LonTalk® Protocol Specification

Version 3.0

ECHELON®
CORPORATION

078-0125-01A

Document No. 19550

Printed in the United States of America.
Copyright ©1994 by Echelon Corporation.

Echelon Corporation
4015 Miranda Avenue
Palo Alto, CA 94304, USA

1. INTRODUCTION....7
1.1 Scope And Objectives....7
1.2 Document Overview....7
2. TERMINOLOGY AND PROTOCOL OVERVIEW....8
2.1 Terminology....8
2.2 Overview of LonTalk Protocol Layering....9
3. NAMING AND ADDRESSING....12
3.1 Address Types and Formats....12
3.2 Domains....12
3.3 Subnets and Nodes....13
3.4 Groups....13
3.5 Neuron_ID....14
3.6 NPDU Addressing....14
3.7 Address Assignment....16
4 MAC SUBLAYER....17
4.1 Service Provided....17
4.2 Interface to the Link Layer....17
4.3 Interface to the Physical Layer....18
4.4 Collision Detection Notification....19
4.5 MPDU Format....20
4.6 Predictive p-persistent CSMA — Overview Description....20
4.7 Idle Channel Detection....21
4.8 Randomizing....22
4.9 Backlog Estimation....23
4.10 Optional Priority....23
4.11 Optional Collision Detection....24
4.12 The Predictive CSMA Algorithm....25
4.13 Timing....27
5. LINK LAYER....29
5.1 Assumptions....29
5.2 Service Provided....29
5.3 LPDU Format....29
5.4 The Transmit Algorithm....30
5.5 The Receive Algorithm....30
5.6 Differential Manchester Encoding....31
6. NETWORK LAYER....32
6.1 Assumptions....32
6.2 Service Provided....33
6.3 Service Interface....34
6.4 Internal Structuring of the Network Layer....35
6.5 NPDU Format....35
6.6 Address Recognition....36
6.7 Routers....36
6.8 Routing Algorithm....37
6.9 Learning Algorithm — Subnets....38
7. TRANSACTION CONTROL Sublayer....39
7.1 Assumptions....39

7.2 Service Provided 39
7.3 Service Interface 40
7.4 State Variables 40
7.5 Transaction Control Algorithm 41

8. TRANSPORT LAYER 43
8.1 Assumptions 43
8.2 Service Provided 43
8.3 Service Interface 43
8.4 TPDU Types and Formats 44
8.5 Protocol Diagram 46
8.6 Transport Protocol State Variables 46
8.7 The Send Algorithm 47
8.8 The Receive Algorithm 49
8.9 RR Pool Size and Configuration Engineering 51
8.10 Number of Retries 51
8.11 Choice of Timers 53

9. SESSION LAYER 54
9.1 Assumptions 54
9.2 Service Provided 54
9.3 Service Interface 54
9.4 Internal Structure of the Session Layer 55
9.5 SPDU Types and Formats 56
9.6 Protocol Timing Diagrams 58
9.7 State Variables 60
9.8 Request-Response Protocol — Client Part 60
9.9 Request-Response Protocol — Server Part 62
9.10 Request-Response Protocol Timers 66
9.11 Authentication Protocol 66
9.12 Encryption Algorithm 67
9.13 Retries and the Role of the Checksum Function 68
9.14 Random Number Generation 69
9.15 Using Authentication 69

10. PRESENTATION / APPLICATION LAYER 70
10.1 Assumptions 70
10.2 Service Provided 70
10.3 Service Interface 70
10.4 APDU Types and Formats 71
10.5 Protocol Diagram 72
10.6 Application Protocol State Variables 74
10.7 Interactions Between the Offline State and Request - Response 77
10.8 Error Notification to the Application Program 77
10.8.1 Error Notification for Messages 77
10.8.2 Error Notification for Network Variables 77

11. NETWORK MANAGEMENT AND DIAGNOSTICS 79
11.1 Assumptions 79
11.2 Services Provided 79
11.3 Network Management and Diagnostics Application Structure 80
11.4 Node States 80

11.5 Using the Network Management Protocol 81
11.5.1 Addressing Considerations 81
11.5.2 Making Configuration Changes 81
11.5.3 Downloading An Application Program 82
11.5.4 Error Handling Conditions 82
11.6 Using Router Network Management Commands 83
11.7 NMPDU Formats and Types 84
11.7.1 Query ID 85
11.7.2 Respond to Query 86
11.7.3 Update Domain 86
11.7.4 Leave Domain 87
11.7.5 Update Key 87
11.7.6 Update Address 88
11.7.7 Query Address 89
11.7.8 Query Network Variable Configuration 89
11.7.9 Update Group Address 90
11.7.10 Query Domain 90
11.7.11 Update Network Variable Configuration 91
11.7.12 Set Node Mode 91
11.7.13 Read Memory 92
11.7.14 Write Memory 92
11.7.15 Checksum Recalculate 93
11.7.16 Install 93
11.7.17 Memory Refresh 94
11.7.18 Query Standard Network Variable Type 95
11.7.19 Network Variable Value Fetch 95
11.7.20 Service Pin Message 96
11.7.21 Network Management Escape Code 96
11.7.22 Router Mode 97
11.7.23 Router Clear Group or Subnet Table 97
11.7.24 Router Group or Subnet Table Download 98
11.7.25 Router Group Forward 98
11.7.26 Router Subnet Forward 98
11.7.27 Router Do Not Forward Group 99
11.7.28 Router Do Not Forward Subnet 99
11.8 DPDU Types and Formats 100
11.8.1 Query Status 101
11.8.2 Proxy Status 103
11.8.3 Clear Status 104
11.8.4 Query Transceiver Status 104

12. BEHAVIORAL CHARACTERISTICS 106
12.1 Channel Capacity and Throughput 106
12.2 Network Metrics 107
12.3 Transaction Metrics 108
12.4 Boundary Conditions — Power-Up 109
12.5 Boundary Conditions — High Load 110

13. Appendix A — PDU Summary 111

This page is intentionally left blank

1. INTRODUCTION

1.1 Scope And Objectives

The LonTalk protocol is designed for communication in control networks. These networks are characterized by short messages (few bytes), very low per node cost, multiple communications media, low bandwidth, low maintenance, multivendor equipment, and low support costs.

This document provides the protocol specifications for the LonTalk protocol layers 1.5-7 (the physical layer protocol is not limited to any particular communications medium, and thus is defined by any number of transceiver designs that can be connected to the Neuron® Chip). See the Neuron Chip Data Book published by Motorola and Toshiba for details on the interface requirements of the communication port. See the LONMARK™ Layers 1-6 Interoperability Guidelines for specifications on standard transceivers. See the LONMARK™ Application Layer Interoperability Guidelines for specifications on how to design interoperable application nodes with the LonTalk Protocol.

1.2 Document Overview

Following the overview in section 2 and the description of addressing in section 3, the document is structured in a uniform fashion. A chapter is dedicated to each protocol layer or sublayer. Each chapter starts with a list of assumptions about the service provided by the underlying layers. For the sake of completeness, a brief service description is then included, followed by a detailed specification of the protocol. Most of the algorithms are specified in structured English, using a Pascal-like notation.

Chapter 11 contains a description of the built-in network management capabilities of the protocol. Chapter 12 describes the essential behavioral characteristics of the protocol. Finally, the syntax of all Protocol Data Units is summarized in Appendix A.

2. TERMINOLOGY AND PROTOCOL OVERVIEW

2.1 Terminology

The primary objective of this document is to provide a concise yet readable specification of the LonTalk protocol. For this reason, the notation and terminology is not as formal as that used in some other protocol specifications.

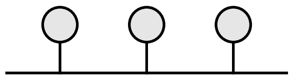
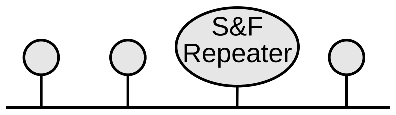
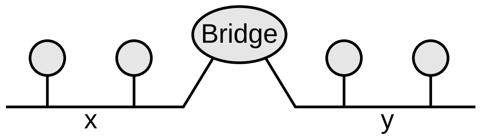
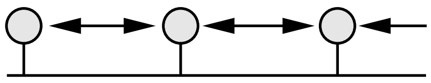
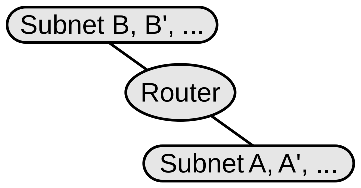
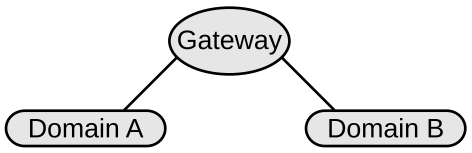
	Simple Channel
	Store & Forward Repeater may repeat on the same channel or may connect two channels; generates duplicates; multiple repeaters may cause packet looping.
	Bridge connects two channels (x and y); forwards all packets from x to y and vice versa, as long as the packets originated on one of the same domain(s) as the bridge
	Subnet a set of nodes accessible through the same layer 2 protocol; a routing abstraction for a channel; LonTalk protocol subnets are limited to ≤127 nodes
	Router routes packets to their respective destinations by selectively forwarding from subnet to subnet; a LonTalk router always connects two (sets of) subnets; LonTalk routers may modify the layer 3 address fields to prevent packets from looping.
	(Application) Gateway interconnects networks at their highest protocol layers (often two different protocols); two LonTalk domains can be connected through an application gateway.

Table 2.1 Basic Terminology

Table 2.1 introduces the basic terminology employed throughout the document. Most of it is commonly used and the terms have the same meaning in both the general and the LonTalk context. However, there are subtle differences. For example, bridges in general do selective forwarding based on the layer 2 destination address. There are no layer 2 addresses in the LonTalk protocol, so LonTalk bridges forward all packets, as long as the Domain address in the packet matches a Domain of which the bridge is a member. Routers, in general, perform network address modification so that two protocols with the same transport layer but different network layers can be connected to form a single logical network. LonTalk routers perform network address modification on packets that might otherwise loop, but typically they only examine the network address fields and selectively forward packets based on the layer 3 address fields.

The LonTalk protocol layering is described using the standard OSI terminology, as shown in figure 2.1.

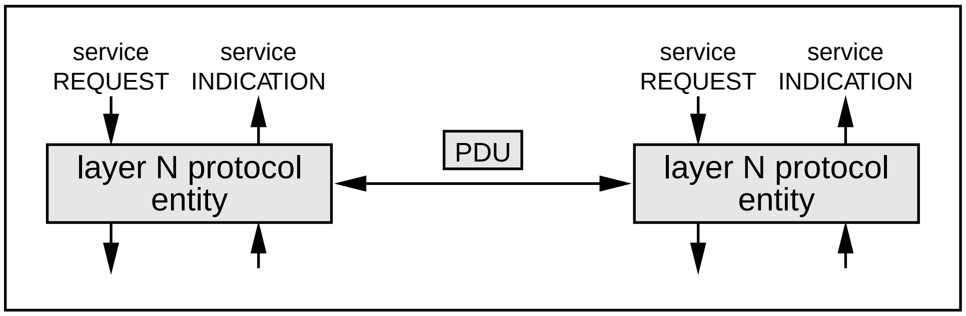


Figure 2.1 Protocol Terminology

The Protocol Data Unit (PDU) abbreviations used throughout this document are:

MPDU	MAC Protocol Data Unit, or frame
LPDU	Link Protocol Data Unit, or frame
NPDU	Network Protocol Data Unit, or packet
TPDU	Transport Protocol Data Unit, or a message/ack
SPDU	Session Protocol Data Unit, or request/response
NMPDU	Network Management Protocol Data Unit
DPDU	Diagnostic Protocol Data Unit
APDU	Application Protocol Data Unit

2.2 Overview of LonTalk Protocol Layering

LonTalk protocol layering consists of the layers shown in table 2.2. At each layer within the table there is a description of the services provided within that layer. Each layer is described below.

Multiple Physical Layer protocols and data encoding methods are used in LonTalk systems. Each encoding scheme is media dependent. For example, differential Manchester encoding is used on twisted pair, both FSK modulation and a modified direct sequence spread spectrum system is used on the power line, FSK modulation is used on RF, etc.

In order to deal with a variety of media in the potential absence of collision detection, the *MAC* (Medium Access Control) sublayer employs a collision avoidance algorithm called Predictive *p*-persistent CSMA (Carrier Sense, Multiple Access). For a number of reasons, including simplicity and compatibility with the multicast protocol, the *Link* layer supports a simple connection-less service. Its functions are limited to framing, frame encoding, and error detection, with no error recovery by re-transmission.

LAYERS 6, 7:	Application & Presentation Layers	
	Application: network variable exchange, application-specific RPC, etc.	Network Management: network management RPC, diagnostics
LAYER 5:	Session Layer request-response service	
LAYER 4:	Transport Layer acknowledged and unacknowledged unicast and multicast	
	Authentication server	
	Transaction Control Sublayer common ordering and duplicate detection	
LAYER 3:	Network Layer connection-less, domain-wide broadcast, no segmentation, loop-free topology, learning routers	
LAYER 2:	Link Layer framing, data encoding, CRC error checking	
	MAC Sublayer predictive *p*-persistent CSMA: collision avoidance; optional priority and collision detection	
LAYER 1:	Physical Layer multiple-media, medium-specific protocols (e.g., spread-spectrum)	

Table 2.2 LonTalk Protocol Layering

The *Network* layer handles packet delivery within a single domain, with no provisions for inter-domain communication. The Network service is connection-less, unacknowledged, and supports neither segmentation nor re-assembly of messages. The routing algorithms employed by the network layer to learn the topology assumes a tree-like network topology; routers with configured tables may operate on topologies with physical loops, as long as the communication paths are logically tree-like. In this configuration, a packet may never appear more than once at the router on the side on which the packet originated. The unicast routing algorithm uses learning for minimal overhead and no additional routing traffic. Use of configured routing tables is supported for both unicast and group addresses, although in many applications a simple flooding of group addressed messages is sufficient.

The heart of the protocol hierarchy is the Transport and Session layers. A common Transaction Control sublayer handles transaction ordering and duplicate detection for both. The *Transport* layer is connection-less and provides reliable message delivery to both single and multiple destinations. Authentication of the message sender’s identity is provided as an optional feature. The authentication server requires only the Transaction Control Sublayer to accomplish its function. Thus transport and session layer messages may be authenticated using all of the LonTalk addressing modes other than broadcast.

The *Session* layer implements a simple Request-Response mechanism for access to remote servers. This mechanism provides a platform upon which application specific remote procedure calls can be built. The LonTalk network management protocol, for example, is dependent upon the Request-Response mechanism in the Session layer -- even though it accesses the protocol via the application layer interface.

The *Presentation* layer and the *Application* layer taken together form the foundation of interoperability for LonTalk nodes. The application layer provides all the usual services for sending and receiving messages, but it also contains the concept of network variables. The presentation layer provides information in the APDU header for how the APDU is to be interpreted for network variable updates. This application independent interpretation of the data allows data to be shared among nodes without prior arrangement. With agreement on which network variables are to be used for sensors, actuators, etc. intelligent components from different manufacturers may work together without prior knowledge of each other's characteristics.

3. NAMING AND ADDRESSING

3.1 Address Types and Formats

LonTalk addresses are *hierarchically* structured. There are three basic address types, with three components per address, as shown below.

(Domain, Subnet, Node)
(Domain, Subnet, Neuron_ID)
(Domain, Group, Member)

The syntax and semantics of the individual address components are described in sections 3.2–3.5. The source and destination addresses are transported within every PDU. For this purpose, the basic addressing formats shown above are further combined into addressing pairs, defined in section 3.6.

Each LonTalk address is a combined layer 3 and layer 4 address; no addressing is employed at layer 2. The domain and subnet address components are used in routing and could be called a network address as a result. The Neuron_ID is a name rather than an address, since it does not change when the node is moved. Thus, address components used in routing (layer 3) and naming (layer 4) are combined in LonTalk addressing.

Address size varies in general while being invariant within each domain. The size of the domain field varies (0,1,3, or 6 bytes); the subnet and group fields are 8 bits wide, allowing for up to 256 groups and 255 subnets per domain (subnet 0 is a reserved value); the size of the node field is 7 bits (an all zeros field not being used); the size of the group member field is 6 bits with a range of 0..63; and the Neuron_ID field is 48 bits wide. This yields 2^8-1 *2^7-1 or ~2^{15} nodes per LonTalk domain. Multiple domains can be used in a single system to increase the number of group addresses, nodes, etc.

3.2 Domains

The LonTalk domain identifier is the first component of the addressing hierarchy. This identifier *uniquely* identifies a LonTalk domain within some context. The size of the domain identifier depends on this context; 48-bit domain identifiers are provided for world-wide uniqueness. When or where this context is otherwise limited (e.g., physically, say within a single building), domain identifiers of smaller size may be used.

A domain is a *virtual network*, with all communication being limited to a single domain. One reason is that the source and destination addresses of an NPDU/TPDU must belong to the same domain (see 3.3 below). Another reason is that the LonTalk protocol stack does not support the equivalent of an internet protocol; where inter-domain communication is needed, it must be facilitated by application level gateways.

A domain is also the unit of *management* and administration. In particular, group and subnet addresses are assigned by the administrator responsible for the domain, and they have meaning only in the context of that domain.

3.3 Subnets and Nodes

The LonTalk subnet identifier is the second component of the addressing hierarchy. Its value uniquely identifies a subnet within a domain; the subnet address of 0 signifies that the subnet is undefined or unknown.

A subnet is a domain subset containing 0-127 nodes with the property that there is *no routing* within a subnet. Subnets are a routing abstraction for a channel; that is, subnets are logical channels, and need not correspond to the physical channel topology. One or more subnets may be mapped onto a single channel (or onto two or more channels connected via store and forward repeaters or bridges).

