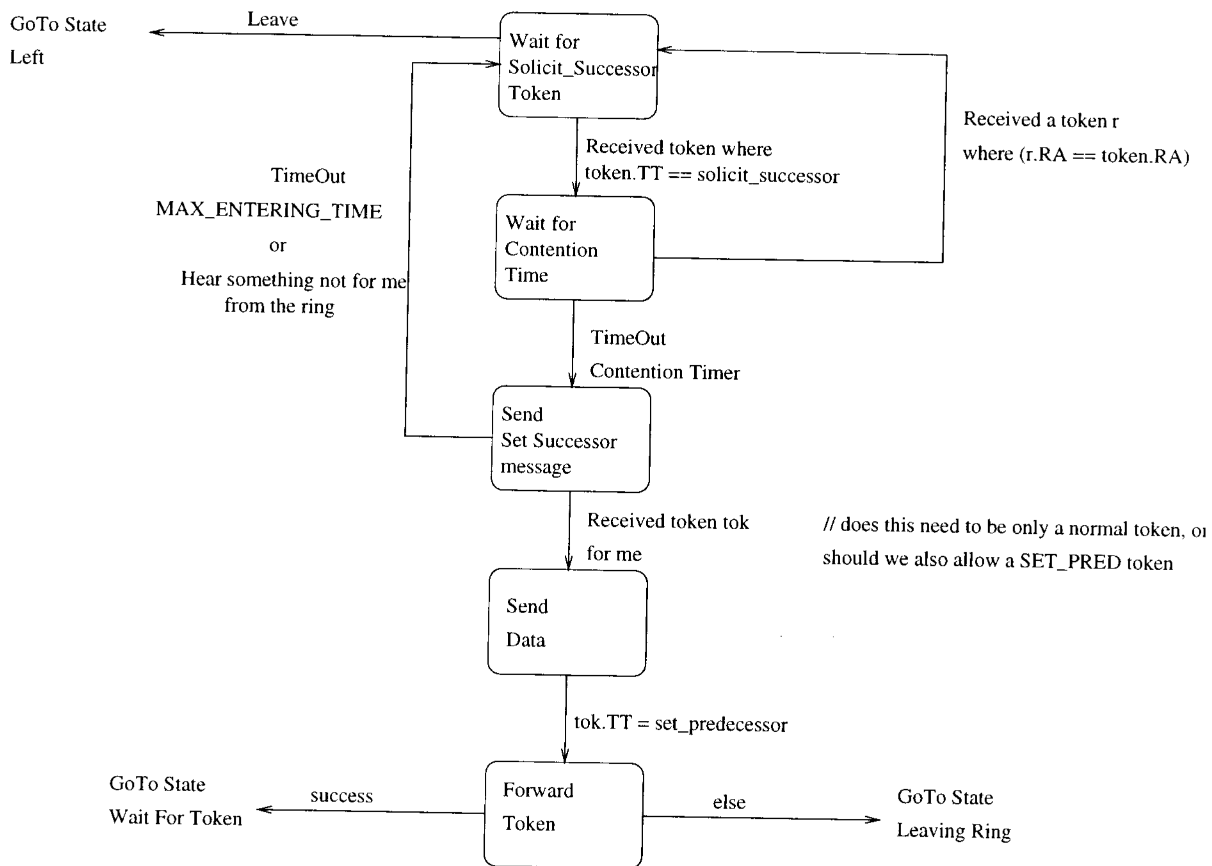


Figure 16: The Wait to Join automaton

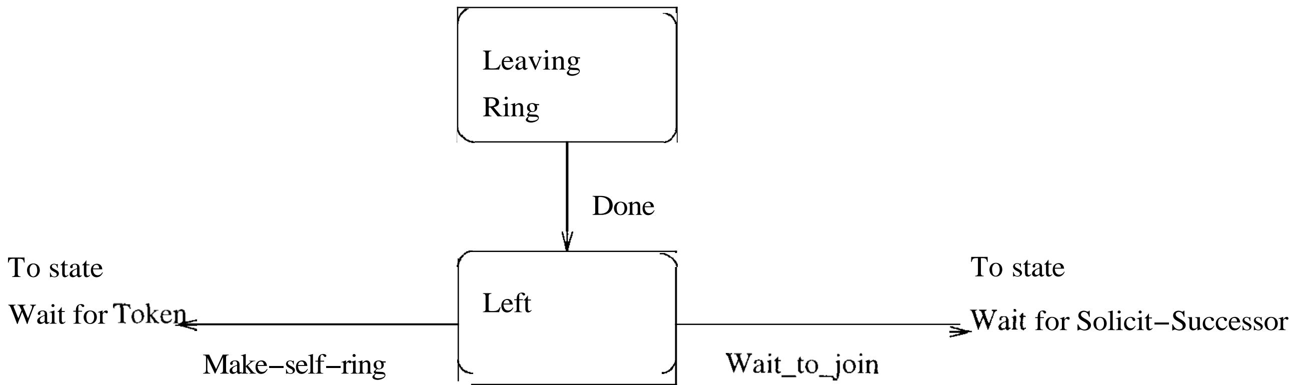


Figure 17: The Idle automaton

- SendSetSuccessorMessage: The station sends a set-successor-join token to the sender of the solicit-successor token. If it hears something from the ring which is not meant for it, it goes to the **WaitForSolicitSuccessorToken** state. Or if it does not hear anything for MaxEnterTime, it timesout and goes to the **WaitForSolicitSuccessorToken** state. Otherwise, if it gets a token before the timeout, it goes to the SendData state.
- SendData: After sending data, it sets the token.TT = set-predecessor and goes to the Forward-Token state.
- ForwardToken: This is the same as the ForwardToken state in Section 5.4.1.

5.4.3 Idle macro-state

Figure 17 shows the automaton for the idle macro-state.

- LeavingRing: The station sends a set-successorleave token to its PS. After this it moves to the Left state.
- Left: Once the station arrives into this state, it stays here for MTRT time. In this state, the station waits for a command from the topology manager. It either makes itself into a ring and moves to the WaitForToken state, or it moves to the WaitForSolicitSuccessor state.

5.5 Connectivity Caches

5.5.1 Need for topology knowledge

In a non-fully-connected network, it is important for stations to know their connectivity, e.g., when a station A wants to send a packet to a station B, it has to know whether a direct connection exists or whether routing has to take place (we discuss support for routing in section 8).

Other situations where topology knowledge is desirable in WTBP are the following:

- When a station wants to join a ring, it will probably choose to join in a logical position where it has the highest connectivity with its neighbors.
- When a station cannot reach its successor, it may decide either to close the ring, or kick itself out of the ring, if closing the ring would leave too many stations out. To make this decision, topology knowledge is essential.

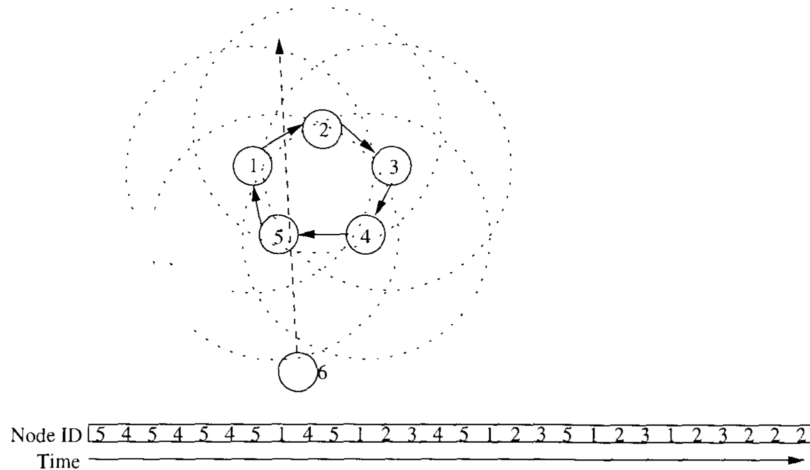


Figure 18: A mobile station passing near a ring

- When a station decides to close the ring, it is beneficial to keep as much nodes in the ring as possible. The knowledge of the transmission order of the stations allows the closing station to choose to close the ring with a station that will form a new ring with the least number of kicked out nodes.

We should note that topology knowledge, although essential for performance, does not affect the correctness of the protocol. Therefore, many approaches to building and maintaining topology knowledge can be envisaged, also depending on the application requirements, degree of mobility, and so on.

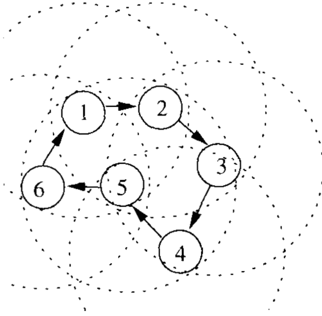
5.5.2 The approach taken in WTBP

Instead of an "active" approach, where stations try to figure out the needed topology information by sending control packets explicitly, the WTBP uses a "passive" approach where stations listen to packets being transmitted around them and update one or more local caches, namely, one `MRcache` (my-ring) and zero or more `NIMRcaches` (not-in-my-ring), for every ring the station can hear from. We briefly discuss the two types of caches in what follows.

NIMRcache. For each `RA` heard, an `RA-NIMRcache` is maintained, which keeps a FIFO queue of entries of the form (Sender ID, Timestamp). An entry (A, t) means station A was heard transmitting at ring `RA` at time `t`. Each time a packet is heard with ring address `RA`, a new entry is inserted at the tail of the queue. If the queue is full, its head is removed.

Figure 18 shows a situation in which a mobile station (station 6) comes in contact with a ring, and then moves away. On the bottom of the figure a possible evolution of the mobile station's `NIMRcache` content (only sender IDs) is shown.

MRcache. This cache relies on the `Seq` field of the token. Before sending out a token, a station increments its `Seq` number by one. The `Seq` number will be incremented for each station visited by the token. Whenever a station hears a token from its own ring, the token is given to a `MRcache` component



Token Sender	Token Seq	Node MRcache tables (IDs)					
		Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
1	1	1	1			1	1
2	2	2	12	2		12	1
3	3	23	3	23	3	123	1
4	4	23	34	4	34	1234	1
5	5	2305	345	45	5	12345	1UUU5
6	6	23U56	345	45	5	6	1UUU56
1	7	23U561	34501	45	5	61	
2	8		345U12	45UU2	5	612	
3	9			45UU23	5UUU3	6123	
4	10				5UUU34	61234	
5	11					612345	

Figure 19: Construction of the MRcache cache. Dashed horizontal lines represent the instant when the cache is cleaned

for processing. As long as the processed tokens have consecutive `Seq` numbers, the component just builds a table whose entries have the same format used in the `NIMRcache`.

If a token is heard whose `Seq` number is $\texttt{oldSeq}+n$, where `oldSeq` is the `Seq` value in the previously heard token and $n > 1$, then before the entry for the sender of the token is added to the table $n - 1$ entries are filled with a special value that will be indicated as `U` ("unknown" or "unreachable" station). `U` entries represent the knowledge of the fact that a certain number of stations are present between the previous and the current sender of the token, but those stations have not been heard sending the token by this station. The reason for not hearing the token may be a large distance among the sender and this station, or a collision/noise that garbled the token (note that the garbling may be just local to this station, while the token correctly reached its destination).

Once a station receives the token, it has built an image of its connectivity to other stations in the ring, as well as a knowledge about the position of stations that are not reachable. After the token has been sent successfully, the cache is emptied to be rebuilt in the next token rotation. Figure 19 shows an example of a possible ring and how the ring cache is updated.

The information carried by the Us in the tables are particularly important for closing the ring. Suppose in the example of the above figure station 6 becomes unreachable. After trying to send the token two times, station 5 will close the ring with station 1, leaving 6 out of the ring.

Suppose now station 1 is the one failing. Node 6 can close the ring only with station 5, because it's the only station remaining within its range. However, doing so would throw out three stations from the ring. Therefore, 6 might decide instead to kick itself out of the ring.

6 Proof of Stabilization

We next provide a proof that our protocol is self-stabilizing. This means that, if transmission errors stop happening and the topology stops changing, then eventually all stations will belong to well-formed rings, with a single token circulating in each ring.

Our proof will be organized into three phases. Assuming that the assumptions hold from time T_0 onwards, we first show that multiple equivalent tokens (i.e., tokens having the same ring address) will be eliminated by some time $T_1 > T_0$. Using this, we then show that stations organize themselves into well-formed rings by time $T_2 > T_1$, and these rings do not change thereafter. Finally we show that there is a single token within each ring by time $T_3 > T_2$.