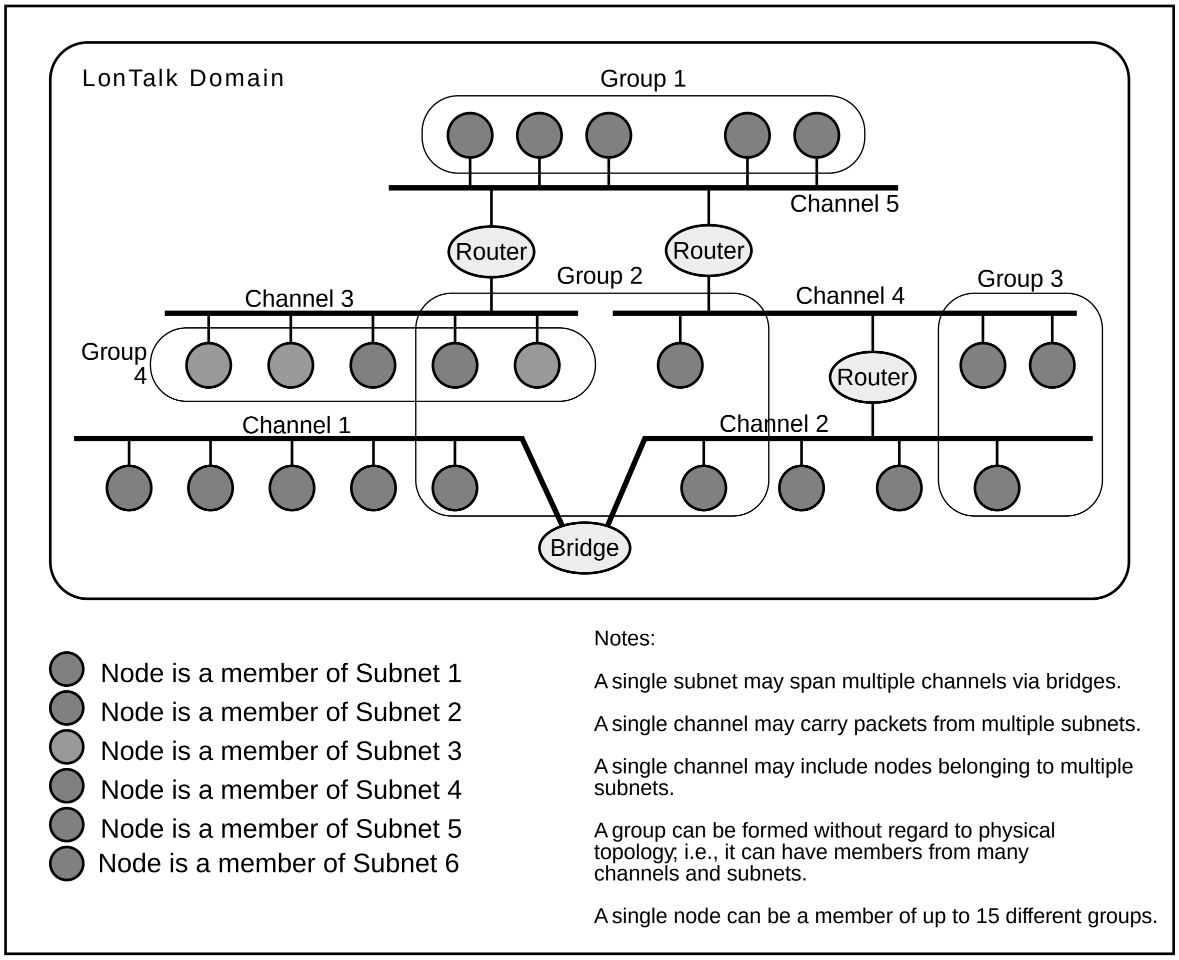
> Note: The term subnet is normally used in communications when referring to a subset of a network such that there is no routing within that subset. LonTalk subnets have the additional property that they contain at most 127 nodes. As a result, two or more LonTalk subnets may be contained in what would normally be called a subnet. In the LonTalk protocol, this is referred to as a channel. That is, a channel is a physical unit of bandwidth and a subnet is a logical construct used for routing in the LonTalk protocol. Note also that a channel is a physical unit of bandwidth, so a channel is composed of one or more network segments. When a channel consists of multiple segments, these segments shall be connected by physical layer repeaters so that the bandwidth of the channel remains constant regardless of the number of segments that it contains.

The node identifier identifies a (logical) node within a subnet. A physical node may belong to more than one subnet; when it does, it is assigned one (logical) node number for each subnet to which it belongs. A physical node may belong to at most two subnets, and those subnets must be in different domains.

3.4 Groups

The LonTalk group identifier is the second component of the addressing hierarchy. It uniquely identifies a set of nodes within a domain. Within this set, individual members are identified by the third addressing component (i.e., the member number).

Group addressing facilitates one-to-many communication. Groups are intended to support functional addressing, and, in particular, the concept of network variables used in the LonTalk application protocol.


LonTalk Domain
Group 1
Channel 5
Router
Router
Group 2
Channel 3
Channel 4
Group 3
Group 4
Router
Channel 1
Channel 2
Bridge
● Node is a member of Subnet 1
● Node is a member of Subnet 2
● Node is a member of Subnet 3
● Node is a member of Subnet 4
● Node is a member of Subnet 5
● Node is a member of Subnet 6

Notes:

A single subnet may span multiple channels via bridges.

A single channel may carry packets from multiple subnets.

A single channel may include nodes belonging to multiple subnets.

A group can be formed without regard to physical topology; i.e., it can have members from many channels and subnets.

A single node can be a member of up to 15 different groups.

Figure 3.1 LonTalk Physical Topology And Logical Addressing (Single Domain)

3.5 Neuron_ID

Each LonTalk node is assigned a unique 48-bit identifier called Neuron_ID. The value of this identifier does not change from the time of manufacture. A Neuron_ID is a name rather than an address. When the Neuron_ID is used as an address, it may only be used as a destination address, and it must be combined with the domain and the source subnet addressing components (see section 3.6).

3.6 NPDU Addressing

For NPDU addressing, the basic addressing formats introduced in section 3.1 are combined into (Source, Destination) address pairs as defined in table 3.1 and figure 3.2. An NPDU carries both the source and the destination address in one of the five possible addressing formats.

Type	Logical Address Format	Used with TPDU/SPDU Type	
#0:	(Domain, sourceSub-Node,destSubnet)	broadcast:	Domain wide or by individual subnet
#1:	(Domain, sourceSub-Node, destGroup)	multicast:	Message or Reminder
#2a:	(Domain, sourceSub-Node, destSub-Node)	unicast:	Message, Reminder, or Acknowledgment
#2b:	(Domain, sourceSub-Node, destSub-Node, Group, Mem)	multicast:	Acknowledgment
#3:	(Domain, sourceSub-Node, destSub, Neuron_ID)	unicast:	Message or Reminder

Table 3.1 NPDU/TPDU/SPDU Addressing—Logical Address Formats

In each address format, a source subnet of 0 means that the node does not know its subnet number. This is the condition of a node prior to configuration with network management messages. In figure 3.2, below, note that each of the address formats contains a 7 bit source node field. The eighth bit in the source node field byte is the selector field. It is used to supply sub-variants of addressing formats. Address format #2 is the only address format using this capability. In figure 3.2 the numbers above each of the fields represent their bit widths. The first byte of the NPDU contains the NPDU header, which contains the protocol version, the format of the enclosed PDU, the addressing format, and the length of the domain field. The next part of the NPDU header specifies one of the four primary address formats. The final part of the NPDU header contains the length of the domain. The address field immediately follows the NPDU header.

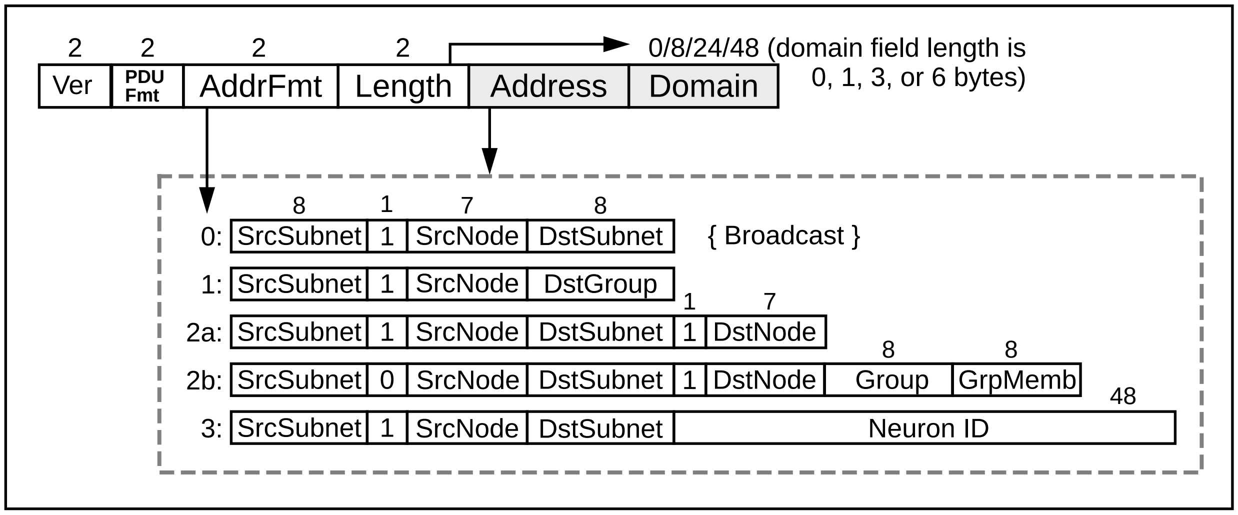

2
2
2
2
0/8/24/48 (domain field length is
0, 1, 3, or 6 bytes)
Ver
PDU Fmt
AddrFmt
Length
Address
Domain
8
1
7
8
0: SrcSubnet 1 SrcNode DstSubnet
{ Broadcast }
1: SrcSubnet 1 SrcNode DstGroup
1
7
2a: SrcSubnet 1 SrcNode DstSubnet 1 DstNode
8
8
2b: SrcSubnet 0 SrcNode DstSubnet 1 DstNode Group GrpMemb
48
3: SrcSubnet 1 SrcNode DstSubnet Neuron ID

Figure 3.2 NPDU/TPDU/SPDU Addressing—Physical Address Formats

The first part of the address field is the source subnet field. This field is used for routers to both learn the topology and to prevent packet looping. The combination of the source subnet field and the source node field is used for acknowledgments, authentication challenges, replies to authentication challenges, responses (when the request/response mechanism is used), and rejection of packets that are received from the same node to which they were sent. The domain field of the length specified in the NPDU header follows the source and destination address pair. Finally, the PDU of the format specified in the NPDU header.

Address format #0 facilitates domain-wide broadcast. The NPDU contains the (subnet, node) address of the source node and the destination subnet. A destination subnet of 0 implies all subnets, while a destination subnet ranging from 1 to 255 shall broadcast only to the nodes on the named destination subnet.

Address format #1 supports multicast message delivery. It uses a source address of the form (subnet, node), while the destination address is (group), implying message delivery to the entire group.

Address format #2 has two variants. With variant #2a, both the source and the destination address are of the form (subnet, node). Variant 2a is used for unicast message delivery and acknowledgments. Variant #2b supports group acknowledgments. Its source component is (subnet, node). The source and destination fields are identical to format #2a to facilitate routing. Then, appended to the source and destination fields, are the group and member numbers of the acknowledging node.

Address format #3 supports addressing by Neuron_ID. Since the primary intention of this addressing mode is to facilitate address assignment, Neuron_ID can only be used as destination address. The ID may be obtained from the node via a special network management message described in the Network Management chapter, and can also be had by actuating the service pin on the node (also described in the Network Management chapter). In cases where the destination subnet is unknown, a destination subnet of zero is used to propagate the packet throughout the network.

3.7 Address Assignment

The 48-bit Neuron_ID is unique worldwide and is set at the time of node manufacture. An unconfigured LonTalk node has no assigned address apart from its 48-bit Neuron_ID. These unconfigured nodes receive packets from all domains, looking for and responding to any packet containing the node’s 48-bit Neuron_ID.

A node may be assigned multiple addresses. In addition to its Neuron_ID, a node is usually assigned one address of type (domain, subnet, node) and zero to fifteen addresses of type (domain, group, member). A node is typically a member of only one domain; a LonTalk node may be a member of up to two domains, however. Nodes that belong to multiple domains have two (domain, subnet, node) addresses—one for each domain.

4 MAC SUBLAYER

4.1 Service Provided

The LonTalk Media Access Control (MAC) sublayer facilitates media access with optional priority and optional collision detection/collision resolution. It uses a protocol called Predictive *p*-persistent CSMA (Carrier Sense, Multiple Access), which has some resemblance to the *p*-persistent CSMA protocol family.

Predictive *p*-persistent CSMA is a *collision avoidance* technique that randomizes channel access using knowledge of the expected channel load. A node wishing to transmit always accesses the channel with a random delay in the range (0..w). To avoid throughput degradation under high load, the size of the randomizing window is a function of channel backlog BL: w = BL*W_{base}, where W_{base} is the base window size. Provided that the real backlog does not exceed the estimated backlog (see 4.6 below), the average *collision rate* does not exceed *1 in 2*W_{base}.

4.2 Interface to the Link Layer

The MAC sublayer is closely coupled to the LonTalk Link layer, described in chapter 5. With the MAC sublayer being responsible for media access, the Link layer deals with all the other layer 2 issues, including framing and error detection. For explanatory purposes, the interface between the two layers is described in the form shown in figure 4.1.

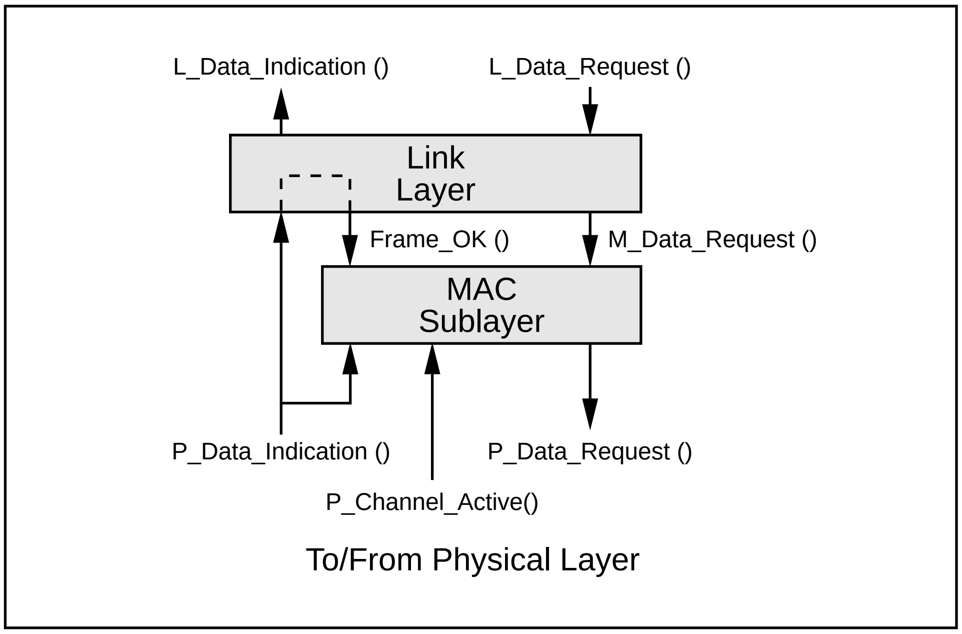


Figure 4.1 Interface Between the MAC and Link Layers

Although the service interface primitives are defined using a syntax similar to programming language procedure calls, no implementation technique is implied. Frame reception is handled entirely by the Link layer, which notifies the MAC sublayer about the backlog increment via the Frame_OK () primitive.

The interface between the Link and the MAC layers is facilitated by the following service interface primitives:

M_Data_Request (Priority, delta_BL, ALT_Path, LPDU)

> This primitive is used by the Link layer to pass an outbound LPDU / MPDU to the MAC sublayer. Priority defines the priority with which the frame is to be transmitted; delta_BL is the backlog increment expected as a result of delivering this MPDU. ALT_Path is a binary flag indicating whether the LPDU is to be transmitted on the primary or alternate channel, baud rate, etc.

Frame_OK (delta_BL)

> On receiving a frame and verifying that its checksum is correct, the Link layer invokes this primitive to notify the MAC sublayer about the backlog increment associated with the frame just received.

4.3 Interface to the Physical Layer

The Physical layer handles the actual transmission and reception of binary data. Every LonTalk node communicates to the physical layer in one of two modes: direct mode and special purpose mode. In direct mode, the Link layer uses differential Manchester encoding. In special purpose mode, data are transferred serially in and out of the node without encoding. In both modes a 16-bit CRC is generated on transmission and checked on reception. These two modes form an abstraction for all the physical layers supported by the LonTalk protocol.

Multiple Physical layer protocols are employed in the LonTalk system. These protocols may employ media-specific features as long as the following three requirements are met:

- Physical idle state, used to represent the idle channel condition, is a low power consumption state;

- Bit error rate is equal to, or better than, than 1 in 10^{-4}, as presented to layer 2;

- For compatibility with the higher layers, all physical protocols must support the service interface defined below.

The service interface to be supported by all physical layers is (see figure 4.1):

P_Data_Indication (Frame)

Physical layer provides this indication to the higher layers once per incoming LPDU/MPDU.

P_Data_Request (Frame, Status)

The MAC sublayer uses this primitive to pass the Frame, the encoded LPDU/MPDU, to the physical layer for immediate transmission. The bit transmission order is defined in Appendix A. The physical layer returns Status as to whether the frame was transmitted. Status has three possible values: success—indicating the frame was transmitted, request_denied—indicating that activity was detected on the line prior to transmission, and collision—indicating that transmission began, but a collision was detected. Whether or not the transmission is aborted depends on whether the interface to the physical layer is direct mode or special purpose mode as well as when the collision is detected (see below).

P_Channel_Active()

The physical layer uses this primitive to pass the status of the channel to the MAC sublayer. This is an indication of activity, not necessarily of valid data.

4.4 Collision Detection Notification

If collision detection is provided by the physical layer, the action taken upon notification of a collision depends upon when the collision is detected and what mode (direct or special purpose) is being used for the communications port.