Definitions and Assumptions

We say tokens a and b are *equivalent* provided $a.\mathtt{RA} = b.\mathtt{RA}$. We define the *priority* (of a station or a token x) to be the pair (g, r) where $g = x$.**GenSeq** and $r = x.\mathtt{RA}$. We define $(g_1, r_1) > (g_2, r_2)$ when either $g_1 > g_2$, or $g_1 = g_2$ and $r_1 > r_2$. A ring R is a set of nodes. We say that R is *well-formed* if for each node $x \in R$, $\mathtt{NS}(\mathtt{PS}(x)) = x$. We call a well-formed ring of one station a *self-ring*.

We assume that from time T_0 onwards:

1. Topology is fixed (i.e., if x and y are within each other's range then they continue to remain so).
2. Messages do not get lost.

Also recall the relations of time parameters from equations **1–3** in section 5.2. These relations, along with the above assumptions, imply that:

3. MTRT is the maximum time it takes for a token to go through all stations.
4. A token does not survive for more than MTRT, unless if it visits its owner.

Elimination of Equivalent Tokens

Using the next few lemmas, we show that starting at time t, multiple equivalent tokens get eliminated by time $t + 2\mathtt{MTRT}$.

Lemma 6.1 *While a station is in the InRing phase, the priority of the station increases with time.*

Proof: A station x accepts a token p only if, either x is not the owner of p and x.**GenSeq** < p.**GenSeq** (after which x sets x.**GenSeq** = p.**GenSeq**), or x is the owner and x.**GenSeq** == p.**GenSeq** (after which x sets x.**GenSeq** = p.**GenSeq** + 1). Therefore, if x accepts a token, x's priority increases. If x does not accept a token for a while, it will generate its own token after increment x.**GenSeq** by 2, therefore, again x's priority increases. ∎

Lemma 6.2 *Pick any token p at time $t_0 \geq T_0$, and build an ordered list of the path taken by p, say $\langle(x_0, t_0), (x_1, t_1), (x_2, t_2), \ldots, (x_m, t_m)\rangle$, where t_n is the time that x_n transmits p and $t_{i+1} > t_i$. If there exists a station $x_i = x_j$ in the list such that $0 \leq i < j \leq m$, then there must be a k such that $i \leq k \leq j$, and x_k owns p.*

Proof: Let's assume the contradiction: Suppose we find $x_i = x_j$ such that $0 \leq i, j \leq m$ but we cannot find the owner of p, x_k, such that $i \leq k \leq j$. This means that the p.**GenSeq** when p arrives at x_i is equal to p.**GenSeq** when p arrives at x_j, because no station other than the owner of the token modifies the generation sequence number. Now, x_j remains at InRing from time t_i until t_j (otherwise, it wouldn't accept any token at time t_j). This is because $t_j - t_i <$ MTRT (by the assumptions) and the fact that a node waits MTRT before it joins again. Therefore, from Lemma 6.1, p.**GenSeq** $\leq x_j$.**GenSeq** at time t_j. Thus, x_j would have deleted the token instead of transmitting it. Contradiction. ∎

Lemma 6.3 *Starting at time T_0, no multiple equivalent tokens exist at time $T_1 = T_0 + 2\mathtt{MTRT}$.*

Proof: Consider tokens at time T_0 whose owner is y. Then from Lemma 6.2 and the assumptions, all such tokens will either visit again y by time $t' \in [T_0, T_0 + \mathrm{MTRT}]$, or get deleted. So, at time t', y.**GenSeq** $\geq p$.**GenSeq** for all tokens p owned by y. Moreover, since during $[T_0, t']$ y deletes all tokens with p.**GenSeq** $\leq y$.**GenSeq**, at time t' there will be at most one token such that y.**GenSeq** == p.**GenSeq**. At most this token will survive when visiting y the next time, which happens at the latest at $t' + \mathrm{MTRT} \leq T_0 + 2\mathtt{MTRT}$. (As in the proof of Lemma 6.2, y must either remain in InRing during $[t', t' + \mathrm{MTRT}]$ or no tokens are accepted by y, that is they all get deleted.) ∎

Lemma 6.4 *If there were no multiple equivalent tokens at time $t \geq T_0$, then no multiple equivalent tokens exist at time $t' \geq t$.*

Proof Because of our assumption of no transmission errors, it is impossible for multiple equivalent tokens to be generated (say, due to packet corruption). Then a station must have generated a token when a token that it has previously generated is still in the graph. But this is impossible, from the fact that MAX-IDLE-TIME > MTRT and assumption 4. ■

Lemma 6.5 *No multiple equivalent tokens exist at any time $t \geq T_0 + 2\mathtt{MTRT}$*

Proof From Lemma 6.3 and Lemma 6.4. ■

Ring Repair

In this section we will show that when there are no multiple equivalent tokens, stations organize themselves into well-formed rings.

Lemma 6.6 *Suppose there are no multiple equivalent tokens in the ring. Then if a station performs a join operation, it joins a well-formed ring, whose owner is also in the ring. Furthermore, no station in such a ring is ever kicked out.*

Proof Assume station s joins a ring between stations u and v. For this to happen, s must observe a token p with $p.\mathtt{Genseq} = k$, and again observe a token p' with $p'.\mathtt{GenSeq} = k + 1$ and $p.\mathtt{RA} = p'.\mathtt{RA}$. This implies that the owner of token is in the ring (otherwise, the RA of the token would have changed by the new owner), and that the ring is well-formed (otherwise, the token would not come back). After the transmission of p', no inring-timer expires, since between p and p' at most MTRT time has elapsed and MTRT < MAX_NO_TOKEN_RECEIVED. Because of this and the fact that no erroneous transmissions occur, no stations are ever kicked out. ■

Lemma 6.7 *Suppose there is a ring which remains non-well-formed for more than* MAX_NO_TOKEN_RECEIVED *time. Then, some station goes to state "Idle" during this time.*

Proof Let R be a non-well-formed ring during the interval $[T, T']$, where $T' > T + \mathtt{MAX_NO_TOKEN_RECEIVED}$. From the definition of well-formed rings, there is some station $y \in R$ such that $(y.\mathtt{PS}).\mathtt{NS} \neq y$. Let $x = y.\mathtt{PS}$. Either x passes a token to y during the interval $[T, T']$, or y's inring-timer expires by time T', y goes into state "Idle", and we're done. Let t' be the last time x passed a token to y in the interval $[T, T']$. Now, at time t', $x.\mathtt{NS} = y$ (since x sends the token to its NS), but at time T', $x.\mathtt{NS} \neq y$. The NS of a station changes only when a station tries to close the ring (this cannot happen because of the assumption for no packet loss), or when the station's inring-timer expires and the station moves to "Idle". Therefore, x must go to "Idle" during the interval $[t', T']$. ■

Lemma 6.8 *Every station is eventually part of a well-formed ring that does not change.*

Proof From Lemma 6.7, all nodes which do not belong to well-formed rings will eventually be kicked out. Since no packets are lost, all these nodes will eventually join some ring. From Lemma 6.6, this will be a well-formed ring that does not change. ■

Multiple token resolution

We next assume that all stations have organized themselves into rings. The next lemma shows that even if there are multiple tokens within a ring, after some time, the ring will stabilize with a single token in the ring.

Lemma 6.9 *Suppose each station belongs to a well-formed ring at time* $t \geq T_1$. *Then by time* $t +$ 2MTRT + MAX-IDLE-TIME, *there will be a single token in each ring.*

Proof Consider the highest priority token at time $t' = t +$ MAX-IDLE-TIME (there must be a token in the ring at time t', because at least one of the tokens transmitted during the interval $[t,t']$ must have survived). By following this token around the ring, we can show that by time within $t' +$ 2MTRT, there will be exactly one token in the ring and each station in ring will have the same ring address as the token. ■

Using Lemmae 6.5, 6.8 and 6.9, we can conclude that eventually every station belongs to a ring, and there is a single token in each ring.

7 The Generalized Wireless Token Bus Protocol

Motivation. Under the basic protocol presented in section 4, a ring that includes all stations cannot always be formed. For example, consider the four stations and the connectivity situation depicted in figure 20(a). Two stations connected with a solid line means they are in range with each other. Assuming we start from a situation where no ring is formed, then no ring that includes all stations can be formed in this case. This is because in order for a station to join a ring, it has to be connected to both its predecessor (who sends the **solicit-successor**) and its to-be successor. This is not always true, as shown in figure 20. Assume, for example that we start with a one-station ring (fig. 20(b)) and then another station joins (fig. 20(c)). At this point, no station can join this ring anymore. In the end, two distinct rings will be formed (fig. 20(d)).

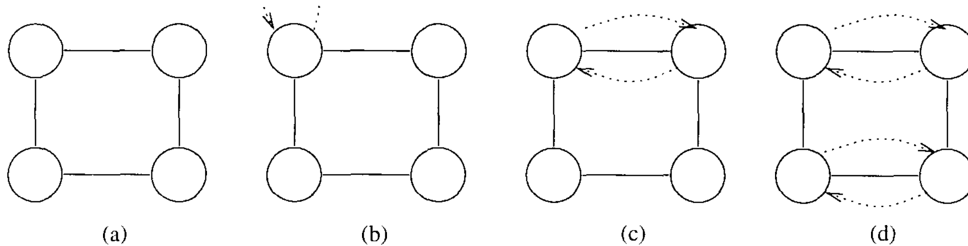


Figure 20: Motivation for the generalized protocol.

The basic protocol can be generalized so that, given any network topology, it can form a token-ring including all stations in this network, as long as every station is connected to at least one other station. We show how this can be done in this section.

7.1 High-level description of the generalized protocol

In a nutshell, the features of the generalized protocol are the following.

7.1.1 No requirements on topology

A station A is not required to be directly connected to its logical successor B. A forwards the token to B along a sequence of *relay* stations $C_1 \ldots C_n$ (using *source routing,* that is, the token contains the route). The sequence $(A, C_1, ..., C_n, B)$ is the *token route* from A to B and is learned when A joins, as explained below. Each station stores the token route to its successor in a variable TR.

For example, consider a network represented by the graph of figure 21. A, B, ... are wireless stations. A link represents the fact that two stations are within range of each other. One possible ring formed on top of this network is shown in figure 22(a), where the dotted arrows represent the successors: $\texttt{NS}(A) = E$, $\texttt{NS}(E) = C$, and so on. The dashed arrows represent the token routes: $\texttt{TR}(C) = (C, D)$, $\texttt{TR}(D) = (D, C, B)$, and so on.

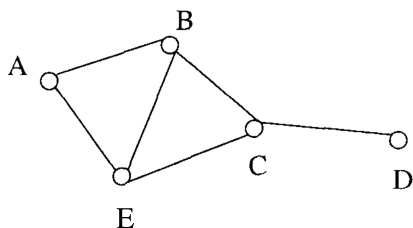


Figure 21: An example network.

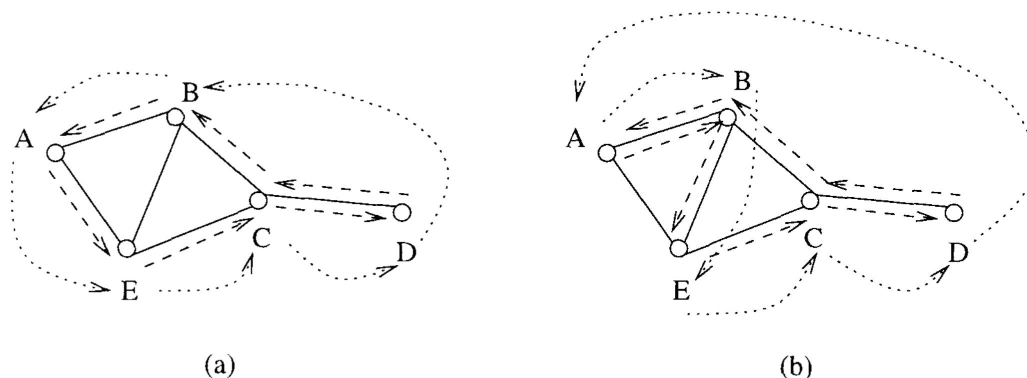


Figure 22: Two possible rings for the network of figure 21.

Each station X with $\texttt{TR}(X) = (Y_1, ..., Y_k)$ maintains invariant the facts: (a) $Y_1 = X$, (b) $Y_k = \texttt{NS}(X)$, and (c) $\texttt{TR}(X)$ is loop-free.