In special purpose mode, transmission of an outgoing packet is aborted by the physical layer immediately upon detection of a collision. The MAC sublayer is then notified that the collision occurred. Transceivers that can perform arbitration based upon a pattern at the beginning of a packet take advantage of this feature to implement collision resolution. Such transceivers must have a sufficiently long preamble after the arbitration pattern so that other transmitting nodes that have lost the arbitration for the channel can notice channel activity and receive the incoming packet from the station that just won the arbitration.

In direct mode transmission of an outgoing packet, the MAC sublayer checks for a collision indication at the end of transmission. Optionally, the MAC sublayer may be configured to check for a collision approximately half way through the transmission of the packet preamble. If this option is chosen, and a collision is detected by the physical layer during the preamble, the transmission of the packet is aborted.

In both special purpose mode and in direct mode the MAC sublayer attempts to retransmit the packet upon notification of a collision using the MAC protocol described in the remainder of section 4.

4.5 MPDU Format

The combined MPDU/LPDU format is shown in figure 4.2. In direct mode, the ByteSync field, which indicates the beginning of a frame, is 1 bit wide and has a value of ‘0’. ByteSync is preceded by BitSync in direct mode. BitSync is a string of ‘1’ bits, the length of which is a channel configuration parameter. BitSync must be long enough for all nodes on the channel to see activity and synchronize on the incoming bit stream. Also, in direct mode, the frame is terminated by the transmitter holding the idle line state for at least 2.5 bit times plus the propagation delay of the channel. Receivers detect end of frame by seeing the idle line state for at least 1.25 bit times.

When the interface to the physical layer is via special purpose mode, the BitSync, ByteSync and end of frame are determined by the external transceiver.

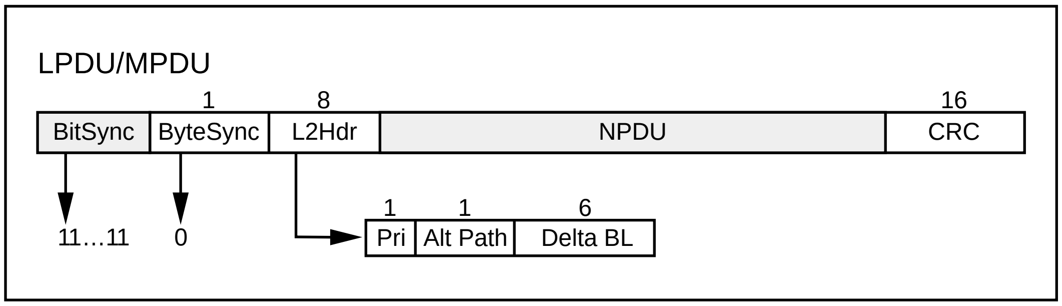


Figure 4.2 LonTalk MPDU/LPDU Format

The MAC layer uses the L2Hdr field, which has the following syntax and semantics:

Pri	1-bit field specifying the priority of this MPDU: 0 = Normal, 1 = High
Alt_Path	a 1-bit field specifying the channel to use. This is a provision for transceivers that have the ability to transmit on two different channels and receive on either one without prior configuration. The transport layer sets this bit for the last two retries, or the MAC sublayer can be configured to always transmit on the alternate path.
Delta_BL	a 6-bit field; specifies channel backlog increment to be generated as a result of delivering this MPDU

4.6 Predictive *p*-persistent CSMA — Overview Description

Like CSMA, Predictive *p*-persistent CSMA senses the medium before transmitting. A node attempting to transmit monitors the state of the channel (see figure 4.3), and when it detects no transmission during the Beta1 period, it asserts the channel is idle.

Next, the node generates a random delay T (transmit) from the interval (0..BL*w_{base}), where w_{base} is the size of the basic randomizing window and BL is an estimate of the current channel backlog. T (transmit) is defined as an integer number of randomizing slots of duration Beta2 (see sections 4.7 and 4.8). If the channel is idle when the delay expires, the node transmits; otherwise, the node receives the incoming packet, and then repeats the MAC algorithm. In figure 4.3 below, D_{mean} is the average delay between packets, and, since the random delay T is uniformly distributed, D_{mean} is given as $W_{base}/2$ for small values of BL. In theory, when BL is large the algorithm tends to overestimate the backlog, which can cause D_{mean} to increase until the backlog decays although in practice this affect is mitigated by processing delays in the nodes. This is because the backlog is incremented when a packet is received with a non-zero backlog increment. The backlog then decays *while* the nodes are formulating their acknowledgments so that empirically it is found that the overestimation of the backlog is brief, and channel utilization remains near saturation.

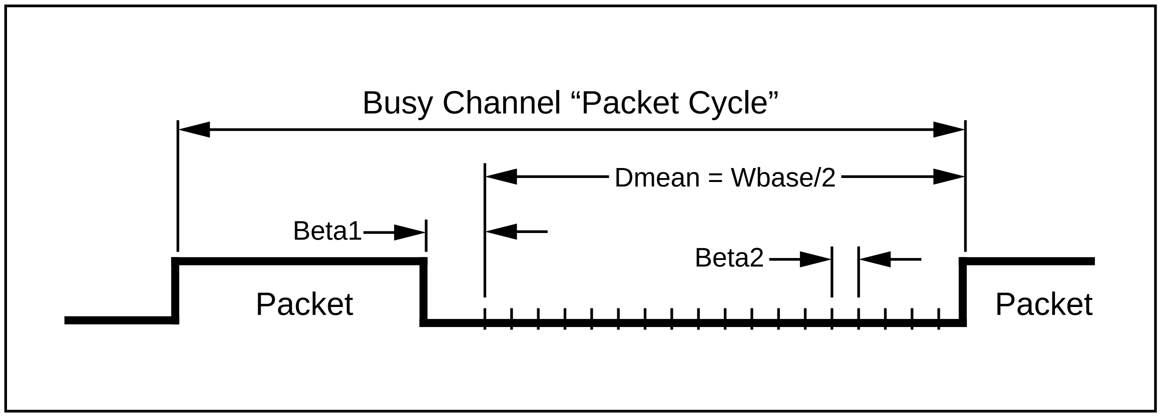


Figure 4.3 Predictive *p*-persistent CSMA Concepts and Parameters
Beta1 = Idle Slot, Beta2 = Randomizing Slot

By adjusting the size of the randomizing window, W_{base}, as a function of the predicted load, the algorithm keeps the collision rate constant and independent of the load. Provided that the estimated backlog is greater than or equal to the real backlog, the following holds:

$$\text{Collision Rate} = \text{Error Pkt Cycles} \;/\; \text{Error Free Pkt Cycles} \leq 1 \;/\; 2W_{base}$$

A base window size of 16 maximizes the L4/L5 transaction throughput. This implies that there are an average of 8 randomizing slots of width Beta2 and one slot of width Beta1 between each packet. Also, the width of the Beta2 period is crucial to efficient utilization of the channel.

4.7 Idle Channel Detection

The idle channel condition is asserted whenever the following two conditions are met:

1) The current channel state reported by the physical layer via the P_Data_Indication () primitive is <u>low</u>; and

2) <u>No transition</u> has been detected during the last period of Beta1.

The length of the Beta1 period is defined by the following constraint:

$$\textbf{Beta1} > \textbf{1 bit time} + (2 * \textbf{Tau}_p + \textbf{Tau}_m)$$

The first term assumes a data encoding method which guarantees a transition and/or carrier during every bit time. In special purpose mode, when encoding methods are used which do not meet this constraint, then the first term must be adjusted to be the longest time that the channel may appear idle without being idle, i.e. the longest run in legal data transmission without a transition and/or carrier asserted on the medium. The second term takes care of propagation and turnaround delays, which are:

Tau_p	is the physical propagation delay defined by the media length;
Tau_m	is the detection and turn-around delay within the MAC sublayer; this is the period from the time the idle channel condition is detected, to the point when the first output transition appears on the output. On media where there is a carrier, this time must include the time between turning on the carrier, and it being asserted as a valid carrier on the medium.

4.8 Randomizing

At the beginning of the randomizing period, a node wishing to transmit generates a random delay T (transmit) from the interval (0..BL*w_{base}). The node then waits for this period, while continuing to monitor channel status; if the channel is still idle when the delay expires, the node transmits.

The transmit delay T (transmit) is specified as an integer number of randomizing slots of duration Beta2; the length of the randomizing slot must meet the following constraint:

$$\textbf{Beta2} > 2 * \textbf{Tau}_p + \textbf{Tau}_m$$

Parameters Tau_p and Tau_m are defined in section 4.7.

4.9 Backlog Estimation

The predictability of the MAC algorithm is based on backlog estimation. Each node maintains an estimate of the current channel backlog BL, which is incremented as a result of sending or receiving an MPDU and decrements periodically—once every packet cycle. The increment to the backlog is encoded into the link layer header, and represents the number of messages that the packet shall generate upon reception. The backlog also decrements if $BL * w_{base}$ randomizing slots go by without channel activity.

The backlog always has a value ≥ 1. The algorithm post-increments rather than pre-increments the backlog by the amount associated with the MPDU being transmitted, because the number of expected responses is of no importance until after transmitting the MPDU.

4.10 Optional Priority

On a channel by channel basis, the LonTalk protocol supports optional priority. Priority slots, if any, follow immediately after the Beta1 period which follows the transmission of a packet. The number of priority slots per channel ranges from 0 to 127. Priority slots are typically not contended for, but rather are uniquely assigned to nodes on the channel. Nodes that have been assigned a priority slot do not have to use it with every message; the node decides on a message by message basis whether or not to use the assigned priority slot. This determination is made by examining the priority bit within the LPDU header (figure 4.2).

It is possible to assign all the nodes on the channel the same priority slot. An example of an architecture where this makes sense would be one where there is a background of peer-to-peer activity, but a single master which cycles around doing something to each node (such as network management, polling, etc.). By giving each node the same slot, and having it used only for this purpose, these transactions (from the single master to multiple slaves) would tend to be completed ahead of the background traffic.

An application may decide that a message is high priority and attempt to send it as such. If the node does not have a priority slot assigned to it, the message shall go out in the usual way, except that the priority bit in the layer 2 header shall be set. If, subsequently, the packet passes through a router that has a priority slot on its destination channel, the packet shall be sent using the priority of the router.

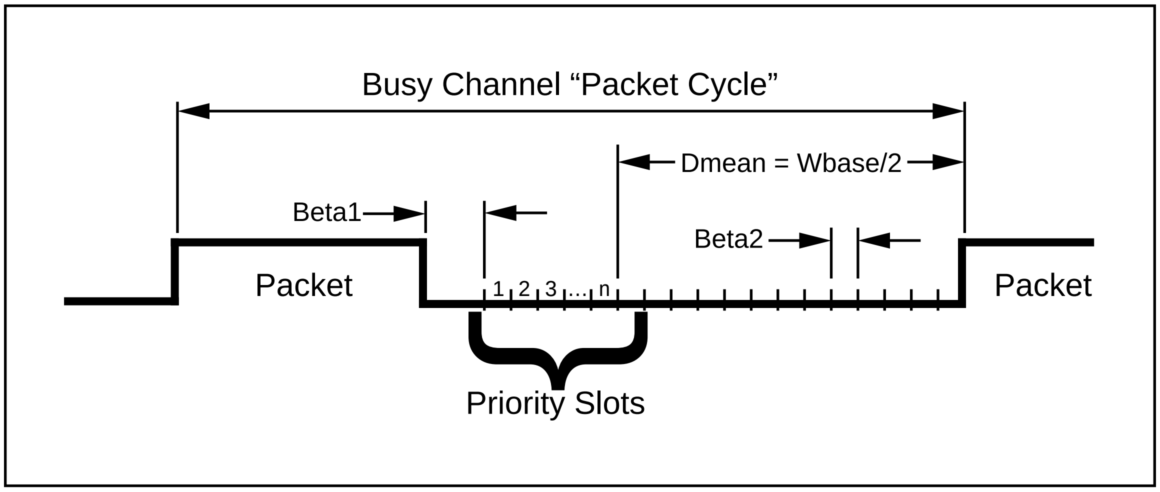


Figure 4.4 Allocation Of Priority Slots Within the Busy Channel Packet Cycle

The LonTalk protocol provides no synchronization among the nodes. Therefore, if the channel has been idle for longer than the randomizing period (Beta1 + number of priority slots + Dmean above), access to the link is random without regard to priority. Once the link returns to the busy state, access to the link shall be in priority order.

If a priority message is sent using either the request/response protocol or reliable message passing, then the responding node shall attempt to send a priority acknowledgment/response by setting the priority bit in the layer 2 header. If a high priority message is generated within a node, it is sent prior to any queued packets of normal priority. Multiple high priority packets are sent in FIFO order. If the application attempts to send a high priority message while its node is sending a packet, the packet in progress completes first.

If a node has multiple priority messages queued within it, it shall not send the priority messages in consecutive packet cycles, as this would effectively tie up the channel. In the case where a node has a priority packet to send, and it has sent a packet in the previous packet cycle, the node does not use its priority slot in the current cycle. Instead it attempts to access the medium using the non-priority MAC algorithm. If the node is not successful in the current packet cycle, it may use its priority slot in the subsequent packet cycle.

4.11 Optional Collision Detection

The MAC sublayer obeys some special rules when collision detection is enabled.

1. If a collision is detected on two successive attempts to transmit a priority packet in the node's priority slot, the next attempt to transmit the priority packet shall not use the configured priority slot,

but rather shall be in a slot picked according to the non-priority MAC algorithm.

2. Whenever a collision is detected by a transmitting node, that transmitting node increments its estimate of the channel backlog by 1.

3. Whenever a collision is detected on 255 successive attempts to transmit a packet, the packet is discarded.

4.12 The Predictive CSMA Algorithm

Algorithm 4.12:

Channel-wide constants:

Beta1	idle slot length (see section 4.6)
Beta2	length of the randomizing slot (see 4.7)
w_{base}	basic randomizing window (for BL=1), $w_{base} = 16$
BL_{max}	maximum value of channel backlog, $BL_{max} \leq$ number of nodes on the channel
Pslots	number of priority slots allocated on the channel (range 0–127)
NodePslot	priority slot assigned to this node (range 0–127 with 0 being no priority slot assigned to the node)

State variables and timers:

BL	an estimate of the current channel backlog
C_{state}	channel state, one of (busy, busy1, ..., idle) defined by algorithm 4.12
PktToXmit	Boolean
T (Cycle)	the “packet cycle” timer, expires every (AvgPktSize + $w_{base}/2$)
T(transmit)	transmit/randomizing timer (see section 4.7)

Events driving the algorithm:

M_Data_Request (priority, delta_BL, Alt_path, Frame)	
Frame_OK (delta_BL)	{indication from the Link layer, see 4.2}
Channel_Idle ()	{ indication from algorithm 4.12 }
Timer Expiration	{of T (transmit) or T (Cycle) }

```
begin { algorithm 4.12 }
Case Event Of

    Frame_OK  (delta_BL):
        Begin
        BL := BL + Delta_BL;
        if BL > $BL_{max}$ then BL := $BL_{max}$;
        end;

    M_Data_Request (priority, delta_BL, Alt_path, Frame):
        begin
        PktToXmit := True;
        if $C_{state}$= Idle then Begin
             /*Depending on priority, generate a random delay in the following  range: */
                 Normal:      transmit := rand(0 .. $w_{base}$ * BL) + Pslots;
                 High:   If NodePslot <> 0
                                 transmit := NodePslot;
                         else
                                 transmit := rand(0 .. $w_{base}$ * BL) + Pslots;
                         /* Backlog value BL used to generate the above random number          */
                         /* should be either the value at time t, or at (t-1), where t is current time */
            Start timer T(transmit);
        end;

    Channel_Idle  ():
        If (PktToXmit = True) and (T(transmit) not running) then
            Start timer T (transmit);
                end;

    T (Cycle) Expiration:
        If BL > 1 Then Begin
            BL := BL - 1;
            Restart T (Cycle);
            Restart T ($w_{base}$ );
        end;

    T ($w_{base}$ ) Expiration:
        If BL > 1 Then Begin
            BL := BL - 1;
            Restart T ($w_{base}$ );
        end;
    T (transmit) Expiration:
        If $C_{state}$ = Idle Then Begin
            { process M_Data_Request () now }
            Transmit Frame;
            if collision_detected then Begin
                BL++;
                if BL > $BL_{max}$ then BL := $BL_{max}$;
                Event := M_Data_Request(priority,delta_BL, Alt_path, Frame);
                end;
            else Begin
                BL := BL + delta_BL;
                if BL > $BL_{max}$ then BL := $BL_{max}$;
                PktToXmit := False;
                end;
            end;
    end case;
end { algorithm 4.12 } ;
```

4.13 Timing

Communication speeds of the Neuron Chip are derived from its input clock. There are five possible input clock values and 8 different communications port values derived from the supplied input clock value. Possible values of the input clock are 10 MHz, 5 MHz, 2.5 MHz, 1.25 MHz and 625 kHz. These input clock values allow the communications port of the Neuron Chip to run at speeds from 1.25 Mbps all the way down to 610 bps. The communications port can be configured to operate in direct mode where the Neuron Chip controls the data rate, preamble length, and data encoding. Alternatively, the communications port can be configured to operate in special purpose mode where the transceiver controls the data rate, preamble length, and data encoding. In all cases, the Neuron Chip controls the beta 1 and the beta 2 times.

Preamble length, beta 1 and beta 2 times must be configurable to support a wide variety of physical communication channels and to allow nodes running with different input clocks to communicate on the same physical channel. The formulae for beta 1 and beta 2 which yield all possible discrete values for beta 1 and beta 2 are shown below. In each formula CT is the cycle time of the Neuron Chip and v, a tuning parameter, has the range of 0 to 255 inclusive, and PAD is a time delay to compensate for other Neuron Chips which are on the same physical channel, but have slower input clocks.

beta 2 = CT * (40 + 20 * v)

beta 1 = CT * (583 + beta 2 + PAD) { direct mode communications port }

beta 1 = CT * (577 + beta 2 + PAD) { special purpose mode communications port }

$$\mathbf{PAD = 41 * v \quad \{for\ v < 128\}}$$
$$\mathbf{PAD = 145 * (v-128) \quad \{for\ 128 \le v < 256\}}$$

Preamble length is either controlled by the Neuron Chip or by the transceiver as stated above. When the preamble is controlled by the Neuron Chip, the formula for all possible values of preamble length is shown below. For this formula the variable v has a range of from 0 to 253 inclusive. CT is as defined above.

$$\mathbf{preamble = CT * (219 + 32 * v)}$$

When the transceiver controls the preamble length, the minimum preamble length allowed is 181 μseconds. This is the value at a 10 MHz input clock. This value scales with the input clock. The maximum preamble length in special purpose mode is totally under the control of the transceiver.