7.1.2 Format of solicitsuccessor token

The format of the solicit-successor token is extended to $\texttt{solicit_successor}(Y_k = X, Y_{k-1}, ..., Y_0 = Y)$, where X is the station that sends the token, $Y = \texttt{NS}(X)$, and $(X = Y_k, Y_{k-1}, ..., Y_0 = Y)$ is the token route from X to Y, $k \geq 1$. The meaning is:

> "If you are station Z, you can join the ring with logical predecessor X and logical successor Y. The token-route from Z to Y should be chosen as $Z \rightarrow Y_i \rightarrow Y_{i-1} \rightarrow \cdots \rightarrow Y_1 \rightarrow Y_0$, where i is the smallest index in $[0,k]$ such that Z is connected to Y_i."

In the best case, Z is connected to Y and chooses $i = 0$. In the worst case, Z chooses $i = k$ (it can do that, since it is certainly connected to X, or else it could not have received the solicit-successor token).

Ring formation. Continuing the above example, figure 23 shows a sequence of possible steps that results in the formation of the ring of figure 22(a).

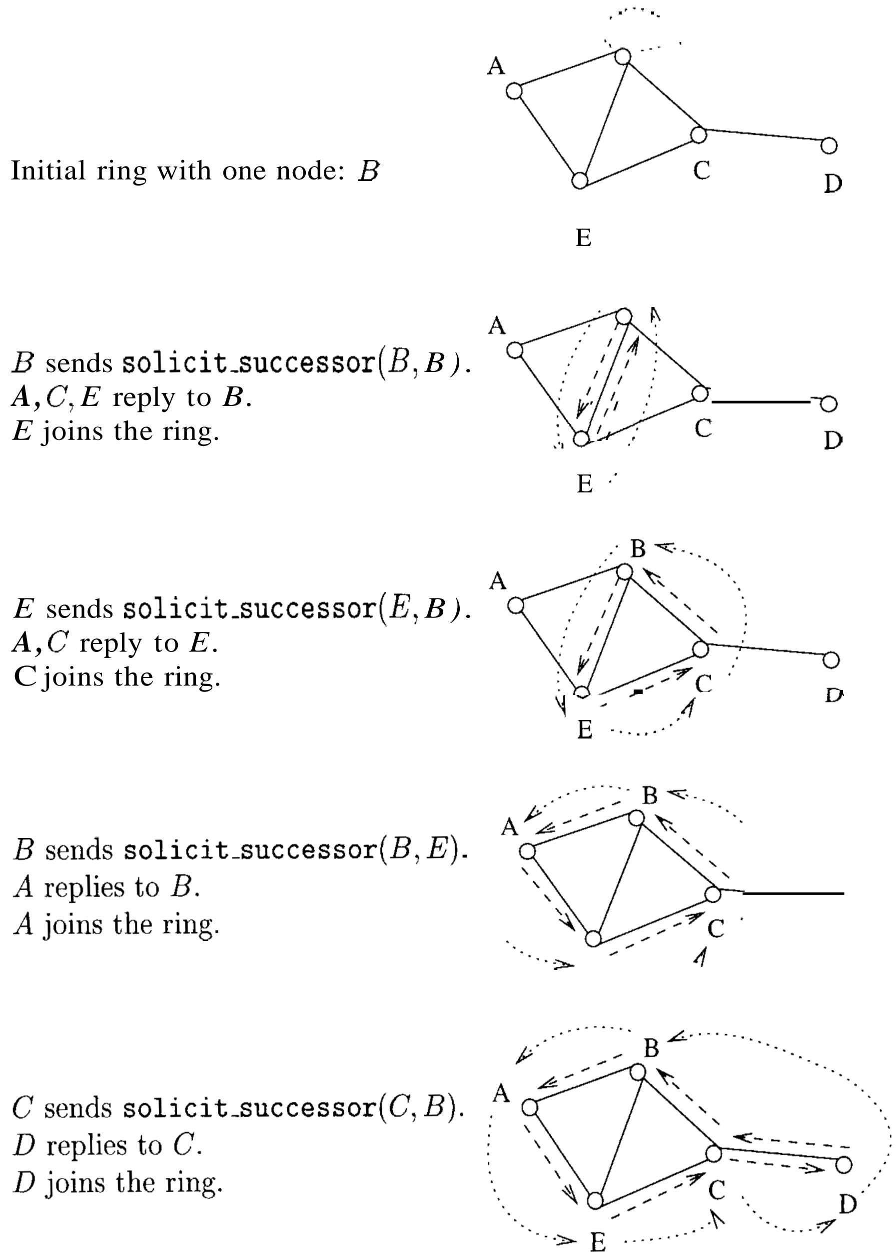


Figure 23: Steps in the formation of the ring of figure 21(a).

Non-uniqueness of logical rings. Different logical rings can be formed on the same network, depending on the order in which stations send solicit-successor packets, and the order in which new

stations are accepted. For example, another possible ring for the network of figure 21 is shown in figure 22(b). This ring could have been formed if the order of joining stations had been A, E, C, D, B. The two rings of figure 22 do not have the same properties in terms of performance: ring (a) can be said to be "better" than ring (b), in the sense that the total number of hops in a rotation of the token along ring (a) is 6, whereas along ring (b) it is 7. This means that more time is spent transmitting the token in ring (b) than in ring (a), which implies a longer token-rotation time.

7.1.3 Per-hop implicit acknowledgments

Implicit acknowledgments are still used in forwarding the token. Since now the token may have to be relayed over more than one hops, a number of per-hop implicit acknowledgments form an end-to-end implicit acknowledgment. For example, let the token route from A to D be (A, B, C, D). Then, when A hears B forwarding the token to C it assumes the token implicitly acknowledged, when B hears C forwarding the token to D it assumes the token implicitly acknowledged, and so on.

7.1.4 Close-route and close-ring operations

The close-ring operation is extended as follows.

The forwarding of the token can fail at any point along the token route (because a relay station does not hear an implicit acknowledgment after a number of retransmissions) and not just at the originator of the token. If this happens at a relay station other than the originator, this station sends its connectivity information *backwards* (i.e., along the reverse route) to the originator, using a special packet, called conn-info. Every station in the reverse route adds its own connectivity information to this packet. Implicit acknowledgments are used in the transmission of conn-info as well. If the originator receives the conn-info packet, it decides whether to try to *close the route* or the ring, or to kick itself out, if it thinks that the first two are not possible. If the conn-info packet is lost then this is similar to loss of the token: the token will eventually be regenerated by some station.

We illustrate the process through an example. Consider the token route (A, B, C, D, E, F). Say C cannot pass the token to D, but thinks it is connected directly to E. C will send a corn-info packet back to A, along the route (C, B, A). B will add to this packet its own connectivity state (suppose none except C and A). A (the originator), upon receiving corn-info, will check its own connectivity cache, to see whether it should attempt to close the route. Say A is connected only to B. Then it may decide to try to close the route by setting $\texttt{TR}(A) = (A, B, C, E, F)$. It will then re-send the token along this new route. Notice that since the route "shrinks" after each close-route, this operation will eventually terminate.

If A decides not to attempt closing the route, it may try to close the ring. For instance, say B thinks it is also connected to some other station G in the ring (this information is contained in the `conn_info` packet). Then A may try to close the ring with G. A will set $\texttt{NS}(A) = G$, $\texttt{TR}(A) = (A, B, G)$, and will send a set-pred token to G along the new route.

One final extension to the close-ring operation concerns the behavior of stations which might be left out of the ring after a close-ring, but still act as relays for some other station. For example, figure 24, shows a network changing its topology, and the corresponding reconfigurations of the ring. Initially, the ring is as shown in (a). Then B looses its connection to E, but gains connection to G. Next time A tries to pass the token to E through relay station B, the forwarding fails, and therefore B sends a conn-info packet back to A.

Suppose A closes the ring with G, along the route (A, B, G). This means that stations E and F have been left out of the ring. However, they are still in the token route from G to D. Eventually, they will realize that they are no longer part of the ring when their inring-timer expires. However, a better solution is the following.

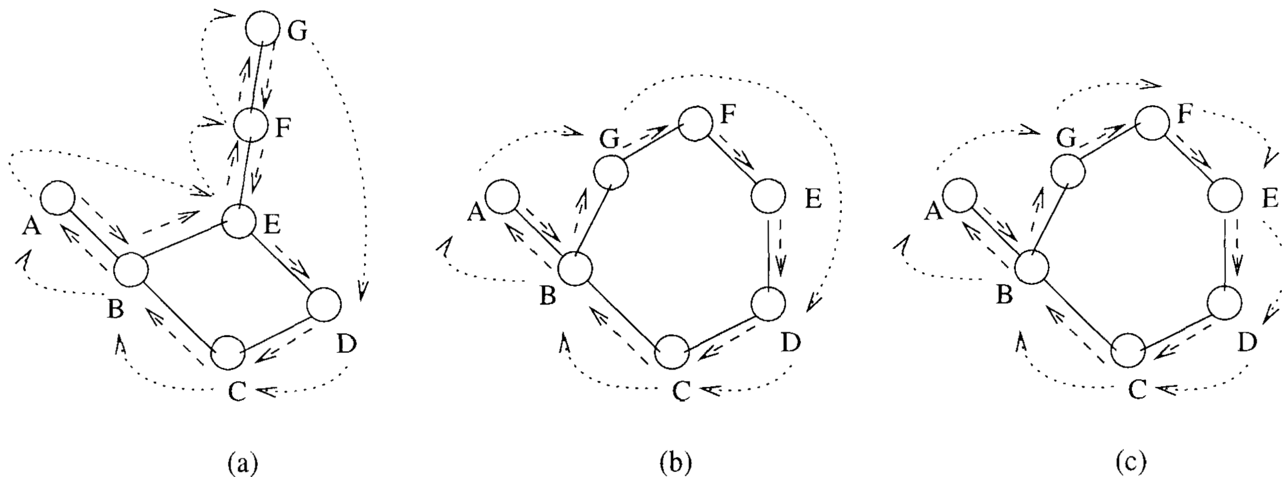


Figure 24: Relay nodes joining during close-ring.

When a station X relays two tokens p and p' in a row, such that the source and destination of p and p' are the same, say S and D, and in the meantime X has not received a token for itself, X assumes that it has been kicked out of the ring. X then **piggy-bags** a message in p', saying that it wants to join the ring. If a token with the same message comes back to X, then X can conclude that it has indeed been kicked out of the ring. X then joins the ring between S and D by sending a message to S along the backward route. We illustrate this in the example of figure 24(b), where nodes F and E have been kicked out when A close the ring with G. F and E will eventually join, resulting in the ring (c) in the figure.

7.1.5 Other details

To be made more precise:

- when do you accept tokens: relay nodes always, others as before
- when do you reset timers: inring/idle reset whenever you forward the token (no matter whether you are relay or not)
- redefinition of MTRT
- must not relay tokens not in my ring

7.2 Properties of the generalized protocol

In this section we argue that the generalized WTBP has the same self-stabilization property as the basic protocol. We also show that the generalized WTBP can form a ring under any topology (provided every station is connected to at least one other station). Finally, we show that joining, leaving and close-ring operations preserve the following property: if the ring has n stations, then the total hop count of token routes is at most $2n$.

7.2.1 Stability

Let us modify the definition of a well-formed ring R as follows. Apart from the fact that for each station $x \in R$, we want $\mathtt{NS}(\mathtt{PS}(x)) = x$, we also require all relay stations to be part of the ring, that is, if there are $x, y \in R$ such that $y \in \mathtt{TR}(x)$, then there exists $z \in R$ such that $y = \mathtt{NS}(z)$.

The stability proof goes through *3* stages, as before:

1. Multiple equivalent tokens get eliminated (same as before).
2. Every station is eventually part of a well-formed ring that does not change (modified, see below).
3. Every ring eventually has a single token (same as before).