5. LINK LAYER

5.1 Assumptions

The Link layer assumes that CRC errors due to both collisions and transmission errors occur with some probability P_e, and that P_e is small enough so that Link level error recovery is not needed.

The above assumption means that successful end-to-end communication is only possible when the sum of error probabilities along the communication path is less than one, i.e.

$$\mathbf{SUM\ (P_e) << 1}$$

In networks where P_e is constant, the maximum communication distance D (network or group diameter) must be:

$$\mathbf{D << 1 / P_e}$$

5.2 Service Provided

The LonTalk Link layer provides subnet-wide, ordered, unacknowledged LPDU delivery with error detection but no error recovery. A corrupted frame is discarded as soon as its CRC check fails.

In the direct mode interface to the Physical layer, frame encoding is done using differential Manchester encoding. When using the special purpose mode interface to the Physical layer, frame encoding is accomplished with an external transceiver using an encoding method appropriate to the medium. When using any of the LonTalk protocol supported media, the link layer must see channel busy and detect an idle line in the same manner as in direct mode, regardless of how these patterns are physically transmitted on the medium.

5.3 LPDU Format

Format of the combined MPDU/LPDU was shown in figure 4.2. In direct mode, the ByteSync field, which indicates the beginning of a frame, is 1 bit wide and has a value of ‘0’. Prior to transmission of ByteSync, a preamble called BitSync is transmitted. In direct mode, BitSync is some number of ‘1’ bits long. The length of the BitSync period is determined by the channel configuration. The frame is terminated by the idle state, which in direct mode, is at least 2.5 bit times long, and is simply the absence of transitions. When using the communication port in special purpose mode, the content of the preamble is under control of the transceiver, as is the end of frame indication.

The CRC is computed over the entire NPDU including the L2Hdr field. The CRC is generated using the polynomial $X^{16} + X^{12} + X^5 + 1$ (the CCITT CRC-16 standard).

5.4 The Transmit Algorithm

Algorithm 5.4:

```
Input:
    L_Data_Request ()       from the Network layer

Output:
    M_Data_Request()        service request to the MAC layer

L_Data_Request (Prio, delta_BL, Alt_path, NPDU):
    begin
        Create LPDU and compute CRC;
        If Direct Mode
            Encode LPDU using differential Manchester encoding and add preamble;
        Make the M_Data_Request (Prio, delta_BL, Alt_path, LPDU)) to the MAC sublayer;
    end; {algorithm 5.4 }
```

5.5 The Receive Algorithm

A valid LonTalk frame starts with the channel active state, and terminates with the channel idle state. Upon reception, valid frames are processed as defined below; invalid frames are discarded.

Algorithm 5.5:

```
        begin
        If Direct Mode
            {Decode frame—from differential Manchester to binary;}
        Compare the computed and the enclosed CRCs;
        If correct then begin
            Provide Frame_OK () indication to the MAC layer;
            Provide L_Data_Indication to layer 3;
            end;
        else begin
            Record CRC error;
            end;
        end; { algorithm 5.5}
```

5.6 Differential Manchester Encoding

When communicating via direct mode, the LonTalk protocol uses differential Manchester encoding. This encoding method has the benefits of zero DC offset, polarity insensitivity, and simple bit synchronization between the transmitter and the receiver(s). In this encoding method, there is a minimum of one transition per bit time at the beginning of the bit time. If there is a second transition within the bit time, it occurs in the middle of the bit. By convention, a single transition per bit time is a ‘1’ and two evenly spaced transitions per bit time is a ‘0.’

6. NETWORK LAYER

6.1 Assumptions

The LonTalk protocol supports a variety of topologies in order that the requirements from many application areas can be met. Within a single channel, the topology can be a bus, a ring, a star, or “free” (see Figure 6.1). Free topology is defined as a total wire specification with no other rules, and a single termination placed anywhere on the network. Thus, the set of all free topologies includes a ring, a star, a bus and virtually any other combination of these constructs.

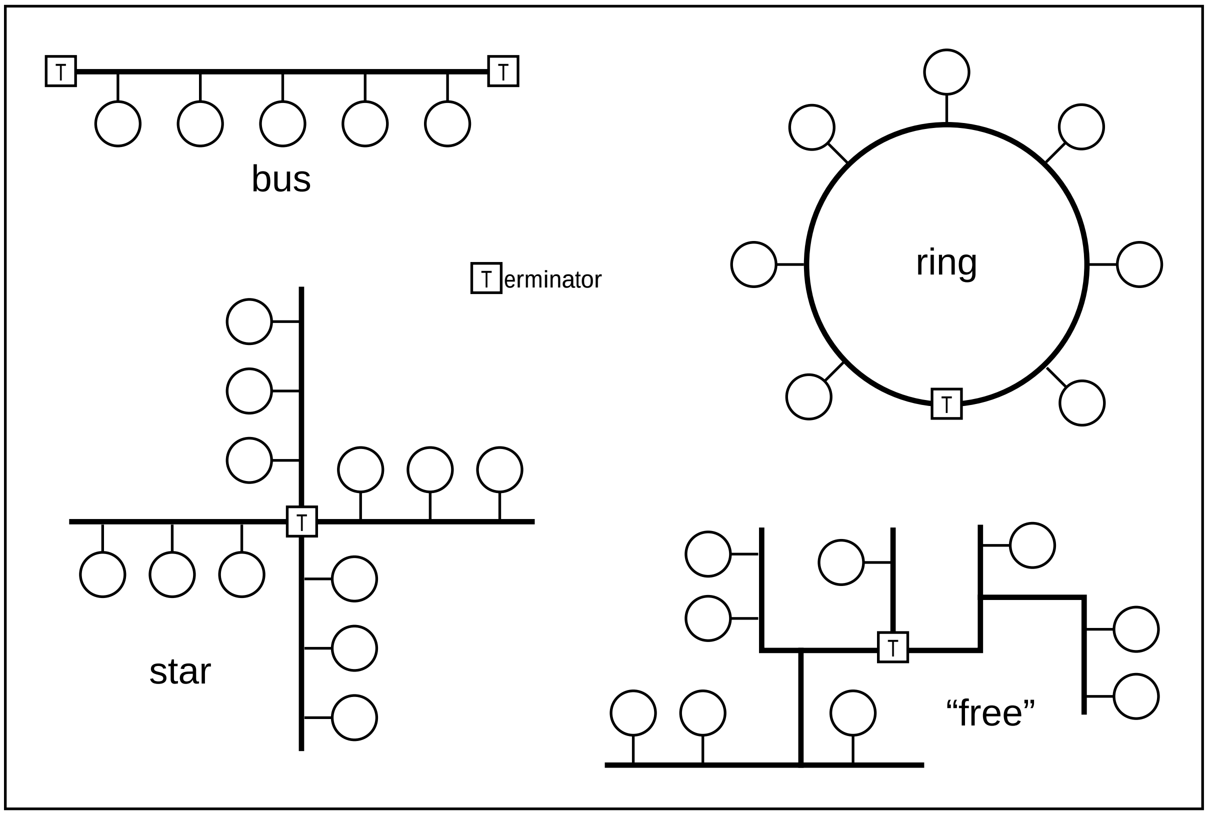


Figure 6.1 Single Channel Topologies

The LonTalk protocol supports physical layer repeaters as well as store and forward repeaters to repeat packets from one channel to another. The protocol also supports bridges to repeat all packets on the bridge’s domain(s) from one channel to another. Additionally, both learning and configured routers are supported to segment traffic and thus increase total system performance.

In networks where there is a possibility of more than one path from one node to another, there is a danger of packets looping indefinitely. In these networks, configured routers must be used to impose a tree structured topology on top of the physical looping topology.

If there is a desire to use repeaters, bridges, and learning routers, then control of the topology must be maintained so that no loops exist. To avoid routing loops within a domain, domain topology must be tree-structured as shown in figure 6.2. Store and forward repeaters may only be used if they connect two different channels together; store and forward repeaters that repeat on the same channel are not supported. This restriction results in an order preserving network.

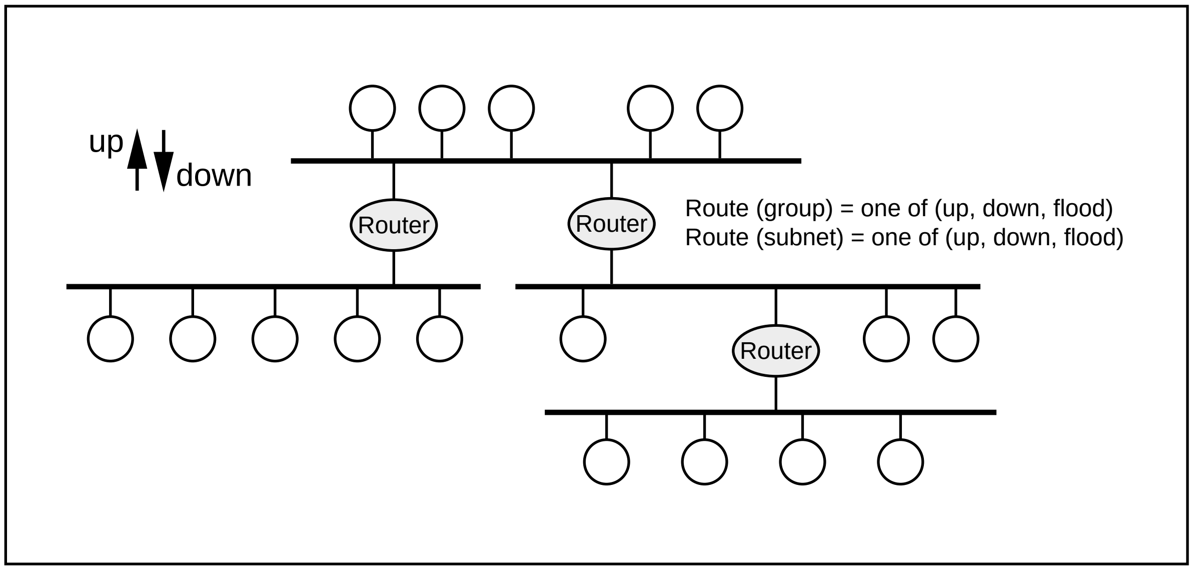

up
down
Router
Router
Route (group) = one of (up, down, flood)
Route (subnet) = one of (up, down, flood)
Router

Figure 6.2 Typical Tree-Like Domain Topology

6.2 Service Provided

The LonTalk Network layer provides a connection-less network service facilitating domain wide packet delivery with the following attributes:

- *Unacknowledged Unicast, Multicast, and Broadcast*. Depending on its destination address, the packet submitted is delivered to one node, multiple nodes, or all nodes within the domain (or optionally all nodes on a specified subnet within the domain). This delivery occurs with some probability $p \leq 1$;

- *Lossiness*. The network layer supports no re-transmissions or acknowledgments. The probability of delivery is inversely proportional to the number of channels traversed;

- *Order Preserving.* Loop-free topology (accomplished logically with configured routers or physically via topological control) coupled with the absence of single channel store and forward repeaters provides natural ordering;

- *No Segmentation.* No message segmentation and/or message reassembly are performed anywhere within the Network layer.

When learning routers are used, the routers discover the topology by examining the layer 3 address fields in the packets. The learning algorithm imposes no additional traffic overhead on the network. It assumes that the domain is loop free, and it learns about the location of subnets by observing the source addresses of NPDUs being routed. NPDUs addressed to groups are routed by flooding, with the NPDU being propagated through the entire domain.

6.3 Service Interface

The Network service interface consists of the Send_Packet () service request and the Rcv_Packet () indication, as shown in figure 6.3. Again this interface is provided for explanation purposes. Actual implementations may, for example, combine this layer with the Transport layer and only expose the Transport layer interface.

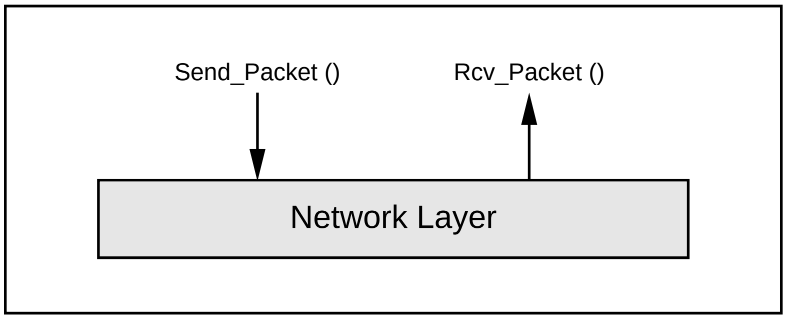


Figure 6.3 Network Service Interface

The syntax of these two interface primitives is:

Send_Packet (address_pair, pduType, PDU, priority, delta_BL, Alt_path)

Rcv_Packet (address_pair, pduType, PDU, priority)

6.4 Internal Structuring of the Network Layer

The LonTalk Network layer performs two functions—address recognition and routing—as shown in figure 6.4.

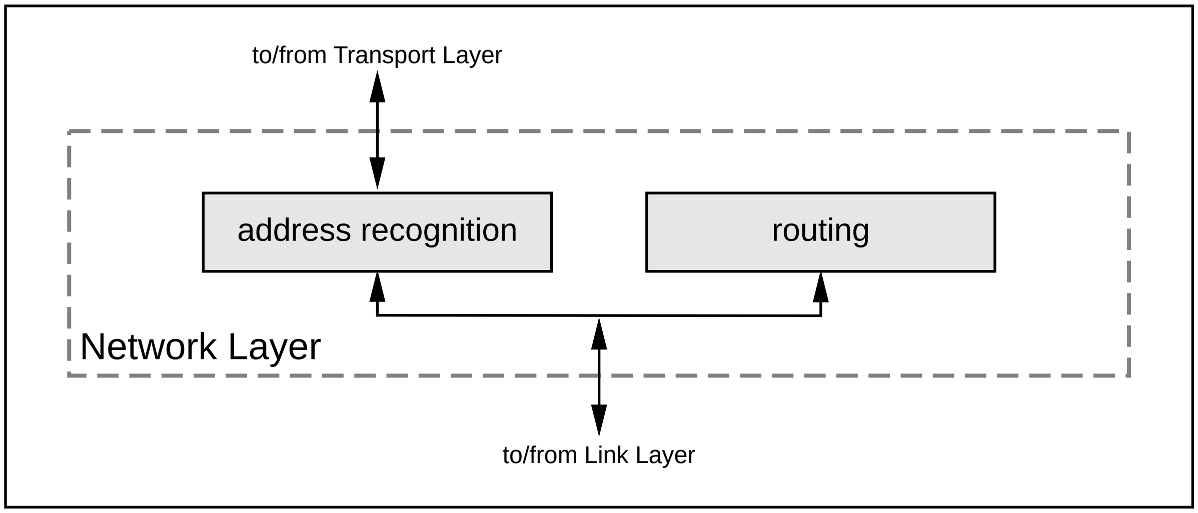


Figure 6.4 LonTalk Network Layer—Internal Structure

6.5 NPDU Format

NPDU format is shown in figure 6.5. An NPDU carries and envelops either a TPDU, SPDU, AuthPDU or APDU. There are no NPDUs defined for internal Network layer use. The numbers above each field in figure 6.5 specify the field size in bits. The symbolic field values used in the figure are assigned in the order

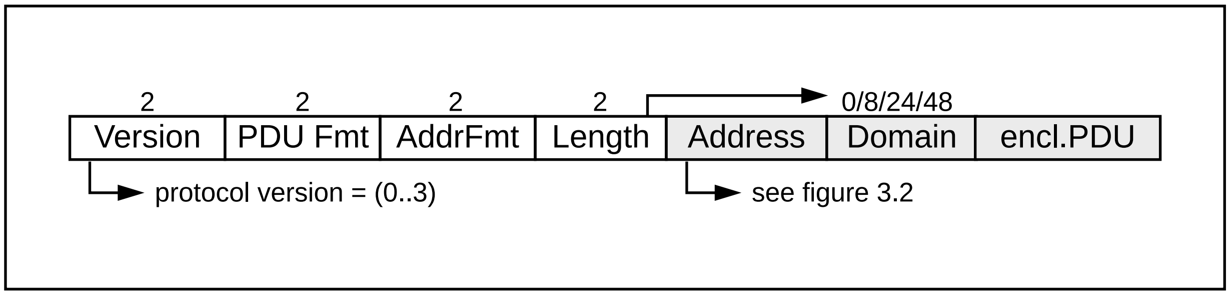


Figure 6.5 NPDU Format

shown, as enumerated ranges (0, 1, 2, 3, …, n). For example, a value of 3 in the encl.PDU field signifies that the enclosed PDU is an APDU. For additional details, including the bit/byte transmission order, refer to Appendix A.

6.6 Address Recognition

Address recognition is performed by each LonTalk node. In this section the information that must be kept by every node is identified without specifying the address matching algorithm, which is implementation specific. For an example of an actual implementation of these data structures, reference the Neuron Chip Data Book published by Motorola and Toshiba.

```
Node_Record = record
    security:           (none, network_mgmt);
    node_address:   array [0..n] of Address_Record;
    end;

Address_Record = record
    case boolean of
        false:  (
            group;   (0..255);
            member:     (0..63);
        true: (
            subnet:  (0..255);
            node:        (0..127);
        end;
```

6.7 Routers

A router performs the routing function for a specific domain. It connects two sets of subnets—one set in the *Up* direction, and the other in the *Down* direction. A router is a logical rather than physical entity; more than one router may be housed within a single routing node.