Stage 2 is modified as follows. First, we define the following time constant:

$$A \stackrel{\text{def}}{=} \max\{\texttt{MAX_NO_TOKEN_RECEIVED}, 4\texttt{MTRT}\}$$

Then, the following lemma is added:

Lemma 7.1 *Suppose at time T there is a relay node not part of a ring. This node will eventually either join the* `ring` *or be kicked out of the ring,* by *time $T + A$.*

Proof: Suppose a relay X is not part of the ring. Either X will then be relaying a token infinitely often or not. If not, its inring timer will expire by time $T +$ MAXNO-TOKENRECEIVED and X will kick itself out. If yes, X will be relaying a token p going from the same source to the same destination over and over (there is a finite number of nodes). Eventually, X will realize it has been kicked out of the ring and join again. It takes at most 3`MTRT` for p to first reach X, return to X and return again piggy-bagged. It takes another `MTRT` at most for X to join. ■

Lemma 6.6 is modified as follows:

Lemma 7.2 *Suppose there are no multiple equivalent tokens in the ring. Then if a station performs a join operation: it joins a ring R whose owner is also in R. Furthermore, no station in such a ring is ever kicked out. Eventually R becomes well-formed.*

Proof The proof is the same as in lemma 6.6. Relay stations are not kicked out, because they also transmit a token every `MTRT` at most. By this and lemma 7.1 it follows that all relays are eventually part of the ring, therefore, the ring is eventually well-formed. ■

Lemma 6.7 is modified as follows:

Lemma 7.3 *Suppose there is a ring which remains non-well-formed for more than* A *time. Then, some station goes to state "Idle" during this time.*

Proof Let R be a non-well-formed ring during the interval $[T, T']$, where $T' > T + A$. From the definition of well-formed rings, either (a) there is some station $y \in R$ such that $(y.\texttt{PS}).\texttt{NS} \neq y$, or (b) there is some relay station which is not part of R. Consider first case (b). By lemma 7.1, the relay station will either kick itself out by T' or join the ring.

Consider now case (a). Let $x = y.\texttt{PS}$. Either x passes a token to y during the interval $[T,T']$, or y's inring-timer expires by time T', y goes into state "Idle", and we're done. Let t' be the last time x passed a token to y in the interval $[T,T']$. Now, at time t', $x.\texttt{NS} = y$ (since x sends the token to its NS), but at time T', $x.\texttt{NS} \neq y$. The NS of a station changes only when a station tries to close the ring (this cannot happen because of the assumption for no packet loss), or when the station's inring-timer expires and the station moves to "Idle". Therefore, x must go to "Idle" during the interval $[t', T']$. ■

Lemma 6.8 can then be proved based on lemmas 7.2 and 7.3.

7.2.2 Connectivity

Lemma 7.4 *The generalized protocol can form a ring over any network topology, as long as any station is connected to at least one other station.*

Proof the proof is by induction on the number of stations. The lemma is true for one station: we assume that the station forms its own ring. Let the lemma be true for n stations. Let the $n+1$ station be z, and let z be connected to (at least) x. Let $y = \mathtt{NS}(x)$ and let $(x = y_k, y_{k-1}, ..., y_0 = y)$ be the token route from x to y, $k \geq 1$. Eventually, x will send a solicit_successor$(x = y_k, y_{k-1}, ..., y_0 = y)$ token. Since z is connected to x, it receives this token and replies. (Possibly other stations reply as well: eventually, at least one of them will succeed in being received by x, and let that station be z.) Now z can certainly join the ring with token route $(z, x = y_k, y_{k-1}, ..., y_0 = y)$, or shorter. ■

7.2.3 Hop-count of token routes

Let $(x = y_k, y_{k-1}, ..., y_0 = y)$ be the token route from x to y. The *hop-count* of this route is defined to be k. Let $x_1, x_2, ..., x_n$ be the stations in a ring R, and let k_i be the hop-count of the token route from x_i to $\mathtt{NS}x_i$. The *total hop-count* of R is the sum $\sum_{i=1}^{n} k_i$. For example, the total hop-count of the rings (a) and (b) of figure 21 is 6 and 7, respectively.

A *contour* of length m is a sequence of stations, $x_1, x_2, ..., x_m$ such that:

1. for all $1 \leq i < m, x_i \neq x_{i+1}$,
2. for all $1 \leq i < j \leq m$, if $x_i = x_j$ then for all $i < k < j < l \leq m, x_k \neq x_l$.

For example, $1,2,3,4,3,5,3,6$ is a contour, but $1,2,3,4,3,4$ is not.[4]

Lemma 7.5 *The length of a contour visiting* n *different stations is at most* $2n$

Proof: We can prove it by induction on the *nesting* of the contour.

Let $C_0 = x_1, ..., x_m$ be the initial contour. If all stations in the contour are distinct, then $m = n \leq 2n$. Assume there is at least one station which appears twice in C_0. Choose such a station $x_i = x_j$, $1 \leq i < j \leq m$, such that all stations $x_{i+1}, ..., x_{j-1}$ are distinct (there will be at least one such station, by condition 1 of the definition of contour). Now, form a new sequence $C_1 = x_1, ..., x_i, x_{j+1}, ..., x_m$. It is easy to check that this will still be a contour. Since all stations $x_{i+1}, ..., x_{j-1}$ are distinct and no longer appear after x_j in C_0, C_1 visits $n_1 = n - (j - i - 1)$ stations (i.e., $j - i - 1$ less stations than C_0). The length of C_1 is $m_1 = m - (n_1 + 1)$. C_1 is smaller that C_0 in the nesting order, thus, by the induction hypothesis, $m_1 \leq 2n_1 \Leftrightarrow m \leq 3n_1 + 1 \Leftrightarrow m \leq 3n - 3(j - i) + 4 \Rightarrow m \leq 3n - 3 + 4$ (since $j - i \geq 1$). Since $n \geq 1$, we get that $m \leq 2n$. ■

The following lemma says that eliminating some stations from a contour results in a contour.

Lemma 7.6 *If the sequence* $\rho = \sigma_1, x, \sigma_2, y, \sigma_3$ *is a contour, where* $x \neq y$, *then the sequence* $\rho' = \sigma_1, x, y, \sigma_3$ *is also a contour.*

Proof: Assume that ρ' is not a contour. Since $x \neq y$, this must be because there exist z and w, such that $\rho' = \alpha, z, \beta, w, \gamma, z, \delta, w, \epsilon$. But then, ρ wouldn't be a contour either. ■

We next show that the concatenation of token routes in a well-formed ring defines a contour, which is preserved by join, close-route and close-ring operations. Therefore, by lemma 7.5, the total hop-count in a ring of n stations is at most $2n$.

[4] Intuitively, a contour can be mapped to the depth-first traversal of a tree, where each time a new station appears in the sequence, it becomes the child of the current station, and each time an old station reappears in the sequence, the stack is popped up to that station. In such a search, once a station is popped from the stack, it is not visited anymore.

Lemma 7.7 *The contour property is preserved by join, close-route and close-ring operations.*

Proof Consider a ring R of n stations, $x_1, ..., x_n$, such that $\mathtt{NS}(x_i) = x_{i+1}$ (addition modulo n) and $\mathtt{TR}(x_i) = (x_i, y^i_1, ..., y^i_{k_i}, x_{i+1})$, where $k_i \geq 0$, $i = 1, ..., n$. We will show that the sequence $\rho = x_1, y^1_1, ..., y^1_{k_1}, x_2, y^2_1, ..., y^n_{k_n}$ is a contour. This is true for $n = 1$, where the sequence is x_1, the self-ring of station x_1.

Consider the join operation. Let $z = x_{n+1}$ be a station joining between x_i and x_{i+1} and let $\sigma_1 = x_1, ..., x_i$ and $\sigma_2 = x_{i+1}, ..., y^n_{k_n}$. That is, z listens and responds to the solicit-successor of x_i, becomes $\mathtt{NS}(x_i)$ and $\mathtt{NS}(z) = x_{i+1}$. The new sequence will be $\rho' = \sigma_1, z, \sigma, \sigma_2$, where σ is a suffix of $y^i_1, ..., y^i_{k_i}$. Now, by lemma 7.6, the sequence $\sigma_1, \sigma, \sigma_2$ is a contour. Moreover, z appears only once in ρ'. Therefore, $\sigma_1, z, \sigma, \sigma_2 = \rho'$ is also a contour.

Consider now the close-route and close-ring operations. These operations only result in some stations being removed from ρ, thus, by lemma 7.6, the resulting sequence, say ρ'', is also a contour.[5]

■

8 Support for Data Forwarding

Our protocol supports bounded-time medium access. For most applications this is not enough: since they also require bounded-time *data delivery*. Medium-access is equivalent to packet delivery in protocols such as Ethernet or FDDI, since the station that captures the medium is also able to broadcast its data to every other node in the network in at most the maximum propagation time along the network.

In our case, the destination nodes are not generally all within range of the transmitting node (the one that holds the token). Therefore, to reach nodes out-of-range, some routing scheme must be used on top of the MAC protocol. We leave the possibilities for this routing scheme open in this paper. On the other hand, we want our protocol to provide bounded-time data delivery. To achieve this under any reasonable routing scheme, the following modification can be made to the MAC protocol:

- Each station A has two FIFO queues, Q_r and Q_o, where the data packets coming from the higher layer (the routing layer) are stored. Q_o stores the data packets originated in A, whereas Q_r stores the packets originated in some other node, and routed through A.
- When A receives the token, it does the following:
  - It transmits all packets from Q_r, without considering the token-holding timer.
  - After having emptied Q_r, it sets its token-holding timer to THT_{max} and proceeds in transmitting packets from Q_o, until the token-holding timer expires.

If the above algorithm is executed by each station in a network of n stations, and if the routing scheme used is *acyclic* [6], then the following can be shown:

> The delivery time for a data packet arriving at some station, when the queue Q_o of this station is empty, is at most
>
> $$n(n-1)THT_{max}$$

In other words, each station is guaranteed to deliver data to its destinations at a minimum rate of $\frac{1}{n(n-1)}$, and with bounded delivery time for each packet.

[5] Although ρ'' might contain some stations which are not part of the ring, these stations will eventually join, by lemma 7.2.

[6] That is, a data packet originated at some station S follows a path $S = A_1, A_2, ..., A_k = D$, to the destination D such that $A_i \neq A_j$ for $i \neq j$.

A Application Domain: Vehicle Communication

One of the trends in transportation studies is the application of inter-vehicles communication between a vehicle and some infrastructures. The following are two examples of such applications:

- Platoon Control — In coordinated vehicles manuvers such as platooning, communication capabilities lead to the design of better algorithms, and may play a fundamental role in stability. Here, periodic communication among coordinating vehicles is used to avoid shockwaves in the traffic flow due to slow propagation of a perturbation in the leading vehicle trajectory.

- Collision Warnings at crossroads — Collision Warning Sytems can be designed relying on vehicle positioning devices (such as GPS) and communication capabilities. E.g., when approaching a crossroads a vehicle could broadcast a packet containing its position and speed. By receiving informations from the others, a vehicle can assess if its trajectory leads to a potential collision with others, and notify the driver of this eventuality.

The first example requires a periodic communication with bounds on the maximum delay a vehicle experiences before it can send a packet. The communication is point to point in the sense that a vehicle addresses its transmission to another vehicle, even if the radio transmission is intrinsecally broadcasted. Depending on the implementation of the control algorithm, vehicles may or may not have to be fully connected (i.e. each vehicle being in the transmission range of each other). Even if they generally are fully connected, due to the mobility it may happen that a vehicle is temporary out of range of another.

In the second example, we can assume a single bursty transmission when the vehicle is at the proper distance from the crossroad. The proper distance is chosen such that the interested vehicles are intrinsically fully-connected but still distant enough from the crossroad to allow the driver to safely react to the possible warning. As there is no precedent coordination among the vehicles approaching, vehicles don't know each other and they have to broadcast their transmission.

So,to support an ad-hoc network in the previous applications, a MAC protocol must satisfy the following requirements:

- provide Quality of Service for periodic traffic. Here by quality of service we mean that it has to be possible to specify on design time the amount of data a node is allowed to transmit when he gains the right to access the media, and the maximum delay between one access and the next one. Still, the network must be able to provide bursty access when needed.

- efficiently support both point to point and broadcast transmissions

- be able to deal with (possibly temporary) non full connectivity situation

Recently IEEE has standardized the 802.11 wireless medium access control protocol. This protocol is mostly based on a well known medium access scheme called **CSMA/CA.** This protocol may be well suited for implementation of wirless **TCP/IP** networks, but we believe it .not to be so efficient for the ad-hoc applications we are interested in.

In the following section we will give a brief introduction on the CSMA scheme that will allow to underline the reasons of the supposed inefficiency.

B CSMA based schemes

One of the protocols that is the base of many MAC schemes is the so called *Carrier Sense Multiple Access,* that we will explainin assuming (for the moment) a full connectivity situation. Here the idea is that whenever a station has something to transmit, it first senses the channel to check if a transmission is ongoing. If this is the case, the station waits for some time (tipically by means of a random backoff) and then tries again. When the station senses a free channel, it starts its transmission.

Even if the channel is checked, it may happen than two station start a transmission at approximately the same time. In this case, due to the characteristics of the media, in a wired scenario is possibile to immediately sense the collision, and interrupt the transmission. Unfortunately most of the time in a wireless scenario, if all stations transmit with approximately the same power, a station listening to the channel while tranmitting could hear only its own transmission, whose power would locally overwhelm the others.

To avoid collision lasting for a long time, a variant of the algorithm called CSMA/CA can be used. Here when a station A has something to transmit and the channel is sensed as emtpy, the station sends an RTS packet to the destination of the transmission, let's say B. Any station other than B, hearing the RTS packet will became silent. B will reply with a CTS packet. In these steps, a collision may occur between two or more stations sending the RTS packet. The sender will not perceive such collision, but the receiver will never hear the RTS and *so* it will not sent the CTS back. Not hearing the CTS, the sender will backoff and will start the protocol again. If the sender receives a CTS back, then it can start its data transmission safely. The protocol ensures that collision can occur only between RTS packets, and *so* will be limited in time. In case of high utilization of the channel adding RTS/CTS can improve the throughput reducing the time the channel is wasted due to collisions. When the utilization is not so high, the performance of and CSMA/CA can be worse than the simple CSMA, due to the overhead of sending RTS/CTS before each transmission. In the IEEE 802-11 standard, the RTS/CTS can be enabled or disabled, allowing to adapt the protocol to the traffic conditions.

The CSMA scheme can be affected by the lack of full connectivity. In fact, in such situation sensing the channel as free doesn't really mean that no station is transmitting. Suppose we have three stations A, B and C, with B connected to both A and B, but A and C non connected to each other.

if A is transmitting to B, not sensing it C can decide to start its own transmission to B, resulting in a collision. This situation is known as "the hidden terminal" problem.

If sensing a channel as free doesn't necessarily mean that none is transmitting, sensing it busy doesn't necessarily mean that a transmission couldn't be succesfully done. Let's assume there is one more station D, which is connected to A and C. Now, if A is transmitting to B, D will sense the channel as busy. Nevertheless, D could transmit data to C which would receive it correctly. This situation is know as "the exposed terminal" problem.

The CSMA/CA scheme instead works pretty well with a few additions. In this case, the collision may happen in any way between RTS and CTS, but is very unlikely to happen between data and some other packet. Here the idea is that when station A sends the RTS, all the stations hearing it will became silent. Possibly there may be a station which is not in the transmission range of A, but is in the transmission range of B. Not hearing the RTS, such station may send packets while A is sending its data to B. To avoid this, the protocol requires that station hearing a CTS packet will became silent too. So, after a sequence of RTS/CTS has been produced, all the station in range of A or B have become silent. There are particular situations in which this scheme doesn't work, but they are very limited and so the number of collisions on data remains very low.

We can now examine the CSMA/CA scheme with respect to the requisites we have identified for our ad-hoc network.

- QOS for periodic traffic -- The CSMA/CA scheme doesn't provide any bound on the medium allocation delay. The scheduling among nodes which are willing to transmit is purely statistical and depends on the implementation of the random backoff algorithm. Control algorithms require guarantees of support for periodic traffic which can not be provided in this case.

- efficiently support both point-to-point and broadcast transmission – As we mentioned, the CA part of the CSMA/CA scheme can not be applied in broadcast situations, where an RTS could be received by an undefined number of stations. Without collision avoidance, the CSMA scheme becomes much more inefficient. This means that. even if the network designer may be able to associate a probability distribution to the delay in the medium allocation based on the study of the behavior of the scheme, this distribution would be totaly different (and much worst) in case of broadcast transmissions.

- be able to deal with loss of connectivity – CSMA/CA is able to deal with loss of connectivity, but CSMA alone is not. This means that, if the application is known to produce (even temporary) lacks of full connectivity, the RTS/CTS scheme must be always enabled, even in low traffic conditions. As RTS/CTS introduces an overhead, this may affect the performance of the system.

C Comparison with the IEEE 802.4 Token Bus Protocol

C.1 The Token Bus MAC protocol

In a wired scenario, one of the protocols that provides QOS (in the sense previously assigned to this term) is the Token Bus Protocol. A standardization of this protocol was defined by IEEE in the *802.4*[2] specification. In a token bus network a set of stations is connected to a common bus, and at any given time a single station is allowed to transmit. The station is said to "have the token". When the station has no more data to send or when its time slot expires, it sends a special packet (called "token") to a logical successor, which in turn will be able to transmit. The token is passed from station to station in a virtual ring. By knowing the number of station in the ring its possible to compute the time each station is allowed to keep the token so that the period of the channel access is bounded.

A station can transmit in a point-to-point or broadcast fashion without introducing any difference in the performance of the protocol.

So, apparently, two of our requirements seem to be satisfied by the token bus architecture. Unfortunately, the *802.4* specification relies on some assumption that are no longer valid in our case, such as full connectivity, and the ability to detect collisions at the source.

This considerations lead us to the design of a new specification for the token bus protocol specifically targeted to wireless networks. To achieve this result, some of the assumptions of the standard were relaxed, finding new solutions for the parts of the algorithm that resulted to be affected.

First of all, our specification never assumes full–connectivity among nodes. As previously explained, this fact leads to the impossibility to use carrier sensing as a reliable tool to avoid collisions. It will be shown that another implication is that contention among nodes willing to join a ring can't be resolved by any node due to different views preceived by different nodes.

Our protocol never assumes the possibility of receiving an invalid packet. This coincides with the following assumption:
whenever a packet is sent by a station, the physical interface of another station may:

- not receive the packet at all;

- receive the packet, check its integrity, and discard it due to a CRC error;

- receive the packet, verify its integrity and pass it to the MAC layer

There are two reasons for this difference with the 802.4 definition, which explicitly distinguish among correct reception/noise reception/null reception.

First of all, completely loosing packets, which is assumed to be very unlikely in the 802.4 scenario, becomes much more probable in a non fully-connected situation. Also, while in the first case hearing noise is very likeley to mean collision, in a wireless scenario it may simply happen due to external electromagnetic interference. In this case, the packet may be so corrupted that the phyisical layer can't even distinguish it from channel noise.

So, both noise and silence partially loose their strong symptomatology.

Another reason for not distinguishing between invalid packets and silence is purely a simplification of the interface between the MAC and the physical interface. In fact, as most of the off-the-shelf radios perform CRC checking in hardware passing to the upper layers only valid packets, dealing at this level simplifies the implementation of the protocol over different radios, providing a more portable solution.

C.2 Comparison with IEEE *802.4*

C.2.1 Joining a Ring

In the 802.4 standard, a node in the ring outputs a `solicit_successor(A,B)` packet where `A` is the address of the sender and `B` is the address of its successor. As all the node not yet in the ring hear that packet, to reduce the contention only those node whose address lies in the range (`A`, `B`) are entitled to participate in the following phases of the join protocol. Note that this condition implies that the ring grows remaining ordered with respect to the MAC addresses of its nodes.

A node entitled to join replies with a `setsuccessor` packet and then start listening to the channel. If this was the only node willing to join, `A` will clearly hear the `set-successor` and will reply with the token. Hearing the token, the new node will know that it has been admitted in the ring.

If more than one node sent the `set-successor`, then `A` hears only noise, which is (correctly) interpreted as a collision of `set-successor` packets. In this case `A` replies with a `resolve-contention` packet. Hearing such packet, each contender waits from 0 to *3* slots of time, depending on the first two bits of its MAC address. While waiting, the contender listen to the channel; if a `setsuccessor` from other stations is heard, the node loose the contention. If nothing is heard, then the contender sends again its `set-succesor` packet.

Again, it may happen that two or more station having the same first pair of bits in the MAC address collide while sending the `setsuccessor`. In this case, `A` will send again a `resolve-contention` packet and the condenders will use the next pair of bits from their MAC address to try resolving the contention. Assuming that two station can't have the same MAC address, the procedure will lead to a single node joining the ring in a fixed amount of time.

Let's now examine the previous procedure in a wireless scenario where full-connectivity is not ensured.

First of all, reducing the number of contending stations according to the range specified by the address of the `solicit-successor` sender and the address of its successor isn't a reasonable solution. In fact, a station who is ready to join a ring may be in range only of a few stations whose addresses are not correct in the sense of the previously explained scheme.

For what concerns contention resolution, the 802.4 protocol assumes that a node abandon it when hearing other contenders sending their `setsuccessor` packets. This scheme fails when full-connectivity

is relaxed, because two contenders can be in range of the solicit-successor sender but not in range of each other.

It's therefore evident that the only node entitled to resolve the contention is the solicit-successor sender. Unfortunately there is no deterministic way for such node to resolve the contention in a fixed amount of time. If there are node willing to join, the 802.4 protocol ensures that exactly one node joins the ring. Due to the previously explained limitation there are two possible choice while designing the protocol:

- keeping the joining subprotocol bounded in time, accepting the sideeffect that possibly no node will join the ring even if there are node who are willing to;
- relaxing the time bound and using a non deterministic iterative algorithm that terminates when one node joins the ring.

The first solution was choosen in the WTBP design, considering prioritary the QOS provided to the node inside the ring with respect to those not yet in it.

Our join algorithm works as follow:
a node A in the ring sends the solicit_successor(A,B) packet, where B is the address of A's successor. Any node who is willing to join and hears this packet, check its connectivity to B by examining a connectivity cache that was built by listening to the channel (the node is connected to A because it heard its transmission). Those node who are connected to B pick a random delay within a response window and then send a setsuccessor packet to A. After sending the solicit-successor packet, A listen to the channel for the whole response window. During this time A will hopefully hear a valid set-successor packet (possibly preceeded and/or

followed by some colliding packets). If after the response window at least one valid set-successor was heard, one of the sending stations is picked as new successor for A and the token is sent to it. If no valid set-successor is heard, A continues with the normal token ring protocol, passing the token to its successor.

C.2.2 Claiming the Token

In the 802.4 protocol, whenever a station sense an empty channel for a certain amount of time, the token is assumed to be lost and the station contends in the creation of a new token.

Again, in our case, contention among all the stations claiming the token is not possible, because they may not be in range one of each other. Therefore, in addition to resolve the contention, the focus was shifted on implementing a sophisticated algorithm for multiple tokens detection and deletion. Other than for multiple tokens (tokens generated by different stations in the ring), this algorithm is applied also to eliminate duplicate tokens, i.e. copies of the same exact token.

Duplicate tokens can be generated in the following situation: after a station A sends the token to its successor B, it listen to the channel for an implicit acknowledgment of the token reception. The acknowledgment consist of hearing the successor sending some valid packet. Due to the conditions of the wirless media, even without any other node transmitting, the implicit acknowledgment can be lost (e.g. due to electromagnetic interference). In this case, A will assume that the token was never received by B and will try to send the same token again, so generating a duplicate token situation.

To discover and delete multiple or duplicate tokens our protocol assume that the token contains a sequence number (GenSeq) and the address of the generator of the token (RA), which is called "owner of the token". Moreover, each station remembers the GenSeq and RA values of the last token it sent out.

When the owner of the token sends the token, it first increment its GenSeq number. Therefore, this number represent the number of rotation of the token in the ring.

When a station receives the token, it first check if the token's GenSeq value is less than its own GenSeq. If this is not the case, then the token is older than some token that passed through this station, and so it's deleted. If the two GenSeq values are the same, and the RA values are the same, then the token is indistinguishable by the previous one received, and so it's considered to be a duplicate. If the two GenSeq are the same but the RA is different, the ring is in a multiple tokens situation, possibly generated due to a claim token operation. In this case, the station deletes the token if this has a RA less then the one of the previous token sent, so establishing a priority in the tokens according to the MAC address of the generating stations.

C.2.3 Passing the Token

In the 802.4 protocol, after sending the token a station enter in a state where it waits for reactions from the station it transmitted to. The station waits one slot-time, which accounts for the time delay between sending the token frame and the arrival back at the sender of the corresponding response.

During this delay, one of the following branches is taken:

- If a valid frame is heard that started during the response window, the station assumes the token pass was successful.
- If nothing is heard, the station assumes that the token pass was unsuccessful and tries to pass the token again (or passes to another strategy if this was already the second trial).
- if noise or an invalid frame is heard, the station continues to listen for additional transmissions.