A router uses three routing functions: $ROUTE_{uc}$(), $ROUTE_{mc}$(), and $ROUTE_{bc}$() to forward NPDUs. The first function specifies how to forward NPDUs addresses to (subnet,-), the second how to forward NPDUs addressed to groups, and the third function specifies how to forward NPDUs when they are sent via the broadcast address format. The functions are tables, where each entry has the following form:

$ROUTE_{uc}$ (DestSubnet) = one of (forward, discard)
$ROUTE_{mc}$(DestGroup) = one of (forward, discard)
$ROUTE_{bc}$ (DestSubnet) = one of (forward, discard)

The entry for an address X specifies whether an NPDU addressed to X should be forwarded or discarded. These functions are called from the side of the router on which the packet was received—that is, algorithm 6.8 executes independently on each side of the router and makes all routing decisions for packets arriving at the side on which it runs. Each side of the router shall have a different set of tables that have the information as to whether the packet should be passed across the router or not.

In order that configured routers do not cause packet looping in those topologies with loops, a configured router should never forward a group or a broadcast addressed packet in the same direction as that of the source subnet encoded within that packet. In the special case of an unconfigured node sending a broadcast message, the source subnet field shall be zero, as shall the domain length. In this case, the router shall modify the packet to have the router's own source subnet and source domain prior to calling ROUTE_{bc}().

6.8 Routing Algorithm

<u>Algorithm 6.8:</u>

Input:

NPDU	the NPDU to be routed

Output:

Decision	one of (Forward, Drop)

Uses:

My_Domain	the domain this router is assigned to
My_Subnet	the subnet within the domain that this side of the router is assigned to
ROUTE_{uc} ()	routing table
ROUTE_{mc}()	routing table
ROUTE_{bc}()	routing table
RouterType	one of: Configured, Learning, Bridge, Repeater

```
Begin { algorithm 6.8 }

    If RouterType = Repeater Then Begin
        Decision := Forward;
        Return;
    end;

    If RouterType = Learning Then
        Execute ROUTING_EVENT of algorithm 6.9;

    If NPDU.Domain <> My_Domain Then
        If RouterType = Bridge or RouterType = Learning Then Begin
            Decision := Drop;
            Return;
        end;
        If NPDU.Domain <> Null_Domain Then Begin
            Decision := Drop;
            Return;
            end;
        Else
            If NPDU.SourceSubnet = 0 and NPDU.DestAddrFmt= Broadcast Then Begin
                NPDU.Domain := My_Domain;
                NPDU.SourceSubnet := My_Subnet;
            end;
        end;
    Else If RouterType = Bridge Then Begin
        Decision := Forward;
        Return;
    end;

    Case NPDU.DestAddrFmt Of
        Subnet/Node:     Decision := $ROUTE_{uc}$ (NPDU.DestSubnet);
        Group:           Decision := $ROUTE_{mc}$ (NPDU.DestGroup);
        Broadcast:       Decision := $ROUTE_{bc}$ (NPDU.DestSubnet);
    end case;

    Return;
end { algorithm 6.8 };
```

6.9 Learning Algorithm — Subnets

The subnet routing table defining the routing function $ROUTE_{uc}$() is created by the algorithm below. Upon initialization, forwarding is used for all subnet addresses. The algorithm subsequently learns about the location of subnets by observing the source addresses of NPDUs being routed. Again, this algorithm executes on each side of the router independently.

Algorithm 6.9:

```
Inputs:

    INIT_EVENT
        always occurs on system reboot; may also occur periodically, allowing the router to adapt to
        changes in network topology

    ROUTING_EVENT
        NPDU          the NPDU to be routed
        MySubnet      the subnet the router is configured on for this side of the router

Output:
    Defines routing function $ROUTE_{uc}$ ()

begin { algorithm 6.9 }
    case event of
        INIT_EVENT:
            Set $ROUTE_{uc}$ () := Forward for all subnet addresses;
            Set $ROUTE_{uc}$ (MySubnet) := Drop;
            Set $ROUTE_{mc}$ () := Forward for all group addresses;

        ROUTING_EVENT:
            $ROUTE_{uc}$ (NPDU.SrcSubnet) := Drop
    end case;
end { algorithm 6.9 };
```

7. TRANSACTION CONTROL SUBLAYER

7.1 Assumptions

The transaction sequencing protocol described in this chapter uses 4-bit transaction numbers which are allocated by the sender and used by the receiver to detect duplicate packets. It is assumed that the network is either order preserving or, if it is not, that packets are delayed approximately uniformly in the multiple paths that they traverse from source to destination. The difference in packet propagation time when there are multiple paths from a given source to a destination must be less that the time it takes for the source to complete one transaction (via an acknowledgment on the shortest path) and begin a second transaction to the same destination. Stale acknowledgments and responses are detected as duplicates, but stale packets which initiate a transaction and arrive after the second transaction has started will not be detected as duplicates.

This implies that store and forward repeaters which operate on a single channel (as opposed to those which connect two channels) must have small buffer pools so that stale packets do not arrive late enough to defeat the duplicate detection mechanism. The same comment holds for multiple bridges or store and forward repeaters, each of which connects the same two physical channels.

The number of concurrent outgoing transactions is restricted to a single priority and a single non-priority transaction. The maximum number of active receive transactions is 16.

The transaction timer, receive timer, and retry count interact to establish the reliability of the duplicate detection mechanism. It is assumed that the receive timer is set to be long enough to cover the configured number of retries, yet short enough so that the transaction ID does not wrap around causing a new transaction to be falsely detected as a duplicate transaction. In implementation it is permissible to have a transaction space for all priority transactions and a second transaction space for all non-priority transactions.

7.2 Service Provided

The transaction control (TC) sublayer is responsible for the common functions related to transaction ordering, sequencing, and duplicate detection. It provides the following services:

- Outgoing Sequencing. To guarantee ordering among outgoing Transport messages and Session layer requests, the TC sublayer controls the allocation of send transaction numbers. It limits the number of concurrent transactions to any destination to ≤ 1 priority and ≤ 1 non-priority transactions;

- Incoming Sequencing and Duplicate Detection. The TC sublayer provides duplicate detection.

7.3 Service Interface

Access to TC services is facilitated by the interface depicted in figure 7.1.

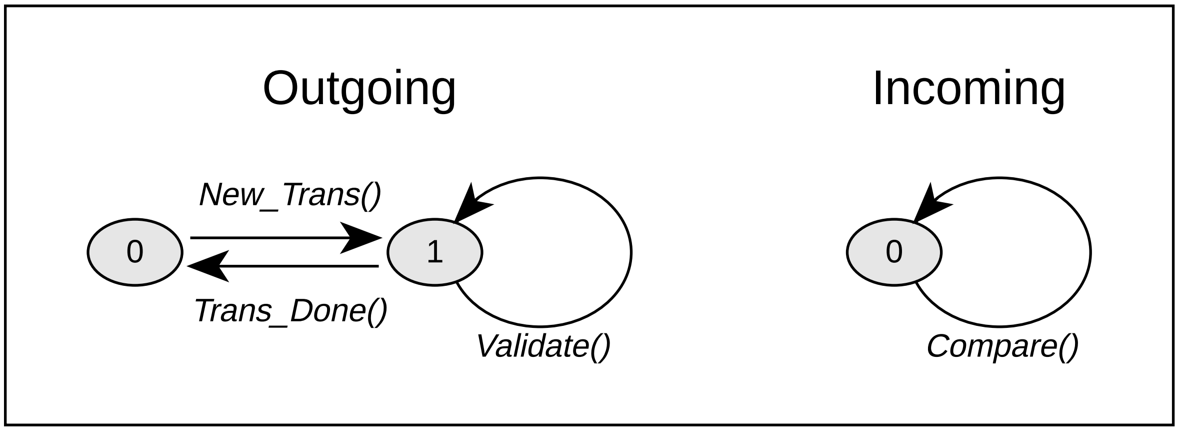


Figure 7.1 Transaction Control Service Interface

The syntax and semantics of the interface primitives are:

New_Trans (Priority) -> (Trans_No)
is used to obtain a transaction number for a new outgoing transaction

Validate (Priority, Trans_No) -> (result)
where result = one of (current, not_current), verifies that Trans_No is in the transmit window

Trans_Done (Priority, Trans_No)
notification of an outgoing transaction completion

Compare (T1, T2) -> (result)
where result is one of (new, duplicate); defines the relationship of T1 relative to T2, where both T1 and T2 are receive transaction numbers

7.4 State Variables

To support the allocation of transaction numbers, the TC sublayer uses a number of destination records shown below.

```
Transaction_CTRL_Record = record
    PriTX                   True or False
    Trans_No:               (0..15); initial value = 0;
    In_Progress:            boolean;
    end;
```

7.5 Transaction Control Algorithm

Algorithm 7.5 is the simple version of the transaction control algorithm in that it provides only two transaction spaces -- one for all priority transactions and one for all non-priority transactions. In any implementation, the first transaction ID provided by New_Trans() after a reset shall be 0 for every transaction space provided by the implementation. New_Trans() shall then increment the transaction ID within the space from 1 to 15 and continue again with 1 so that the transaction ID 0 is used exclusively by the first transaction per transaction space after a reset.

Algorithm 7.5:

```
Inputs and Outputs:
    System_Reset            an event signaling power-up or rebooting
    New_Trans ()->()        service request from Transport or Session sublayer
    Validate ()->()         service request from Transport or Session sublayer
    Trans_Done ()           service notification from Transport or Session
    Compare ()->()      service request

State Variables:
    PriTX_ID                priority transaction ID
    nonPri_TX_ID            non priority transaction ID

begin {algorithm 7.5}
    case event of

    System_Reset:
        begin
        PriTX_ID := 0;
        nonPriTX_ID := 0;
        Initialize both transaction control records—by assigning:
            Trans_No := 0;
            In_Progress := false;
        end;

    New_Trans (priority) -> (Trans_No):
        begin
        Obtain corresponding transaction control record ;
        if {record found }then begin
            block the request; { details implementation specific };
            end;
        else begin{ no current transaction record }
            if (priority = True ) then
                Trans_No := PriTX_ID;
            else
                Trans_no := nonPriTX_ID;
            end;
        end;

    Trans_Done (priority, trans_no):
        Begin
        Obtain transaction control record TXCTRL;
        TXCTRL.Trans_In_Progress := false;
        if (TXCTRL.PriTX = True )then begin
            PriTX_ID := (PriTX_ID+ 1) mod 16;
            if (PriTX_ID = 0 )then
                PriTX_ID := 1;
            end;
        else begin
            nonPriTX_ID := (nonPriTX_ID+ 1) mod 16;
            if (nonPriTX_ID = 0 )then
                nonPriTX_ID := 1;
            end;
        end;


    Validate (Priority, Trans_No) -> (result):
        begin
        Look up transaction control record TXCTRL;
        if (TXCTRL.In_Progress = True ) and (TXCTRL.Trans_No = Trans_No) then
            result = current
        else
            result := not_current;
        end;

    Compare (T1,T2) -> (result):
        begin
        if (T2 = 0 )then
            result := new;
        else if (T2 = T1) then
            result := duplicate;
        else { no need to do more with order-preserving network; old transactions do not exist };
            result := new;
        end;

    end case;
end { algorithm 7.5 };
```

8. TRANSPORT LAYER

8.1 Assumptions

The Transport protocol makes no assumptions apart from relying on the Transaction Control sublayer for correct TPDU sequencing and duplicate detection.

8.2 Service Provided

The Transport sublayer provides the following services:

- Reliable Multicast and Unicast. The transport protocol supports both multicast within a group, and multicast to a group with the sender not being a member of the group. All reliable services have the following attributes: (i) reliable delivery with best effort determined by the number of retries; (ii) assuming the layer 4 timers are set correctly, duplicate detection is provided in all cases except when the sender or receiver just reset. In this case, the reset node shall start with transaction number 0 which may result in a transaction in progress between the two nodes being acted upon more than once; (iii) partial ordering—ordering is preserved but a message is not delivered when delivery fails within the specified number of retries; and, (iv) immediate re-synchronization—following a network partitioning, the very first message is delivered.

- Unacknowledged-Repeated Multicast and Unicast. These services differ from the reliable ones described above only in that no acknowledgment is expected, and that the message is sent repeatedly until the number of repetitions is equal to the retry count. When using this service, the limit of 63 members in a group does not apply—the only limit on the number of members in a group addressed via this service is the number of nodes in a domain.

LonTalk protocol groups are symmetric in that every member of the group can both send and receive.

8.3 Service Interface

For the purpose of the protocol specification it is assumed that the service interface provided to the Session and Application layers has the form shown in figure 8.1.

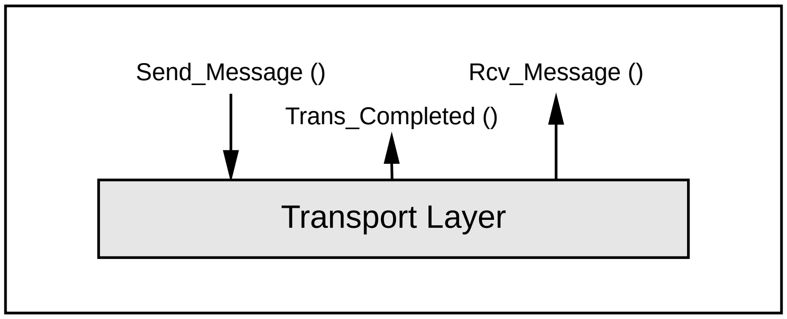


Figure 8.1 Transport Interface To Upper Layers

The syntax and semantics of the Transport layer interface are:

Send_Message (Address, APDU, priority) -> (TID)

Trans_Completed (TID, Result)

Rcv_Message (APDU)

TID, above, is a unique identifier for the transaction.

8.4 TPDU Types and Formats

TPDU syntax is shown in figure 8.2; the number above each field specifies the field size in bits. The symbolic field values shown in the picture are mapped onto numeric ranges (0, 1, 2, 3, ...) in the order shown. Additional details, such as the bit/byte transmission order are defined in Appendix A.

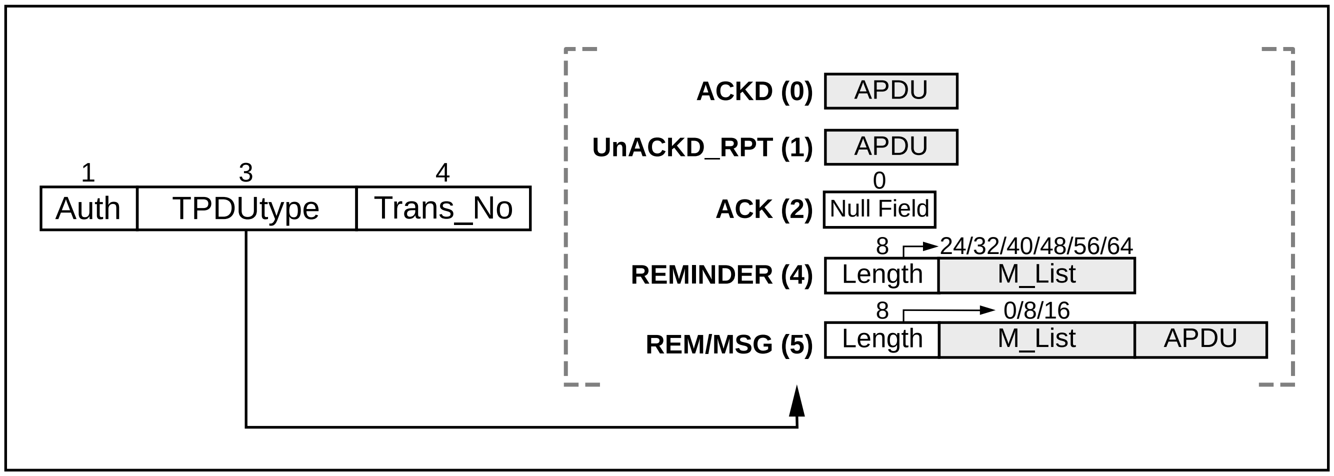


Figure 8.2 TPDU Types and Formats

The *Acknowledged Message* (ACKD) TPDU is used for the first transmission of a message. It is used with addressing formats #1, #2a, and to a limited extent #3. Unlike the Unacknowledged message TPDU, it must be acknowledged by all addressed recipients. This TPDU is used for acknowledged initial TPDUs (both unicast and multicast) as well as unicast reminders. Multicast reminders are covered below. Refer to figure 3.2 or Appendix A to review the addressing formats.

The *Unacknowledged-repeated Message* (UnACKD_RPT) TPDU is identical to its acknowledged counterpart with one exception: on its reception, no acknowledgments are returned to the sender. This TPDU is used with no modifications for the unacknowledged-repeated service. Simple unacknowledged messages have no TPDU header, and thus have no duplicate detection.

The *Message-Reminder* (REM/MSG) TPDU facilitates selective soliciting of acknowledgments for multicast transactions. REM/MSG *type 5* is used when the highest numbered group member from which the sender has received an acknowledgment is < 16; this TPDU contains both the member list (M_List []) and the APDU. The length field specifies the size of the M_List field (in bytes); a value of 0 in M_List [X] indicates that member X’s acknowledgment has not been received by the sender, whereas a value of 1 indicates that the acknowledgment has been received. Finally, when Length=0 the M_List[] field is absent and the meaning is “all members should acknowledge."

Type 4 is a plain *reminder*, without an APDU (see figure 8.2). It is used in cases where the highest numbered member that has acknowledged the message is ≥ 16. Acknowledgments are solicited using the TPDU pair (REMINDER, ACKD) and this pair is logically equivalent to a single type 5 REM/MSG TPDU used when the members needing to acknowledge may be encoded within the type 5 format. (A separate solution is provided for large groups because of the need to limit maximum TPDU size.)

The *Acknowledgment* (ACK) TPDU is null. It uses addressing format #2a (unicast acknowledgment) or #2b (group acknowledgment). The Trans_No field conveys the transaction being acknowledged.

Any TPDU that requires an acknowledgment can be flagged as an authenticated packet by setting the Auth bit to ‘1’.

8.5 Protocol Diagram

The diagram in figure 8.3 is intended to augment—but not replace—the protocol description in sections 8.7-8.8.