As previously explained, in our specification either valid frames are received by the MAC layer or nothing is heard. Due to the bigger probability of noise and packet corruption, the window for the implicit acknowledgment may be bigger than in the wired case.

Suppose station A sends the token to its successor, station B. Even with a bigger window, A may miss **A's** transmissions, e.g. due to some noise corrupting **A's** reception (note that it may be the case that only A's reception is compromised, while the transmission is correctly received by the destination). In this case A will send its token again, again waiting for an implicit acknowledgment. As B already sent the token, it will not forward this second copy of it to avoid duplicate tokens to circulate in the ring. Unfortunately, simply ignoring the second copy of the token wouldn't work, because A would assume a second failure of B and would try to close the ring with B's successor. So, in this situation, B will generate an explicit acknowledgment for A.

Of course: this acknowledgment can get lost too, in which case B will be left out of the ring, and will join it again in the next solicit-successor turn.

D Detailed Description of the Protocol

In this section we give a detailed description of the protocol which maps directly to the implementation. We have used a *hybrid automaton* model extended with C++ code to specify our protocol. We have used the tool *Teja* which takes as input a hybrid automaton model, performs simulations and generates C++ code for various platforms as the implementation of the model. We first explain the model and then describe the WTBP in detail.

D.1 A Hybrid Automaton Model

In a simple description, an hybrid automaton is a paradigm used to modelize systems alternating phases of continuous behavior. Within a phase, the system evolution is regulated by a flow describing continuous evolution of the state in time. The flow is a set of differential equations describing how continuous variables, representing the continuous state, change in time. When this evolution produces certain conditions, or due to external events, the system moves to a different phase, where a different set of equation is regulating its flow. By representing each phase with a state of a finite state machine and by drawing transitions from a state and all the possible subsequent phases, a graphical representation of the system is obtained. This states will be called discrete states to distinguish them by the set of continuous variables.

In addition to continuous variables and discrete states, the status of the system includes discrete variables. Discrete variables are not modified during the continuous evolution of the system, when the finite state machine is sitting in a given state. However, when a transition is taken to move from a phase to another, an action associated with the transition can modify the value of both discrete and continuous variables. From a teorethical point of view, actions are executed in no time[7]

In a more precise definition an hybrid automaton is a finite state machine, characterized by the tuple $\{S, C, O, I, T\}$ where:

- S is the set of discrete states, corresponding to the states of a traditional automaton. A discrete state has a flow associated describing how continuous variable have to evolve in its corresponding phase
- C is the set of the continuous states. A continuous state c_i is a variable representing a real number whose evolution is determined by the flow applied in the current discrete state.
- O is the output of an automaton. It's composed by a tuple $\{V, L, F\}$, where:
  - V is the set of (discrete) variables. In Teja's model a variable can be of any C/C++ standard or user-defined type
  - L is the set of links. Links are reference to other automaton instances.
  - F is the set of functions. Functions provide an abstract interface to the internal state of an automaton, but they can also be used to factorize code shared by actions of different transitions.
- I is the set of the input variables. An input variable of an automaton A_i is simply an alias for a discrete variable of a different automaton A_j.
- T is the set of transitions, where $t_{a,b} \in T$ is a transition going from the discrete state a to the discrete state b. Each transition has a guard. The guard may be a boolean condition or the comparison of a continuous variable with a given value. Whenever a boolean guard evaluates to `true`, or the continuous variable passes the comparison value, the transition is said to be enabled.[8]

  Transitions can produce output events. An output event is an event issued by the automaton, directed to one or more automata. The event carries a state which can be set in the issuing

[7] When running a simulation based on hybrid automatons, actions execution doesn't consume simulation time. However, when the hybrid system paradigm is applied to realtime control, actions are implemented in terms of code that requires a certain time to be executed.

[8] Teja's guards can be more complex and are always evaluated into a time delay before the transition can be enabled, but for the sake of presenting our model the give description is precise enough.

transition, and is characterized by an event label. An output event generated by an automaton is propagated to other automata according to a dependency list. Each automaton has a different (possibly empty) dependency list for each event label, including all the automata who should be interested in the event. How receiving automata reacts to the propagated events depend on the current discrete state they are in and the type of transition exiting from that state.

The action of a transition can also produce an *Alert*. There is a similarity between alerts and events, in the sense that they're both mechanism that can be used to communicate informations across automata. However, alerts are generated by one automaton running in a process and addressed to an automaton running in a (possibily) different process. Output events are multicast (generated by an automaton and propagated to many) and synchronous (actions of transitions triggered due to the event reception are executed immediately after the action of the triggering transition, and in simulation no time passes), while alerts are point-to-point (automaton to automaton) and asynchronous, because there is no common clock shared by different processes.

There are three type of transitions:

- proaction: if enabled, the transition can be taken at any time, without the need of any event/alert
- reaction: A reaction is characterized by an event label which identifies the event type the reaction is interested in. To be taken, the transition need to be enabled and an alert from an automaton running in a (possibly) different process must be sent to the this automaton. The alert must have the same label characterizing the reaction.
- response A response is characterized by an event label which identifies the event type the reaction is interested in. To be taken, the transition need to be enabled and an event with the same label must be generated by an automaton A_j running in the same process as the automaton A_i containing the response. A_j must have A_i in its list of dependents for the specified event label.

Each transition has an associated *action*. The action is a sequence of statements that can modify the automaton in different ways. Among the possible actions there are:

- reset continuous or discrete variables and links;
- creating new components;
- creating/destroying connections to processes;
- sending Alerts to connected processes;
- modify event propagation;
- running generic C/C++ code

D.2 Hierarchical specification

The MAC protocol here proposed can be seen at an high-level view, by distinguishing a certain number of macro-states of the system. Example of this states are the "join state", where the stations tries to enter in a ring, or the "have-token state", where the station has received the token and uses it. Each macro-state can be exploded in a hybrid automaton describing the details of the behavior of the system in this particular state. This hierarchical view is very convenient because it simplyfies designing a clean, understandable and maintainable specification. To implement this hierarchical design methodology, appropriate design conventions were adopted.

Every automaton corresponding to a macro state has a state called `start`.

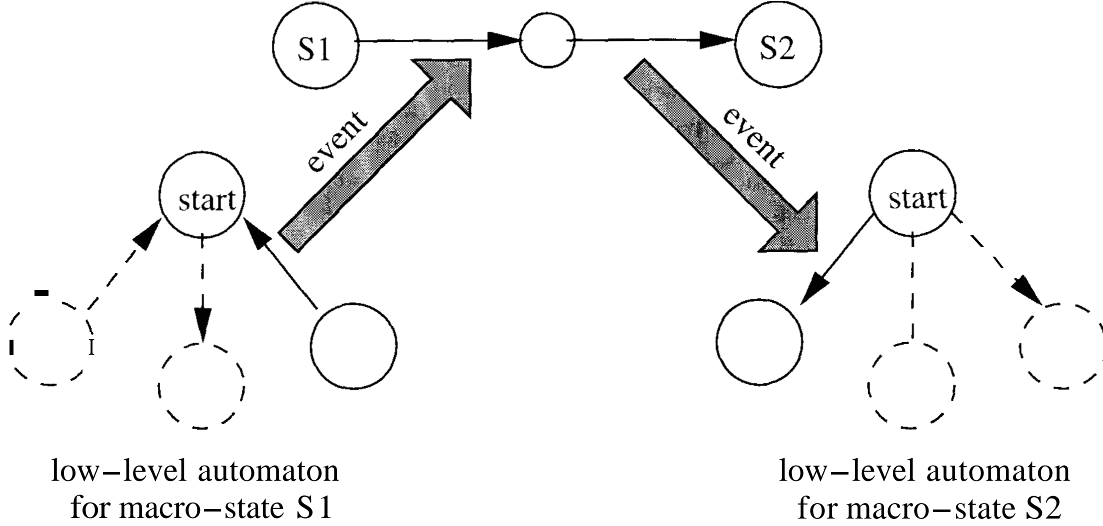


Figure 25: Hierarchical design for hybrid automata

Let A_i be the automaton corresponding to the ith macro state, and $T_{a,b}$ be the set of transition going from state a to state b. Let also $state_t(A_i)$ be the state of the automaton A_i at time t. The following applies:

- At any given time during the continuous evolution of the system it exist one and only one i s.t. $state_t(A_i) = start$
- Whenever an automaton A_i takes a transition from a state to the *start* state, this transition is synchronized with a corresponding transition that takes another automaton out of its *start* state.

This scheme allows to implement a complex automaton in terms of many simple automata. Also, the state the corresponding macro-automaton is in corresponds to the only automaton not being in its *start* state.

A possible way of implementing this scheme is the following: the system is composed by the macro-automaton and each of the low level automata. Whenever the low-level automaton A_i corresponding to the macro state i takes a transition entering in the *start* state, such transition outputs an event directed to the high-level automaton. This event triggers a transition from the macro state i to a macro state j, which in turns generate an output event directed to the low-level automaton A_j. This event takes the automaton A_j out of its start state. Fig 25 shows an example of the three-automata synchronization scheme. In this figure and in the following ones, these conventions were adopted:

- for responses, the input event label is shown on the transition arc
- if an output event is generated by a transition, its label is shown on the transition arc prefixed by a "/" (slash) sign.

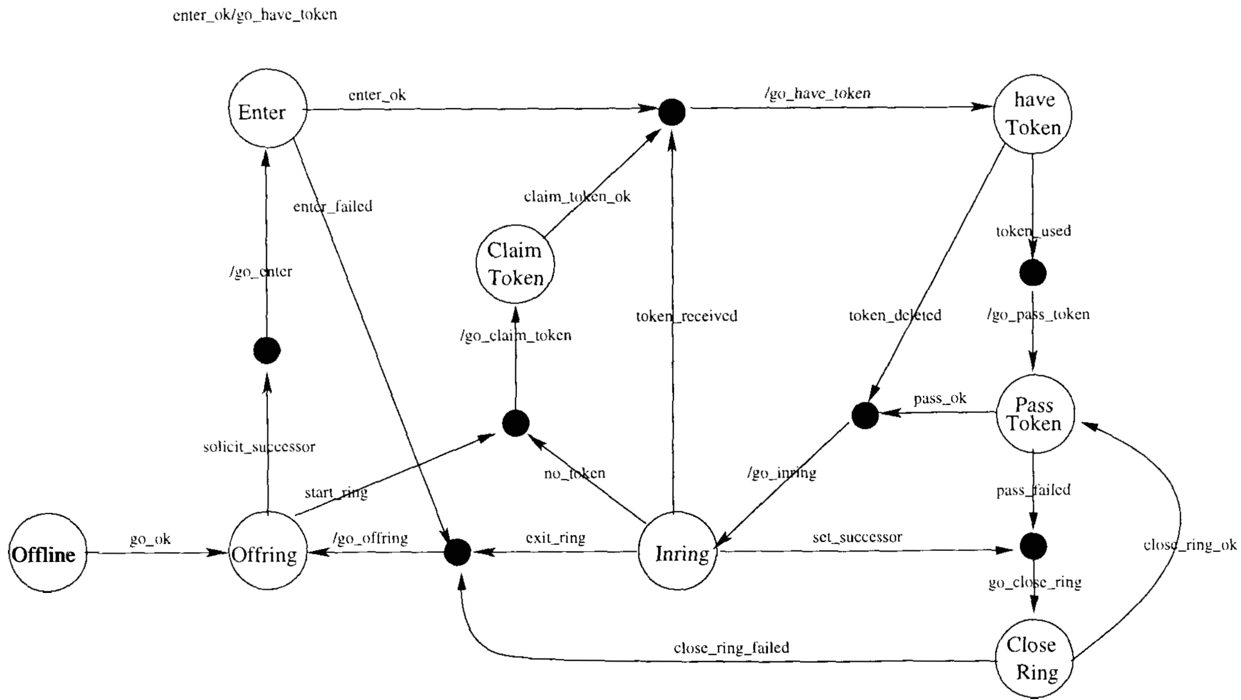


Figure 26: Wireless Token Bus high level FSM

Observing the figure you may have noticed that instead of just taking a single transition, the macro-automaton takes a two-transitions step to move from state $S1$ to state $S2$. The reason for this is related to Teja's model. In fact, Teja's model doesn't allow a response to generate an output event, so in the real implementation the transition in the high level automaton is splitted into two transitions with an intermediate state. The first transition is the response triggered by the event generated by the first low-level automaton entering in the *start* state, while the second transition is the one outputting the event to take the second low-level automaton out of its *start* state. The intermediate state is called transient, because it's a temporary state where the automaton is not supposed to spend any time (at least in simulation).