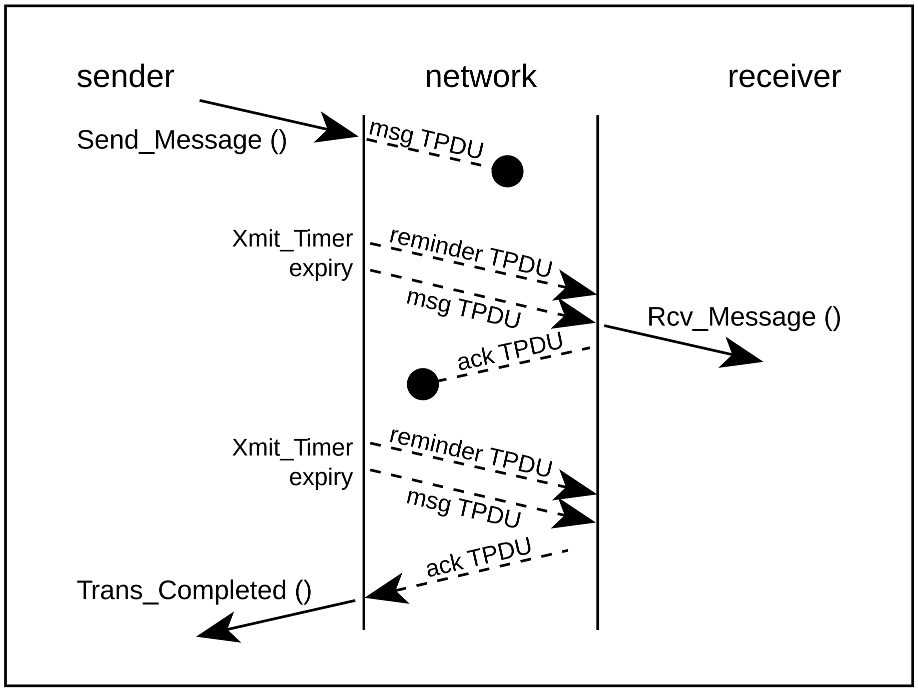


Figure 8.3 Transport Protocol Diagram
Multicast with a Loss of Both the Message and the ACK TPDUs

8.6 Transport Protocol State Variables

The Transport protocol consists of two independent functions—Send and Receive. To support the send function, the Transport layer keeps one Transmit record per transaction in progress. A shared pool of Receive records facilitates message reception.

```
TRANSMIT_Record = record
    ACK_Received:     array [0..63] of Boolean;
    Dest_Count:       0..63; { number of destinations }
    ACK_Count:        0..63;
    Xmit_Timer:       timer;
    Retries_Left:     0..15;
    priority:         one of True/False;
    TPDU_ptr:     pointer to the TPDU being transmitted;
    end;

RECEIVE_Record = record
    Source:           Unicast address;
    Destination:      Unicast or Multicast address;
    Trans_No:         0..15;
```

```
Rcv_Timer:       timer;
L4_State:        one of (not_delivered, delivered , authenticated, authenticating,
                          non-authenticated, waiting;
priority:        one of {True, False};
checksum:        used for authentication
response:        byte—used for request / response protocol—Layer 5
end;
```

8.7 The Send Algorithm

The simplified transmit FSM is shown in figure 8.4, with full details in algorithm 8.7 below. The algorithm resets the re-transmission timer, Xmit_Timer, whenever an acknowledgment is received, as opposed to once per Expiration period.

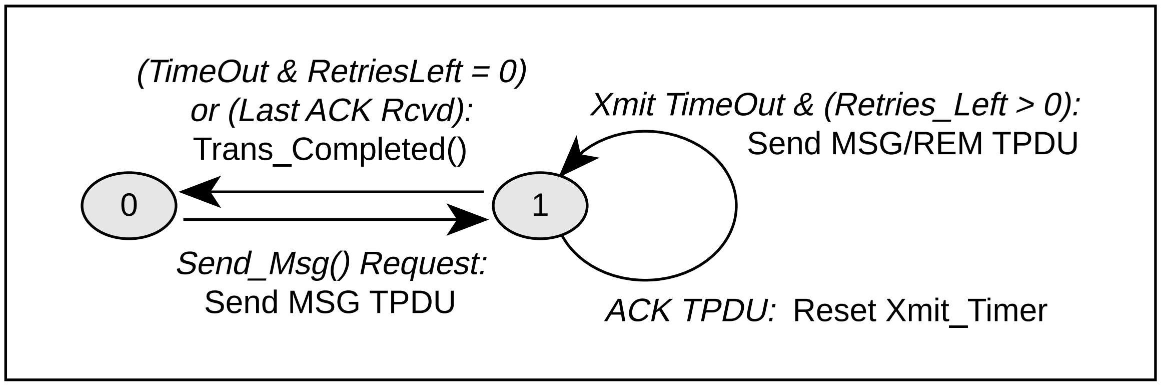

(TimeOut & RetriesLeft = 0)
or (Last ACK Rcvd):
Trans_Completed()
Xmit TimeOut & (Retries_Left > 0):
Send MSG/REM TPDU
0
1
Send_Msg() Request:
Send MSG TPDU
ACK TPDU: Reset Xmit_Timer

Figure 8.4 Transport Protocol—Send FSM

<u>Algorithm 8.7:</u>

Events driving the algorithm:

Send_Message ()	service request (see 8.3)
TPDU_In	ACK TPDU from the Network layer
Xmit_Timer_Expiration	timeout of the retransmission timer
Challenged	receiver node issued an authentication challenge
AuthPDU_in	challenge from receiver node

Outputs:

Trans_Completed ()	transaction completion indication to the Application layer

State Variables:

XR	transmit record for this transaction

```
begin { algorithm 8.7 }
Case event of
    Send_Msg (Address, APDU, priority) -> (TID):
        begin
        New_Trans (priority)-> (Trans_No) ; {request to the TC sublayer}
        Allocate and initialize transmit record XR;
        if { addr_type = multicast } then
```

```
            XR.Dest_Count := {group_size - 1};
        else XR.Dest_Count :=1;
        Create MSG TPDU;
        Send_Packet (...,TPDU) request to the Network layer—see 6.3;
        Start  Xmit_Timer;
        end;

    ACK TPDU Received:
    begin
        Retrieve the associated Transmit record XR;
        Validate (TPDU.priority,TPDU.Trans_no) -> (result) ;
        if (result = current) then begin
            if (addr_type = multicast) then begin
                if (XR.ACK_Received [member] <>1) then begin
                    XR.ACK_Received[member] := 1;
                    XR.ACK_Count := XR.ACK_Count + 1;
                    end;
                if (XR.ACK_Count = XR.DestCount )then
                    Terminate_Trans (XR, Success);
                else Restart Xmit_Timer;
                end;
            end;
        end;
    end;

    Challenged:
        begin
        Retrieve the associated Transmit record XR;
        Validate(AuthPDU_in.priority,AuthPDU_in.Trans_no) -> (result);
        if (result = current)then
            Reply(XR.Trans_no, AuthPDU_in);
        end;

    Xmit_Timer_Expiration :
        begin
        Retrieve transmit record XR;
        If (XR.Retries_Left = 0) then
            if (XR.TPDU_ptr.TPDUtype <> UnAckd_RPT ) then
                Terminate_Trans (XR, Failure);
            else Terminate_Trans(XR,Success);
        else begin
            XR.Retries_Left := XR.Retries_Left - 1;
            Start  Xmit_Timer;
            if (addr_type = multicast) then begin
                Depending on max member # ack'd, compose MSG/REM TPDU or (REM, MSG) pair;
                Send the MSG/REM TPDU or the (REM,MSG) pair;
            else begin
                Send the initial packet pointed to in XR.TPDU_ptr;
                end;
            end;
        end;
    end;
    end case;
```

```
Procedure Terminate_Trans (XR, status);
    begin
    provide Trans_Completed (TID, status) indication to the Application layer;
    Trans_Done (XR.TPDUptr.priority, XR.Trans_No);
    de-allocate XR;
    end;

end { algorithm 8.7 };
```

8.8 The Receive Algorithm

Message reception uses the timer-based mechanism. Figure 8.5 shows the receive FSM for a single transaction, while algorithm 8.8 specifies full details.

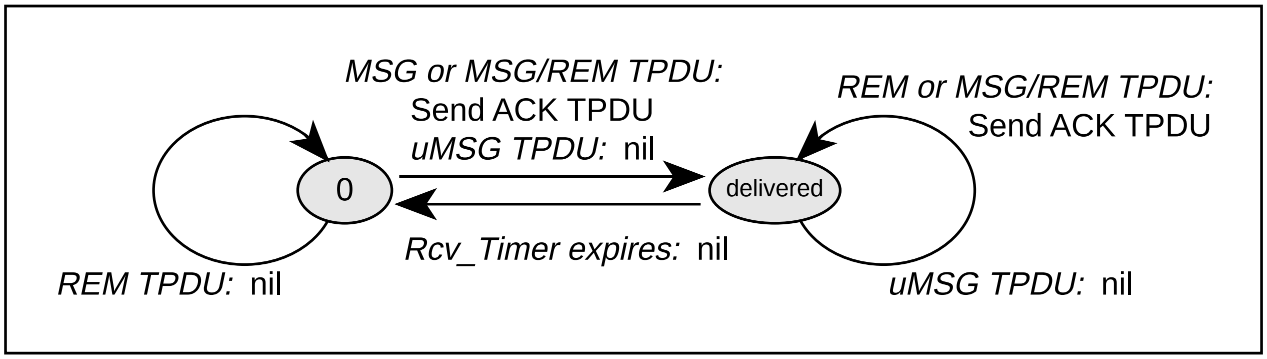


Figure 8.5 Transport Protocol—Receive FSM

<u>Algorithm 8.8:</u>

Input:	
TPDU_in	a TPDU received from the Network layer
Timer Expiration	of Rcv_Timer in Receive_Record
AuthPDU_in	Reply to authentication challenge issued by transaction initiator
Outputs:	
ACK TPDU	to the remote Transport entity
Rcv_Message ()	indication to the Application layer
State Variables	
RR pool	pool of Receive Records

```
begin { algorithm 8.8 };
case event of:
        TPDU_In:
                Process_TPDU(TPDU_in, priority)

        Rcv_Timer Expiration (RR):
                De-allocate receive record RR;
                De-allocate authentication record if Authrcd.TID = RR;
        end case ;

Procedure Process_TPDU (TPDU_in, priority);
    var result: integer;
    begin
    Retrieve the associated RR (RR = nil if none exists);
    if (RR <> nil )then begin
        Compare (RR.Trans_No, TPDU_in.Trans_No) -> (result);
        if (result <> duplicate )then
            Reset Rcv_Timer;
        end
    else begin
        Allocate and initialize a RR by assigning:
            RR.Source := TPDU_in.src_addr;
            RR. Trans_No := TPDU_in.Trans_No;
            RR.L4_State := not_delivered;
            RR.Priority := priority
            RR.Destination := {initialize destination multicast or unicast address};
            Start Receive Timer;
        end;

    case TPDU_in.TPDUtype of

        ACKD, UnACKD_RPT:
            begin
            if (RR.L4_State <> delivered )then
                if (auth = True)then
                    Initiate_Challenge (RR, TPDU_in) ;        {algorithm 9.11 }
                else begin
                    provide Rcv_Message () indication to the Application layer;
                    RR.L4_State := delivered;
                    end;
            if (TPDUtype = ACKD) and (RR.L4_State = delivered)then
                Compose and send ACK TPDU;
            end;


        REMINDER, REM / MSG:
            begin
            int temp;

            if (RR.L4_State <> delivered) then
                if (auth = True) then
                    Initiate_Challenge (RR,TPDU_in) ;         {algorithm 9.11 }
                else begin
                    provide Rcv_Message () indication to the Application layer;
                    RR.L4_State = delivered;
                    end;

            if (TPDUtype = ACKD)then
                if (RR.L4_State = delivered) then begin
                    if (TPDU.Length <> 0)then begin
                        temp = my_member_no / 8 + 1;       {integer math with truncation };
                        if (temp ≤ TPDU.Length) then
                            if (TPDU_in.M_List [my_member_no] = 0) then begin
                                compose and send ACK TPDU;
                                end;
                    else begin
                        compose and send ACK TPDU;
                        end;
                    end;
                end;
            end;

        AuthREPLY:
            begin
                Process_Reply(RR, AuthPDU_in);          { algorithm 9.11}
                provide Rcv_Message () indication to the Application layer;
                RR.L4_State = delivered;
            end;

        end case;
    end procedure;
  end { algorithm 8.8 };
```

8.9 RR Pool Size and Configuration Engineering

Space for the Transport protocol variables defined in section 8.6 is allocated at the time the application program is linked with the protocol code. With the exception of the Receive Record (RR) pool, these variables are of fixed size, and consequently no engineering decisions have to be made regarding how many of each size to allocate. The size of the RR pool limits the number of *concurrent receive* operations on a node ("concurrent” means concurrent within the context of Rcv_Timer) and should be engineered according to the number of concurrent transactions expected within the receive timer interval.

8.10 Number of Retries

The number of retries should be large enough to ensure that message delivery is successfully completed with acceptable probability, e.g., ≥ 99%.

If the delivery probability of a single attempt is p, then the probability that message delivery within a group of n succeeds in ≤ k attempts is shown in table 8.1 below. (It is assumed that the single attempt probability p is the same for all destinations.)

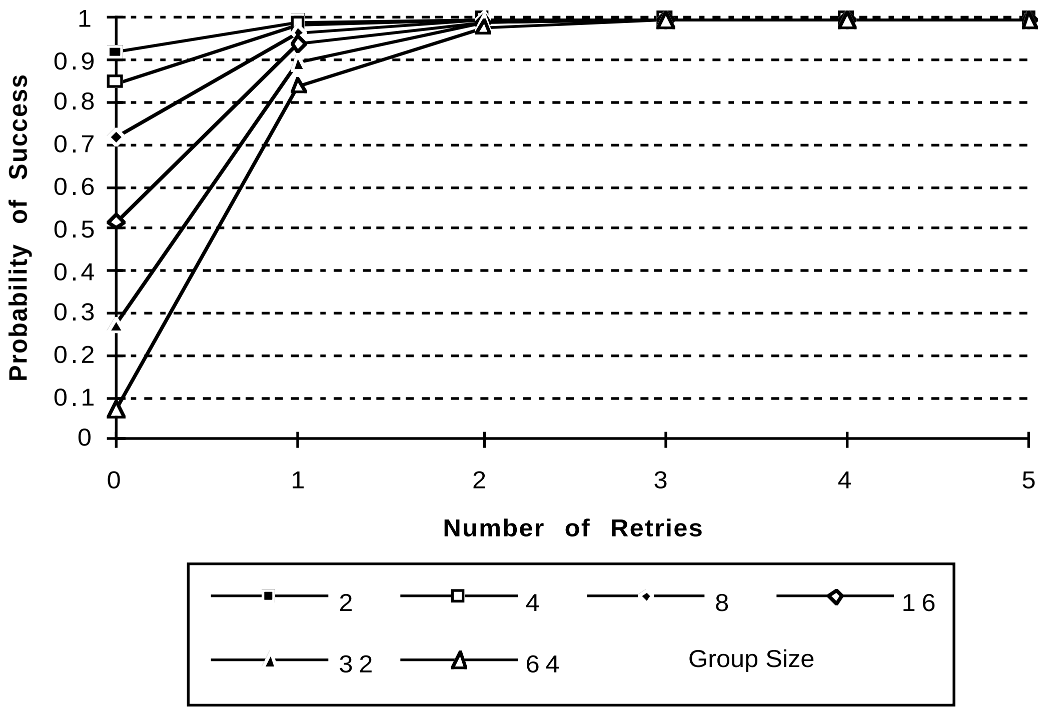
$$P\{\text{no retry}\} = (1-p)^{n}$$

$$P\{\le k \text{ retries}\} = \sum_{i=0}^{k} \binom{k+1}{i} p^{i} (1-p)^{k-i+1} (1-p)^{n-1} \qquad k \ge 0,\ n \ge 2$$

Table 8.1 Probability of Delivery

Note: For a single channel $p = 1 - (1/2w + p_e)$, where w is the size of the MAC layer randomizing window and p_e is the probability of packet loss because of a transmission error.

The above probabilities are tabulated in graph 8.1. Given a group size and the error probability p, the required number of retries for, say 99.5% probability of success, can be read off the graph.



Graph 8.1 Probability of Transaction Completion in k Retries for a Single Channel Without Priority Messages

However, pragmatic considerations must be also included in any real system design. For example, there is no need for a large number of retries with transactions that are repeated periodically. Additionally, when the acknowledgments to a multicast message would take up a significant percentage of the bandwidth of the channel, it is more likely that a message will be properly delivered using the "unacknowledged-repeated" service rather than the acknowledged multicast service. In practice, it is expected that a Retry_Count in the (2..5) range will cover most situations.

8.11 Choice of Timers

There are three timers used by the Transport protocol. They are the:

Xmit_Timer	the layer 4 retransmission timer
Repeat_Interval_Timer	the UNACKD_RPT interval timer
Rcv_Timer	the receive record timer (see 8.6, 8.9)

Xmit_Timer is reset on every ACK TPDU or Authentication challenge reception, while Rcv_Timer is reset whenever a MSG or MSG/REM TPDU that has a new transaction number is received for this destination. The Repeat_Interval_Timer determines the interval between the UNACKD_RPT packets sent by the sender. The recommended methodology for calculating the timer values is shown in table 8.1. These recommendations are for a single channel. For multichannel networks, the calculation depends upon the speed of the router and the number of buffers. Empirical measurements with LONWORKS routers show about 4 ms of delay across a router with the default buffering and with both sides running at a 10 MHz input clock rate. Since buffering and clock rates are adjustable, the system designer must make some measurements to create an actual network configuration (unless it is the one just mentioned).

Retry Count = 2-5	(see section 8.10 and graph 8.1)
Xmit_Timer ≥ 3* packet cycle time+margin	(margin = best case tx completion time)
Rcv_Timer ≥ Xmit_Timer * (Retry_Count+2)	

Where	"3* packet cycle time" assumes that the average station on the channel must wait two packet cycles of delay to access the network. Then, following network access, the transmission time of the average packet (also the packet cycle time) is added to the timer value. Finally, the time it takes the receiver to process the packet and send the acknowledgment is added. This is a function of the clock speeds of the associated nodes.