D.3 High-level automaton

In this section a description of the macro-automaton showing the high-level view of the WTBP will be provided. The reader will notice that the automata shown here are different than the ones shown in section 5. In fact, the two specifications are equivalent, and we have used the latter in the main text to make the description and the proof easier to follow.

The (high level) macro-automaton is shown in Fig 26 on page 44 . Events used to synchronize the low level automata with the macro automaton are shown on the transitions.
A description of each state follows:

- `Offline` — The station is not active.

  The station will leave this state when turned on, after a short initialization period, going to the `Offring` state.

- Offring — The station is active, but it's not part of any ring. In this phase the station will listen to the channel to gain a knowledge about the topology of the network. In particular, the station will possibly build an internal representation of its connectivity to any ring. This knowledge can be used later to take decisions about what ring to join and in what position.

  The station can leave this state due to two different events.

  The first event is a solicitsuccessor packet broadcasted by another node being in a ring. In this case, and if it is interested in joining the ring of the sender of the packet, this node will move to the Enter state, where the protocol to join a ring is carried on.

  The second event is an indication from an upper layer in the protocol stack to create a new ring. In this case the station will move to the Claim-token state.

- Enter — Periodically some node in a ring may broadcast a solicit-successor packet. The role of this packet is to open the possiblity for stand-alone nodes to join the ring. If a node in the Offring state is interested in joining a ring, it will enter in the Enter state.

  The sub-protocol active in this state may fail, in which case the station will go back to the Offring state, or succeed, in which case the station will move to the Have-Token state.

- Inring — The station is inside a ring, and it currently doesn't have the token. In this state the station basically waits for a packet to move to a different state. Actually, the station will listen to the channel updating its knowledge about its current connectivity on the basis of the packets heard.

  From this state, the station can go to many other states.

  If a token is received, the station goes to the Have-Token state.

  Due to an indication coming from the upper layers of the protocol, or having noticed that it has been thrown out of the ring, the station can move to the Offring state.

  If for a time $T_{offring}$ this token's ring is heard but never received, this node assumes that it has been left out of the ring and it moves to the Offring state.

  If for a time $T_{notoken}$ this token's ring is not heard, then this node assumes that the token has been lost and moves to the Claim-Token state, where it will generate a new token.

  If for some reasons the successor of a node A decides to leave the ring gracefully, A may be asked to attempt to close the ring with one of the nodes following **A's** successor. In this case A will receive a set-successor packet and will move to the Close_Ring state.

- Have-Token — In this state the station has the token and so it's entitled to transmit data packet for a certain amount of time (Token Holding Time). If the station has no data to transmit and there is enough time to complete such subprotocol, a solicit-successor packet will be broadcast and the subprotocol will be carried on.

  When the station has completed sending data or soliciting a successor, or if the token holding time has expired, it moves to the Pass-Token state.

- Pass-Token — The station is passing the token to its successor. After sending the token, the station waits for an implicit aknowledgement from the successor. The implicit acknowledgement consists of hearing the successor sending the token to the next station. If the acknowledgement is not heard, the station sends the token again. If again the token is not heard, the station assumes that its successor is unreachable and moves to the CloseRing state. Note that even if the successor has received the token, the implicit acknowledgement can be missed by its

predecessor due to a garbling of the transmission. This garbling may happen even if there are no other stations transmitting, due to some external electromagnetic disturbance. This situation would result in the generation of multiple tokens in the ring. To reduce this probability, the station may decide to wait some more time and listen to the channel, hoping to hear an implicit acknowledgement from any of its successor's successor.

If the token passing succeed, the station moves to the Inring state.

A special case is handled in this state. When the station has just created a new ring and therefore is alone in it, the token doesn't really need to be passed. To keep the model simple and avoid generating a special set of states to cope with this situation, the station simulates sending the token to itself (there is no real transmission involved).

- CloseRing — The station have to try closing the ring. This may be due to two different cases.

  In the first case, the station was unable to send the token to its current successor (i.e. no implicit acknowledgment was received). Here the station will set its successor to the next station and will move to the Pass-Token state again. To accomplish this operation, the station has to know the address of the successor's successor. This knowledge comes from the connectivity information which are collected while the token is rotating in the ring (see Inring state). Due to a bad connectivity, the station could be unable to close the ring without leaving '(too many" nodes out. If this is the case, the station may decide do kick itself out of the ring, passing the task of closing the ring to its predecessor. This leads to the second case of closing the ring: the station receives a set-successor packet coming from its successor because this last one is leaving the ring. In this case the packet contains an indication of the number of times the set-successor has been sent backward, so that the receiving station knows who in its image of the ring it should try to close the ring with.

- Claim-Token — The station enters in this state either because it was in a ring and the token was lost, or because a new ring containing only this node has just been created (see Pass-Token state for further information about this special case).

D.4 Low-level automata

In this section the detailed description of the protocol will be given by showing each of the low-level automata representing a macro-automaton. For automata figures, The same graphical convention previously adopted for indicating input and output events will be kept. Moreover, some transition will be have a slanted label associated to it. This doesn't correspond to any event, but it's just used to refer to the transition within the document.

D.4.1 Offring Automaton

When a station is initially turned on, after an initialization phase it moves to the offring macro-state, triggering the transition that moves this low-level automaton to the offring state. Two events may happen taking the low-level automaton back to the start state:

- A solicitsuccessor frame broadcasted by a node in a ring may be received. In this case the automaton goes to the start state generating a solicitsuccessor output event. This event will start the chain of transitions taking th macro-automaton from the offring to the enter state and the corresponding low-level automaton from the start to the demand-in state.

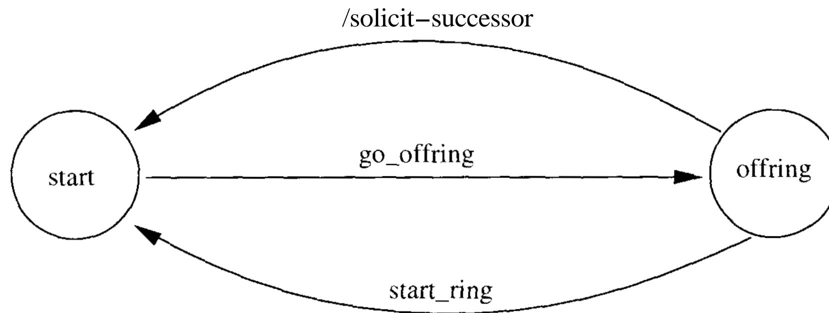


Figure 27: Low-level automaton for macro state Offring

- an indication from an upper layer is given to the MAC to create a new ring all alone. This event will start the chain of transitions taking th macro-automaton from the offring to the claim-token state and the corresponding low-level automaton from the start to the claim-token state.

D.4.2 Inring Automaton

While a node is inside a ring waiting to receive the token the macro-automaton is in the inring state. The corresponding low-level automaton is usually in the inring state. Here follows the description of each state:

- inring – In this state the automaton waits for any frame coming from the ring, or for a timeout. The outgoing transitions are:
  - got-frame – triggered by the reception of a frame from this ring
  - no-token – if no token is heard by this station for at most MAX-IDLE-TIME, then the token is assumed to be lost and this transition is triggered. This transition output a no-token event taking the macro-automaton to the claim-token state.
  - exit-ring – if no token is received by this station for at most MAX_NO_TOKEN_RECEIVED, the station assumes it has been left out of the ring. As MAX-IDLE-TIME is less than MAX_NO_TOKEN_RECEIVED, if the timer associated to the latter expires it means that the one associated to the first didn't, i.e. the token hasn't been received, but it has been heard circulating in the ring. This transition outputs the exit_ring event taking the macro-automaton to the offring state.
- got-frame – the automaton enters this state when a frame from the ring is heard. If the frame is addressed to this station, the automaton moves to the for-us state, otherwise it moves back to the inring state.
- for-us – the frame heard is addressed to this station, and it must be processed differently according to its type. The outgoing transitions from this state are:
  - valid-token – the frame is a token and it's coming from this station's predecessor. In this case a token_received event is output that takes the macro-automaton to the have-token state.

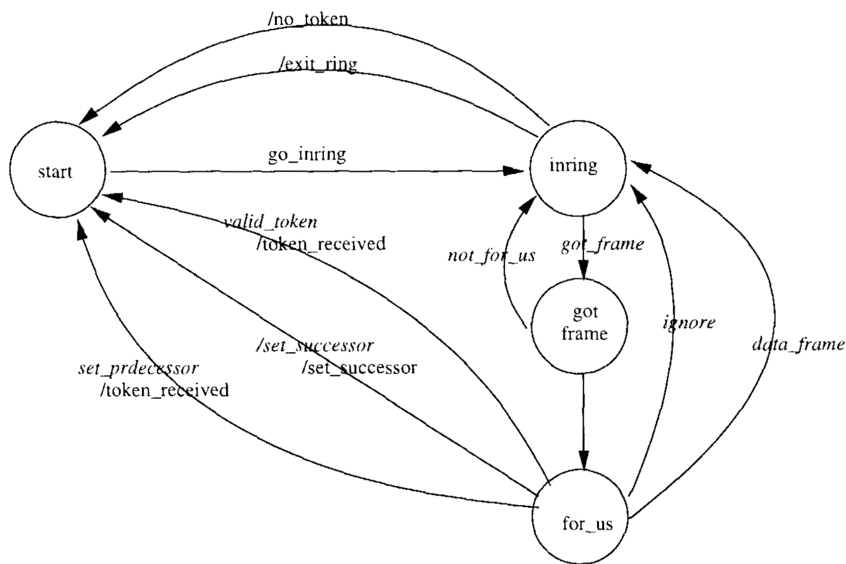


Figure 28: Low-level automaton for macro state Inring

- set-successor – the frame is a setsuccessor, i.e. this station is asked to change its successor. The transition outputs a set-successor event taking the macro-automaton to the closering state.
- set-predecessor – The frame is a valid set-predecessor, i.e. it's a set-predecessor and it has a GenSeq value equal to, or one more than the station's GenSeq. The transition outputs a token_received event that takes the macro-automaton to the have-token state.
- data-frame – the frame is a data packet, and it's enqueued to the upper layer.
- *ignore* – the packet is none of the above and so it's ignored

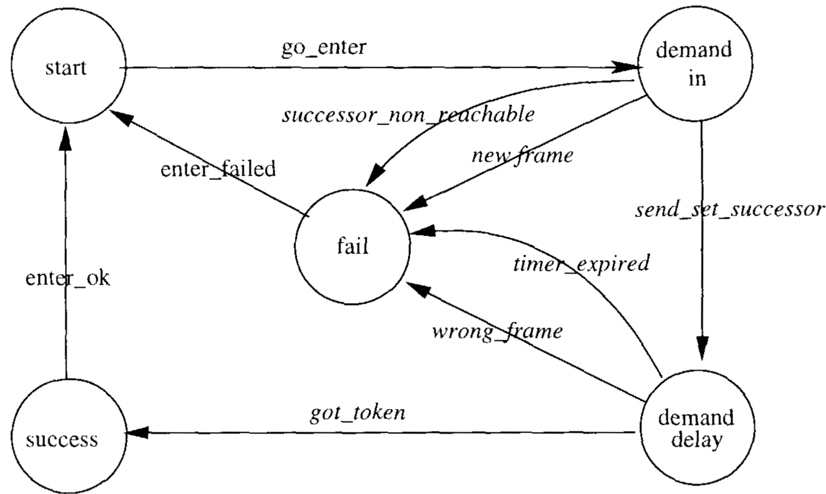


Figure 29: Low-level automaton for macro state Enter

D.4.3 Enter Automaton

When a node out of a ring and willing to join has received a solicitsuccessor packet, its macro-automaton moves to the enter state, and the enter low-level automaton moves to the demand-in state. A description of states and outogoing transitions from each state follows.

- demand-in – after receiving the solicitsuccessor, the station replies with a set-successor to notify its desire to join. However, to reduce the probability of collisions, nodes contending in joining are de-synchronized by delaying the set-successor by a random amount of time between 0 and MAX-CONTENTION-TIME. this is the state where the node waits for the random delay, unless some other condition applies or event happens (see outgoing transitions). Outgoing transitions are:
  - *successor_non_reachable* – before starting to wait, the connectivity to the successor mentioned in the solicit-successor packet is checked. If this node appears not to be connected to the successor, then this transition is taken to the `fail` state.
  - send-set-successor – the random delay has expired and the station is free to send its set-successor packet, moving to the demand-delay state.
  - *new_frame* – while waiting for the random delay, a frame from the same ring of the node who sent the solicitsuccessor is heard. This condition leads to the abortion of the attempt to enter, because it's symptom of activity in the ring.
- demand-delay – in this state the station waits for the expiration of the time window dedicated to the entering process (`MAX_ENTERING_TIME`). Outgoing transitions are:
  - got-token – if a token from the node who sent the solicit-successor arrives before the timer expires, the station has successfully joined the ring. The transition goes to the success state.
  - wrong-frame – if a frame from the ring other than the token arrives, the joining has failed.

- timer-expired – If the automaton has spent MAX_ENTERING_TIME in the demand-delay state, the joining has failed.

- failed – this is a transient state with a single transition that outputs the enter_failed event taking the macro-automaton to the offring state.
- success – this is a transient state with a single transition that outputs the enter-ok event taking the macro-automaton to the have-token state.

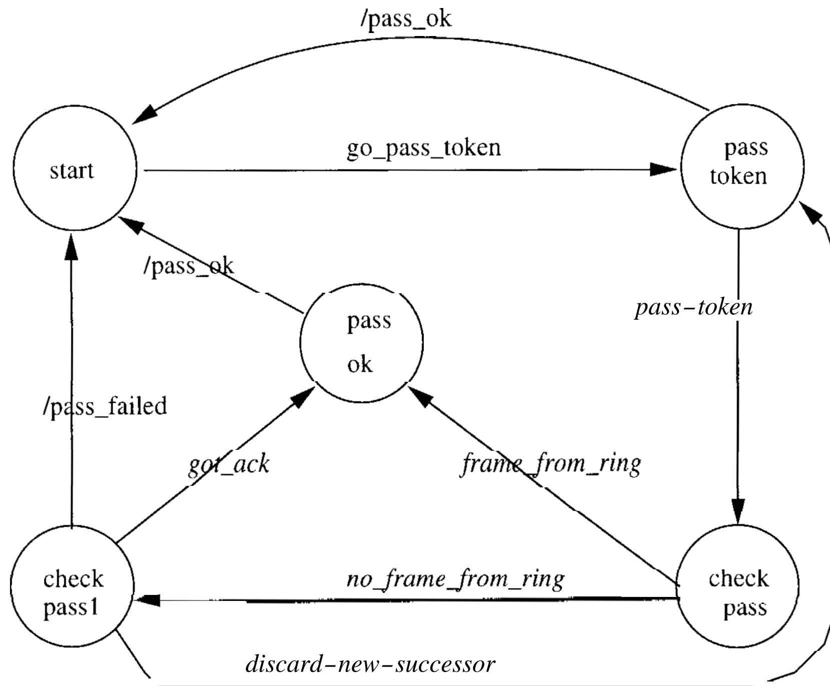


Figure 30: Low-level automaton for macro state Pass-Token

D.4.4 Pass-Token Automaton

This automaton attempts to pass the token to the next station in the ring. A description of states and their outgoing transitions follows.

- pass-token – the station is ready to pass the token. This transient state has a single outgoing transition:

  pass-token that sends the token to the successor. If the station has just joined or closed the ring, a set-predecessor token is sent, otherwise a normal token is sent.

- check-pass – once the token has been sent, the station waits for at most TOKENPASS-TIME listening to the channel. Outgoing transitions from this state are:
  - *frame_from_ring* – if a frame is heard from the ring, the station assumes that either the token has reached the successor and this one has started some transmission (implicit acknowledgment), or some other node in the ring has the token (multiple tokens). In any case, the station doens't try to send the token again.
  - *no-frame-from-ring* – If the timer expires without receiving any frame from the ring, than the transmission of the token has failed and a retransmission is attempted in this transition.

- check-pass1 – this state is equivalent to the previous one, except that if again no acknowledgment is heard, the successor is assumed to be unreachable and, a transition outputting the pass-failed event is taken. This event takes the macro-automaton to the closering state.

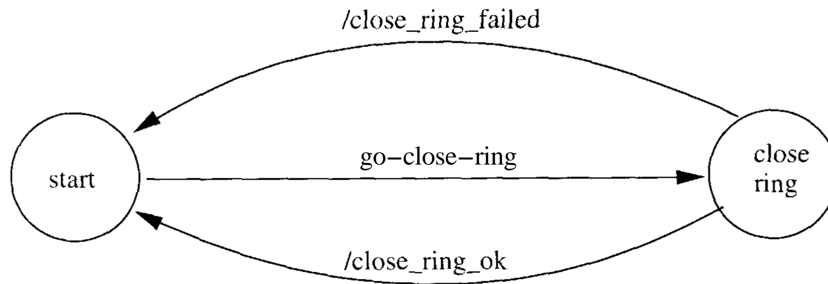


Figure **31:** Low-level automaton for macro state CloseRing

D.4.5 Close-Ring Automaton

This automaton takes care of finding an appropriate node to close the ring with. The station can reach this macro-state due to the reception of a set-successor packet or due to the impossibility to pass the token to the successor. One of the two following conditions is possible:

1. a "suggested successor" is available. This could be the NS field of a received setsuccessor or the node immediately after the current successor if it was impossible to pass the token. In this case the station checks the connectivity to the suggested node, and outputs the close_ring_ok event if the node is connected, the close_ring_failed otherwise. The close_ring_ok event takes the macro-automaton to the pass_ring state, while the close_ring_failed takes it to the offring state.

2. a number of nodes to be skipped when searching a successor is available. This situation happens if the current successor volunteerly left the ring. As such node doens't know the connectivity of its predecessor, instead of sending a set-successor frame indicating its successor, it specifies the number of nodes to be skipped by the predecessor when searching a new successor. The first set-predecessor specifies one, but if the predecessor is unable to close the ring he will send the same packet with a value of two, and so on.

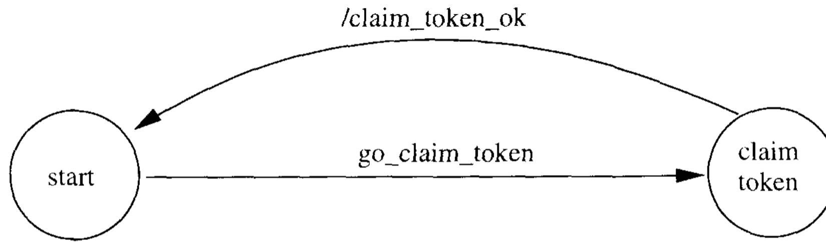


Figure 32: Low-level automaton for macro state Claim-Token

D.4.6 Claim-Token Automaton

This very simple automaton just generates a new token according to the current value of the station state (RA, GenSeq, Seq). The station may enter in this state because the previous token was lost or because the station just formed a ring on its own.

excluding the start state, the only state of this automaton is claim_token. It's a transient state, and the only outgoing transition outputs an event that takes the macro-automaton to the have-token state.

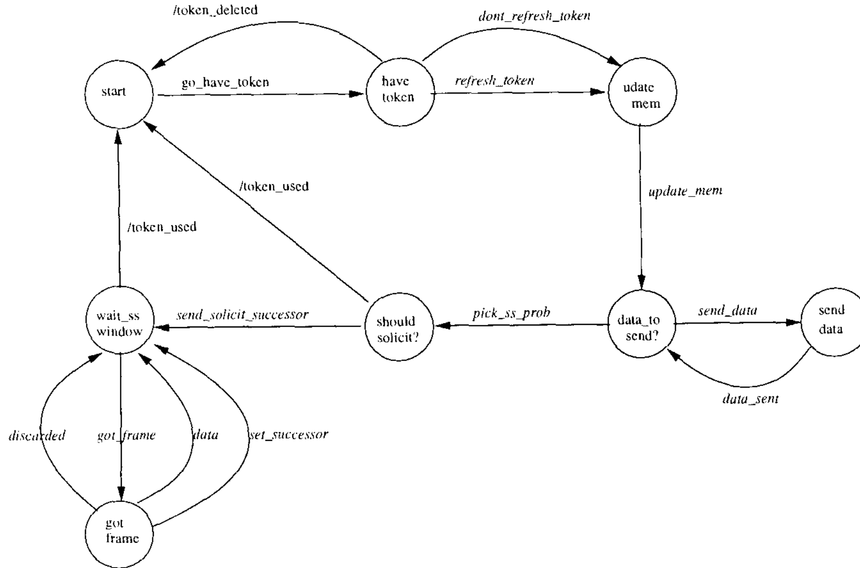


Figure **33:** Low-level automaton for macro state Have-Token

D.4.7 Have- Token Automaton

In this state the station may transmit data or solicit new nodes to join the ring. States and their outgoing transitions are:

- have-token – this transient state is reached when a token is received or claimed. Outgoing transitions are:

  token_deleted – if the token does not satisfy certain constraints, it's deleted and the token-deleted event is output taking the macro-automaton to the inring state. Here follows the set of conditions that applies:

  1. the token GenSeq is less than the station GenSeq (the token is old), and the token RA is equal to this stations' TS (it was generated by this station);
  2. the token has the same value of GenSeq of the station or less (the token is old) and it's coming from this station's ring;
  3. the token has the same value of GenSeq of the station or less (the token is old) and it has a ring address lower than this station's ring address;

  If any of the previous condition was true, this transition is taken and the token is deleted.

  - refresh-token – if the token wasn't deleted and this station is the one who generated it (the token's RA matches this station's TS), then the token must be refreshed. Refreshing the token means increasing it's GenSeq number by one.
  - no-refred-token – if the token wasn't deleted but was not generated by this station, let's just move to the updatemem

  updatemem this transient state is just used to join the two previous transitions into the *update-mem* transition, where part of the state of the station (Seq and GenSeq) is updated according *to* the token content.

  data-to-send? – in this state the station check if there is any data packet to send.

- send-data – if there is at least one data packet to send, and there is enough time to do it, the packet is sent. As there is a maximum time the station is allowed to keep the token (MAX-TOKENHOLDING-TIME), the data transmission must fit in what remains of that time. This implies that data packet as seen by the MAC layer must fit in MAX-TOKENHOLDING-TIME minus the time required to reach the data-to-send? state from when the token is received and the time required to send the token once and obtain the implicit acknowledgement.
- *pick_ss_prob* – once all data packets available have been sent or there is no more time to send them, the this transition picks a random value used in the should-solicit state.

- send-data – the MAC waits in this state for the time that it takes to the physical layer to send the data packet, and then it goes back to the data-to-send? state [9].

- shouldsolicit? – after data has been sent, the station may decide to let one other station join.
  - send-solicit-successor – if enough time remains for a solicit successor phase, and with probability P_SOLICIT_SUCCESSOR, this transition is taken. The probabistic behavior is obtained by means of the number randomly extracted in the pick-ss-prob transition. This transition sends a solicit-successor packet and takes the automaton to the wait-ss-window state.
  - *token_used* – if no solicit successor phase is started, this transition is taken. The transition outputs a token-used event taking the macro-automaton to the inring state.

- wait-ss-window – once a solicitsuccessor packet is sent, the station waits in this state for at most SOLICIT_SUCCESSOR_TIME.
  - got-frame – if a frame is received, the automaton moves to the got_frame state, where it recognizes the packet.
  - token-used – when SOLICITSUCCESSOR-TIME has passed, the automaton moves back to the start state with this transition. A token-used event is output taking the macro-automaton to the pass-token state.

- got-frame – if the received frame is a data frame for this station, it's passed to the upper layer; if it is a set-successor for this station, then it's assumed to be sent by one of the node willing to join, and that node's id is set as the new successor. Any other packet is ignored. In any case, the automaton comes back from this transient state to the wait-ss-window state.

References

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

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

[3] *The Teja Technical Reference* — Teja Technologies Inc. — www.teja.com

[4] http://robotics.eecs.berkeley.edu:80/bear

[5] http://www.metricom.com

[9] in the current implementation, the MAC layer and the physical layer are decoupled by means of queues, and the MAC waits in this state by means of a timer. However, to be able to correctly provide a quality of service, the MAC must be synchronized with the activity at the physical layer. Implementing the correct synchronization requires just the replacement of the timer with a signal provided by the physical layer notifying that the data packet has been sent.

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