Table 8.1 Methodology for Calculating Timer Values

9. SESSION LAYER

9.1 Assumptions

The Session sublayer makes no assumptions apart from relying on the Transaction Control sublayer for correct SPDU sequencing and duplicate detection.

9.2 Service Provided

The Session layer provides a single service:

- Request-Response. This service facilitates application communication similar to a remote procedure call. In particular, it allows a client to make a request to a remote server and receive a response to this request.

  Non-idempotent transactions are to be executed *"at most once"* (i.e., exactly once or not at all). Non-idempotent transactions are those where the action depends on a prior state, such as "open the valve an additional 10%." The protocol considers a transaction to be non-idempotent if and only if its response length is ≤ 1 byte.

  Requests with response length > 1 byte are considered *idempotent*; such requests may be executed *"several times"* (i.e., zero or more times). Idempotent transactions are those where the action may be repeated any number of times, and the effect is the same. An example of an idempotent command is "read the first 10 table values".

  The distinction between idempotent and non-idempotent transactions is based upon the size of the response, in order to limit the amount of storage required for transaction records within a node. If a transaction has a response length > 1 byte but may not be executed more than once, it is the responsibility of the application to save the response and send it again. This is facilitated by the session layer in that notification that the request is a duplicate is provided to the application layer by the session layer.

  A request-response transaction fails unless the server response is generated within the limit imposed by the Request-Response timers and retransmissions (see 9.10).

9.3 Service Interface

For the purpose of this protocol specification it is assumed that the service interface to the application layer has the form shown in figure 9.1.

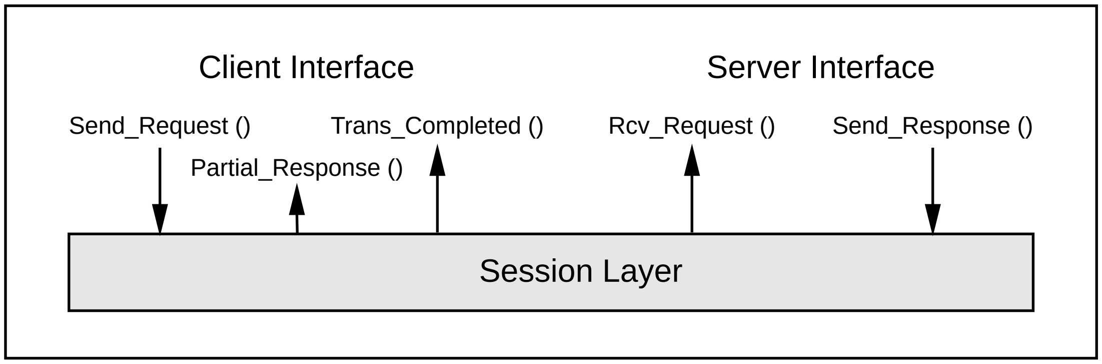


Figure 9.1 Session Layer Interface

The syntax of the service interface primitives is given below. TID is a unique identifier for the transaction (section 7.5 provides implementation details). Duplicate is a boolean that, when true, indicates that the client is retrying a previously executed request.

```
Send_Request (Address, APDU, priority) -> (TID)          { client }
Partial_Response (TID,  APDU, priority)
Trans_Completed (TID, Result)

Rcv_Request (TID,  APDU, duplicate, priority)       { server }
Send_Response (TID,  APDU, priority)
```

9.4 Internal Structure of the Session Layer

The LonTalk Session sublayer is internally structured as shown in figure 9.2.

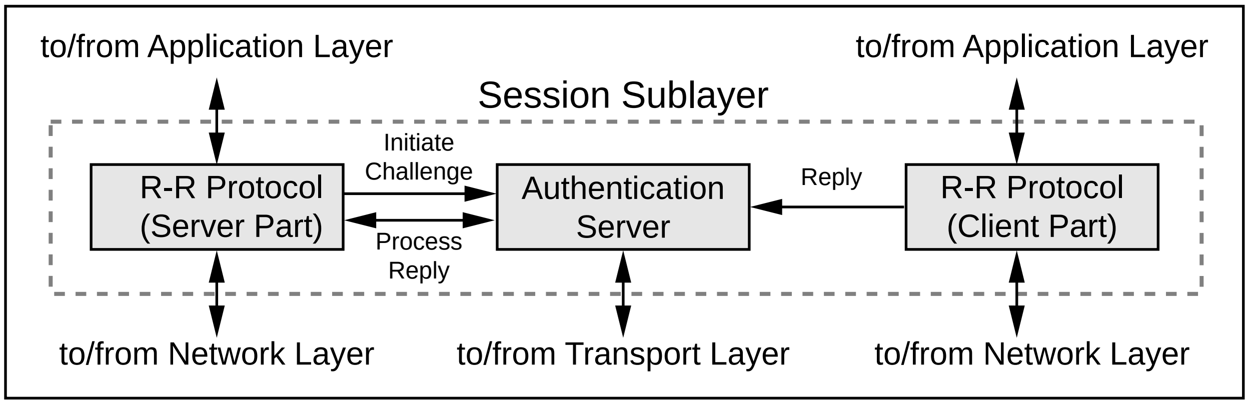


Figure 9.2 Session Sublayer—Internal Structuring

The Request-Response protocol accesses the Authentication server via the following three calls(details in 9.10). The transport layer also accesses the Authentication server with the same three calls.

Initiate_Challenge(RR,PDU) -> (null)	{ challenger }
Reply(XR, PDU)-> (null)	{ challengee }
Process_Request(RR,PDU) -> (pass/fail);	{ challenger }

9.5 SPDU Types and Formats

SPDU formats are shown in figure 9.3, where the number above each field specifies the field width in bits. The symbolic values shown in the picture are mapped onto numeric ranges (0, 1, 2, 3, ...) in the order shown.

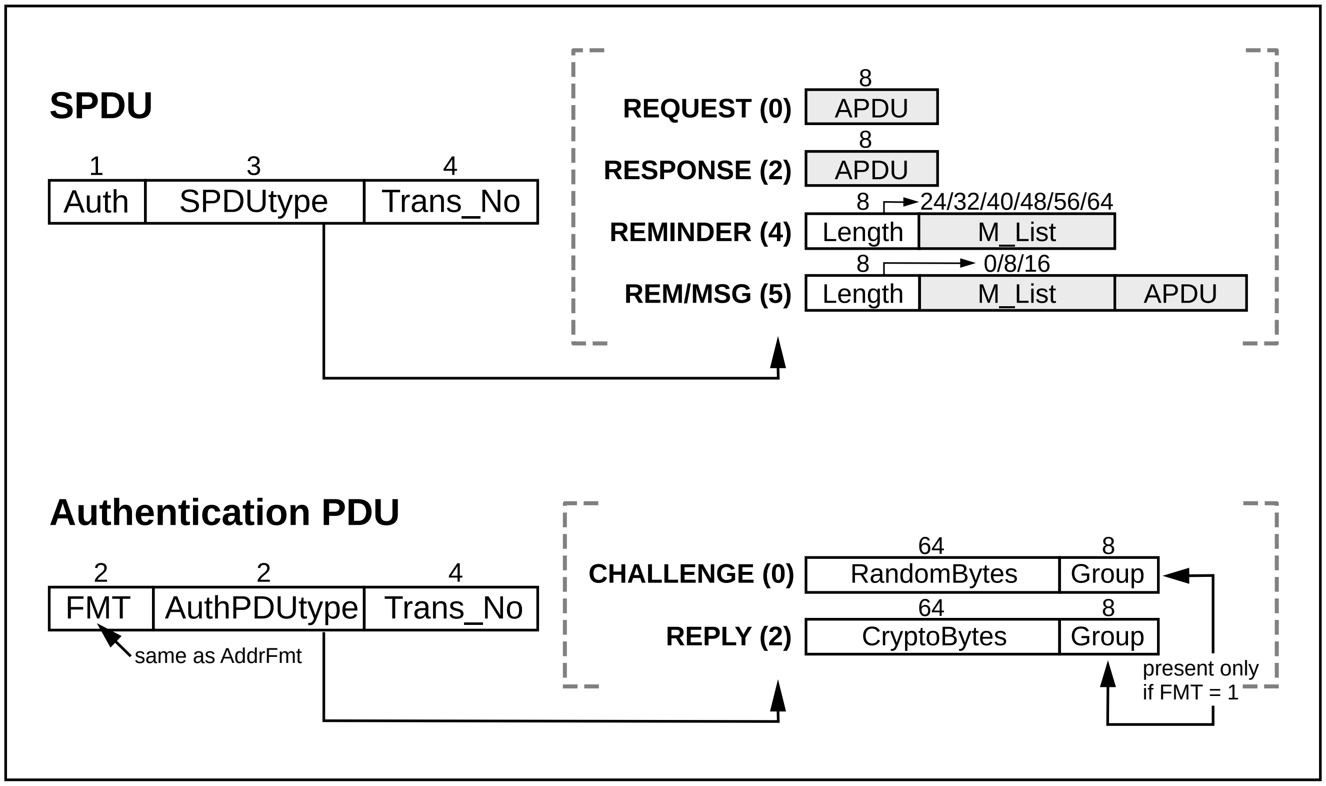


Figure 9.3 LonTalk SPDU Types and Formats

The Request-Response protocol uses three basic PDU types: Request (REQUEST), Response (RESPONSE), and combined Request-Reminder (REM/MSG). The syntax (packet layout) and semantics (packet processing) of these three basic SPDU types correspond closely to that of ACKD, ACK, and REM/MSG TPDUs.

The *Request* (REQUEST) SPDU is used with the first transmission of the request. It employs addressing formats #1, #2a, and to a limited extent #3 and #0. Address format #0 is used by a special network management command to broadcast a request searching for any nodes that have not been configured. Address format #3 is then used to configure those nodes that respond to this special network management command.

The *Request-Reminder* (REM/MSG) SPDU facilitates selective soliciting of responses. REM/MSG *type 5* is used in groups where the highest member number needing to acknowledge is < 16; this SPDU contains both the member list (M_List []) and the APDU (i.e.., the request itself). The Length field specifies the size of the M_List field (in bytes); a value of 0 in M_List [X] indicates that member X's response has not been received by the requester, whereas a value of 1 indicates that the response has been received.

Type 4 is a plain reminder, without a request (see figure 9.3). It is used where the highest member number needing to acknowledge the reminder is ≥ 16; in this case, responses are solicited using the SPDU pair (REMINDER-type 4, REQUEST-type 0) and this pair is logically equivalent to a single type 4 REM/MSG SPDU used in small groups. (A separate solution is provided for large groups because of the need to limit maximum SPDU size.) Finally, when Length = 0 the M_List[] field is absent and the meaning is "all members should acknowledge."

The *Response* (RESPONSE) SPDU uses addressing format #2a (unicast acknowledgment) or #2b (group acknowledgment). The Trans_No field conveys the transaction being acknowledged. The length of the APDU implicitly defines the type of transaction: if the response can be stored in a single byte, the transaction is treated as non-idempotent. Otherwise the transaction is treated as idempotent.

Authenticated SPDUs (Auth bit set to '1') identify requests that are to be authenticated by the recipient. In all other respects, they are identical to the SPDUs which are not authenticated.

Authentication . The authentication server is available to the transport and session layer protocols. It provides a one-way authentication service. It is the client's responsibility to initiate authenticated transactions when required. This is done by setting the Auth bit in the SPDU or TPDU. When a TPDU or SPDU is received with the Auth bit set, the server shall challenge using the "challenge" AuthPDU. The client then computes a transformation based upon the server's challenge, the original APDU sent by the client, and the client's authentication key. The result of this transformation is sent to the server using the "reply" AuthPDU. When the server receives the reply, its contents are compared to the transformation computed by the server. If they match, the transaction is authenticated. In all cases the SPDU/TPDU is passed to the application layer for processing along with notification as to whether the authentication failed or succeeded. Note that if the application layer on the server node has no requirement for authentication of the particular transaction, it may choose to honor the request even if the authentication failed. If the application layer chooses not to honor the request, it simply discards the APDU without further processing.

9.6 Protocol Timing Diagrams

The protocol timing diagrams in figures 9.4 (a) and (b) are intended to provide an intuitive feeling for the session layer protocols and to augment (but not to replace) the protocol specification in sections 9.7-9.13. Note again that for needed acknowledgments from group member numbers < 16, the REM/MSG SPDU is used and is functionally equivalent to the (REMINDER, REQUEST) pair.

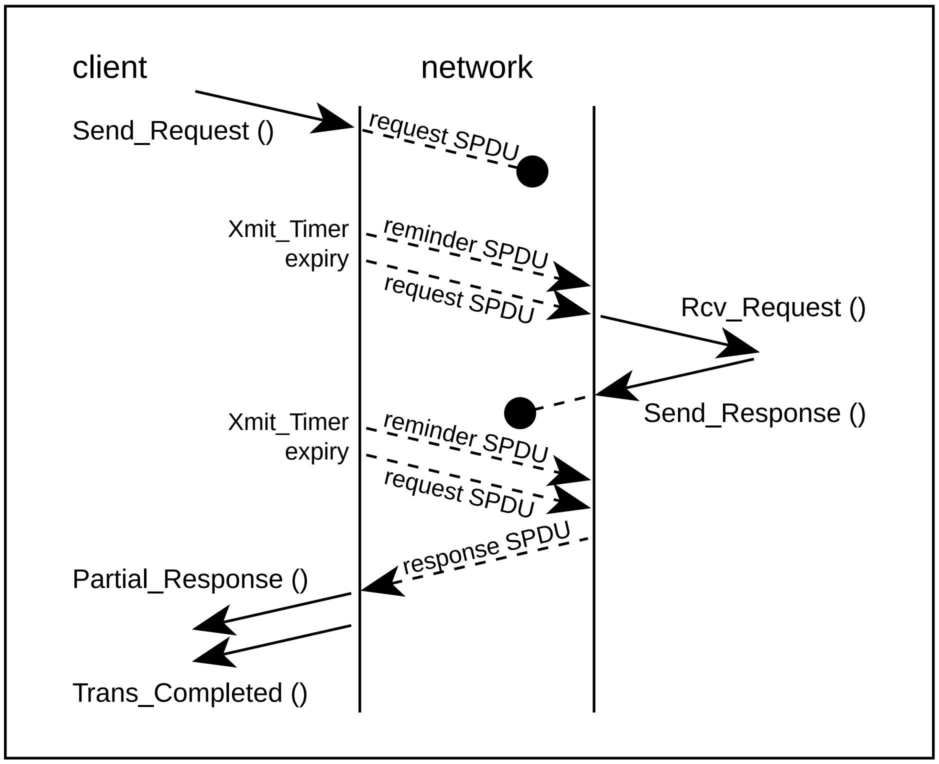


Figure 9.4 (a) Non-Idempotent Request with Multiple SPDU Losses

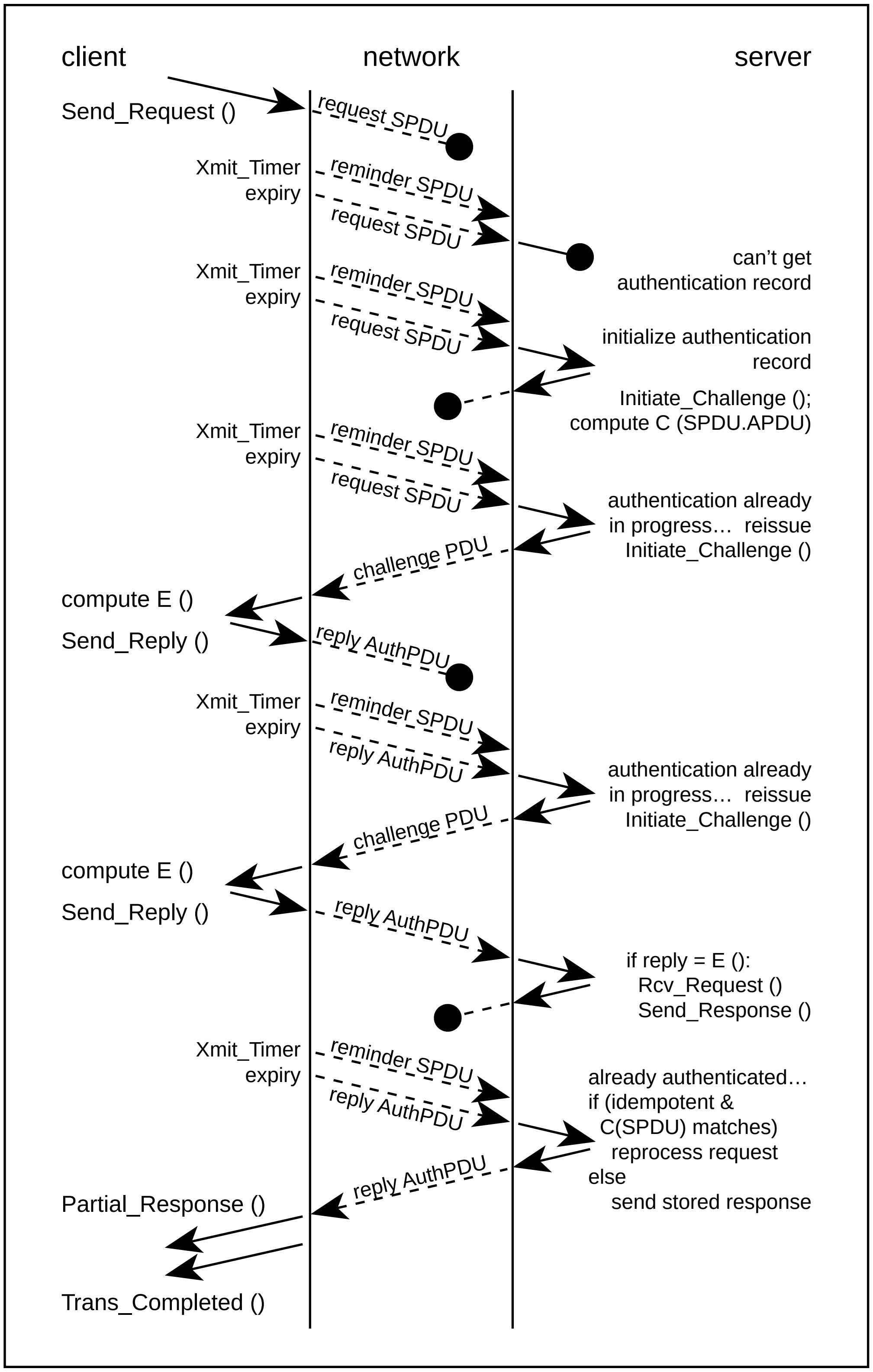


Figure 9.4 (b) Secure Idempotent Request with Multiple SPDU Losses

9.7 State Variables

Like the Transport protocol, the Request-Response protocol maintains one Transmit record per transaction in progress, and a shared pool of Receive records facilitates message reception. As shown below, these records differ in minor details from those used by the Transport protocol.

```
TRANSMIT_Record = record
    ACK_Received:     array [0..63] of Boolean;
    Dest_Count:       0..63;        { number of destinations }
    ACK_Count:        0..63;
    Xmit_Timer:       timer;   { implementation dependent }
    Retries_Left:     0..15;
    SPDU_ptr:     pointer to the SPDU being transmitted;
    priority:         one of True/False;
    Linked_Trans:     TID of a receive transaction in suspended state;
    end;


RECEIVE_Record = record
    Source:           Unicast or Multicast address;
    Destination:      Unicast or Multicast address
    Trans_No:         0..15;
    Rcv_Timer:        timer;   { see section 7.10 }
    L5_state:         one of {nil, executing, done1, done2}
    priority:         one of True/False;
    Reply_type:       one of (application, netmgmnt, netdiagnostic, foreign frame)
    Authenticated: one of True/False;
    Checksum          byte checksum for authentication
    Reply:        Data;        /* valid if and only if L5_State = done1 */
    end;
```

9.8 Request-Response Protocol — Client Part

A simplified FSM for the client part of the Request-Response protocol is shown in figure 9.5, with the detailed specification following in algorithm 9.8.

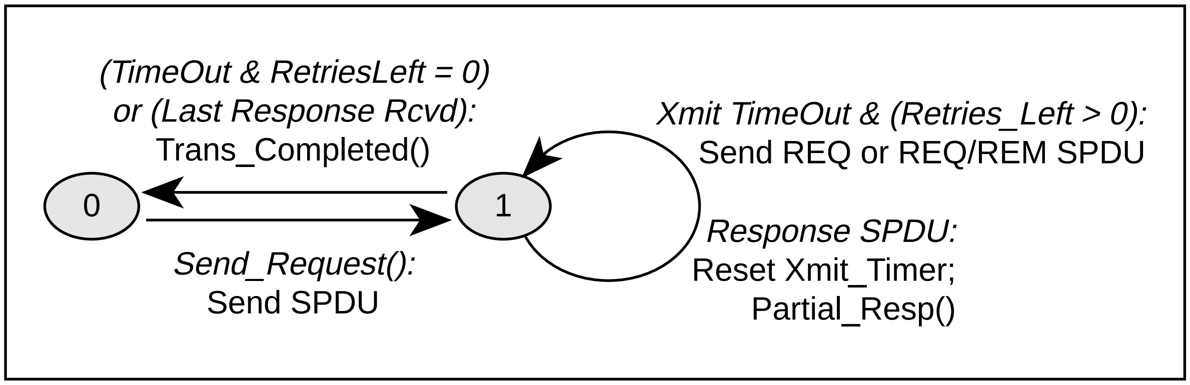


Figure 9.5 Request-Response Protocol—Client FSM

Algorithm 9.8:

Events driving the algorithm:

Send_Request ()	from the Application layer
RESP SPDU	from the Network layer
Xmit_Timer Expiration	timeout of the retransmission timer
Challenged	authentication challenge received from remote server
AuthPDU_in	

Outputs:

Partial_Response ()	indication to Application layer
Trans_Done ()	indication to Application layer

State Variables:

XR	transmit record for this transaction

```
begin { algorithm 9.8 }

Case event of
    Send_Request (Address, APDU, priority) -> (TID):
        begin
        New_Trans (priority) -> (Trans_No); {request to the TC sublayer}
        Allocate and initialize transmit record XR;
        if {addr_type = multicast } then
            XR.Dest_Count := {group_size - 1};
        else XR.Dest_Count :=1;
        Create Request SPDU;
        Make Send_Pkt (...,SPDU) request to the Network layer;
        Start Xmit_Timer;
        end;

    RESP SPDU Received:
        begin
        Validate (SPDU.priority, SPDU.Trans_No) -> (result);
        if (result = current) then begin
            Retrieve the associated Transmit record XR;
            provide Partial_Response indication () to the Application layer;
            if {addr_type = multicast } then begin
                if (XR.ACK_Received [member] <>1) then begin
                    XR.ACK_Received[member] := 1;
                    XR.ACK_Count := XR.ACK_Count + 1;
                    end;
                if (XR.ACK_Count = XR.DestCount ) then
                    Terminate_Trans (XR, Success);
                else Restart Xmit_Timer;
                end;
            end;
        end;

    Challenged:
        begin
            Retrieve the associated Transmit record XR;
            result := Validate(XR.SPDU_ptr.priority,XR.SPDU_ptr.Trans_no);
            if (result = current) then
                Reply(XR.SPDU_ptr.Trans_no,AuthChallengePDU);       {algorithm 9.11}
            end;
        end;

    Xmit_Timer_Expiration :
        begin
        Retrieve Transmit record XR;
        If (XR.Retries_Left = 0 ) then
            if (XR.SPDU_ptr.SPDUtype <> UnAckd_RPT) then
                Terminate_Trans (XR,Failure)
            else
                Terminate_Trans (XR,Success);
        else begin
            XR.Retries_Left := XR.Retries_Left - 1;
            Start Xmit_Timer;
            if {addr_type = multicast} then begin
                Depending on max member # ack'd, compose MSG / REM SPDU or (REM, MSG) pair;
                Send the MSG / REM SPDU or the (REM, MSG) pair;
                end;
            else Send the initial packet pointed to in XR.SPDU_ptr;
            end;
        end;
    end case;

    Procedure Terminate_Trans (XR, Status);
        begin
        provide Trans_Completed () indication to the Application. layer
        Trans_Done (XR.SPDUptr.priority,XR.SPDU_ptr.Trans_no);
        de-allocate XR;
        end;

end { algorithm 9.8 };
```

9.9 Request-Response Protocol — Server Part

A simplified FSM for the server part of the Request-Response protocol is shown in figure 9.6, with the full description following. The protocol treats all transactions with response size > 1 byte as idempotent, implying that it may execute them more than once. Transactions with response size of only 1 byte are never executed more than once.

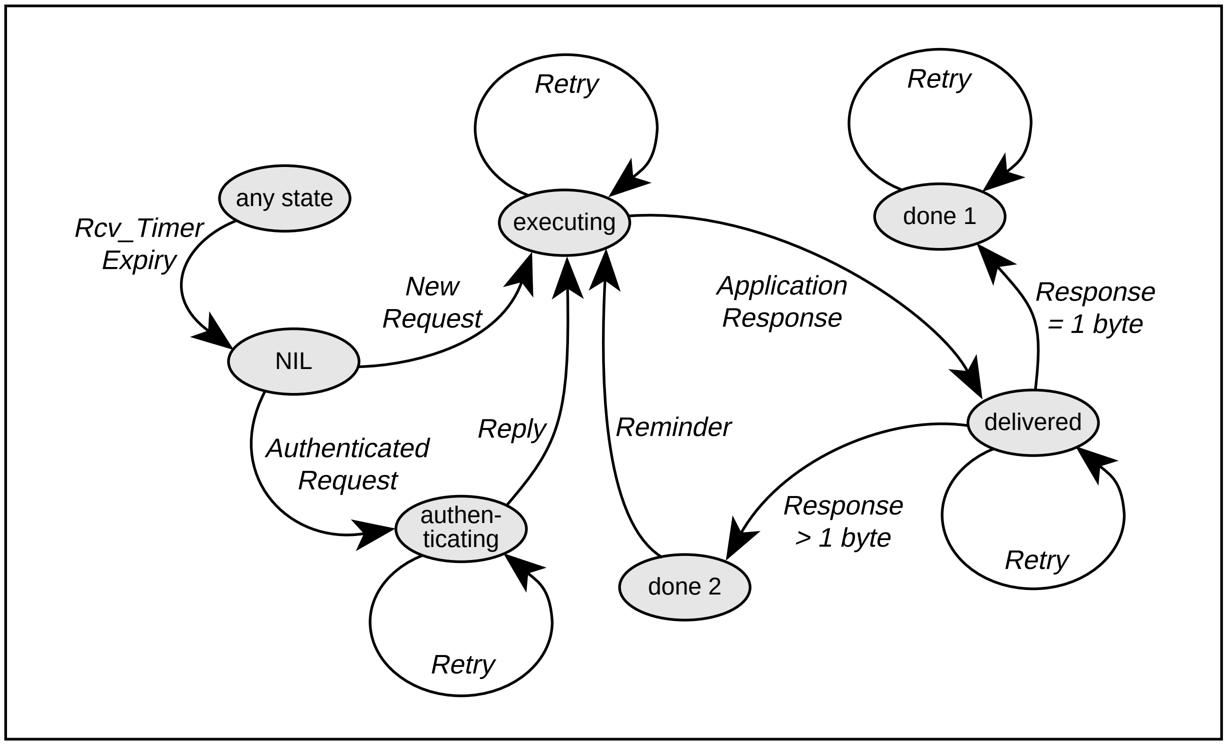


Figure 9.6 Request-Response Protocol—Simplified Server FSM

Algorithm 9.9:

Input:

SPDU_in	Request , Reminder, SPDU for authentication reply from Network layer
Send_Response ()	service request from the Application layer
Rcv_Timer Expiration	timeout of the Rcv_Timer

Outputs:

Response SPDU	to the remote Session entity
Rcv_Request ()	indication to the Application layer

State Variables:

RR pool	pool of Receive Records

```
begin { algorithm 9.9 }

case event of
    SPDU_in:
        Process_SPDU;

    Send_Response (TID, APDU, priority):
        begin
        if (APDU.DataLength = 1) then begin
            RR.L5_state := done1;
            RR.Reply_type := type;
            RR.Reply := APDU.Data;
            end;
        else begin
            RR.L5_state := done2;
            Compose and send response SPDU;
            end;
        end;

    Rcv_Timer Expiration:
        begin
        case RR.L5_state of
            nil, done1, done2:
                null;
            executing, authenticating:
                {lock up receive record and wait for response};
            end case;
        De-allocate receive record RR;
        De-allocate authentication record if Authrcd.TID = RR.
        end;
    end case;

Procedure Process_SPDU (SPDU_in);
    begin
    Retrieve the associated RR (RR = nil if none exists );
    if (RR <> nil) then begin
        Compare (RR.Trans_No, SPDU_in.Trans_No) -> (result);
        if (result <> duplicate) then
            Reset Rcv_Timer;        else if (RR.L5_state = authenticated )then
                if (RR.checksum <> C(SPDU_in.APDU)) then begin
                RR.Authenticated := False;
                end;
        end;
    else begin
        allocate and initialize a RR by assigning:
        RR.Authenticated := False;
        RR.Source := SPDU_in.src_addr;
        RR.Trans_No := SPDU_in.Trans_No;
        RR.Priority := priority;
        RR.Destination := {initialize according to addressing mode in SPDU_in };
        RR.L5_state := nil;
        Start Rcv_Timer;
        if (SPDU_in.Auth = True) then begin
            RR.L5_state := authenticating;
            Initiate_Challenge(SPDU_in,RR.Trans_no);
            end;
        end;
    end;

    case SPDU_in.SPDUtype of
        REQ:
            Process_Request (RR, SPDU_in);
        REQ/REM: begin
            Split SPDU_in into two PDUs: spdu1 (the REM part) and spdu2 (the REQ part);
            Process_Reminder (RR, spdu1);
            if spdu2 <> nil then Process_Request (RR, spdu2);
            end;
        AuthREPLY:
            RR.L5_state := Process_Reply(RR.Trans_no,SPDU_in);
        end case;
end procedure;

Procedure Process_Request (RR, SPDU);
    begin
case RR.L5_state of
            nil:
            begin   { plain REQ SPDU };
                if (SPDU.Auth = True) then
                    Initiate_Challenge(RR.SPDU);
                else begin
                    provide Rcv_Request () indication. to the Application. layer;
                    RR.L5_state := executing;
                    end;
            end;
        executing, done:
            null;
        done1:
            compose and send RESP SPDU;
        done2:
                begin   { plain Request }
                if {authenticated transaction }then
                    if (C(SPDU.APDU) <> RR.Checksum) then
                        RR.Authenticated := False;
                provide Rcv_Request () indication. to the Application. layer;
                RR.L5_state := executing;
                end;
            end
        end case;
    end procedure;

Procedure Process_Reminder (RR, SPDU);
    int temp;

    {identify my member # in the destination group};
    if (SPDU.Length <> 0) then begin
        temp = my_member_no / 8+ 1; {integer division with truncation };
        if (temp ≤ SPDU.Length) then
            if (SPDU.M_List [my_member_no] = 1) then      { my response received by the requester}
                RR.L5_state := done;
        end;
    end procedure;

end { algorithm 9.9 };
```

9.10 Request-Response Protocol Timers

The two timers—Xmit_Timer and Rcv_timer—used by the Request-Response protocol follow the function and the form of Transport timers. The recommended values are identical to those for the transport layer as defined in section 8.11, with the exception that if the request takes a significant amount of processing time on the server (relative to the transaction time), that time should be included in the calculations.

9.11 Authentication Protocol

The Authentication server is accessible by the session layer and the transport layer. It relies on the duplicate detection mechanism in the transaction control sublayer, and no other transport layer services. Authentication allows a server to verify the identity of the requester. Use of this service is controlled by network management commands, which specify the messages/network variable exchanges to be authenticated.

The Authentication protocol has two asymmetric parts: the challenger and the challengee. The authentication process is initiated by the challenger, which generates a random number X; next, the challengee responds with Y = E(X, msg), an encryption of X and the original message using a private key; and finally the challenger compares Y with its own version of E(X, msg), and makes a pass/fail decision based on the outcome of the comparison. The Authentication algorithm described below defines both the challenger and the challengee functions. All the server calls are synchronous.

Algorithm 9.11:

```
Inputs and Outputs:
    Initiate_Challenge(RR,PDU)            challenger server call
    Reply(XR,PDU)                         challengee server call sends reply to challenger
    Process_Reply(RR,PDU) ->(result)      challenger server call—result is pass or fail

State Variables:
    Authen_rcd                            a record used for the random number and client's APDU
    RR                                    Receive record
    XR                                    Transmit record

begin { algorithm 9.11 };

    case event of

    Initiate_Challenge(RR, PDUptr) :
        begin
            {get receive transaction record (RR) from TID, PDUptr}:
            if (Authen_rcd.TID = nil )then begin
                Authen_rcd.TID := TID;            {reserve authentication record }
                RR.state := authenticating;
                Authen_rcd.ptr :=PDUptr;
                Authen_rcd.rand := rand();          { generate 8 byte random number }
                RR.checksum := C(PDUptr);     { encryption of initiating APDU algorithm 9.13}
                compose AuthPDU challenge using 8 byte random number and issue Send_Packet()
            else if (Authen_rcd.TID=TID) then {retry the challenge }
                compose AuthPDU challenge using 8 byte random number and issue Send_Packet()
            else
                RR.state := waiting;                { could not allocate authentication record }
        end;

    Reply (XR,Challenge_PDU) :
        begin
            if Validate(Challenge_PDU.priority, TID) = current then begin
                compute E(Challenge_PDU.RandomBytes, XR.TPDU_ptr.APDU) {algorithm 9.12}
                compose Auth PDU reply using result of E() and issue Send_Packet()
            end;
        end;

    Process_Reply (RR, PDU) -> (result):
        begin
            if Validate(PDU.priority, RR.Trans_no) = current then begin
                test := E(Authen_rcd.rand, Authen_rcd.ptr.APDU);
                if (test = PDU.CryptoBytes )then begin
                    result := pass;
                    RR.authenticated := True; {notification to application layer }
                    end;
                else begin
                    result := fail;
                    RR.authenticated := False;
                    end;
            end;
        end;
    end case;
end { algorithm 9.11 } ;
```

9.12 Encryption Algorithm

The LonTalk encryption algorithm facilitates one way encoding rather than real encryption. It uses a 48-bit encryption key K, a variable length APDU, A[len], and a 64-bit input string R to produce a 64-bit output string Y. Desirable properties of the random number R are defined in 9.14. Any 48-bit number is a valid encryption key.

The encryption function is not published in this version of the specification. Echelon has obtained expert advice on one way encryption functions. The advice is that it is impossible to prove beyond any doubt that a function has no inverse. Those who have seen the function as of June, 1994 believe it has no inverse, but Echelon has been advised that it is more secure if it is not published. Nevertheless, Echelon has, and shall continue to make the function available on a need to know basis provided that there is written agreement to keep the function confidential.

Algorithm 9.12:

```
Input:
        R: array [0..7] of (0..255);        input string ("plain text")
        A: array [1..240] of (0..255); APDU

Output:
        E: array [0..7] of (0..255);        E = E (R,A) ("cipher text")

State  variables:
        K: array [0..5] of (0..255);        encryption key


begin { algorithm 9.12 }:

    Procedure E (R,A);
        begin

    end { algorithm 9.12 };
```

9.13 Retries and the Role of the Checksum Function

The checksum function is used for validating APDUs in client retries. The client shall retry if any of the original message, the challenge, the reply, or the acknowledgment/response is lost. Upon receiving a retry, the action taken by the server is a function of the transaction state as follows:

waiting	the server is waiting for the authentication record. In this case, the server shall attempt to allocate the record again;
authenticating	the server has issued a challenge and is waiting for a reply. In this case, the server simply reissues the same challenge (with the same random number);
authenticated	the authentication exchange has completed, with successful verification. If the original message was acknowledged, then the acknowledgment is reissued and the retry is discarded. If the original message was a request of an unknown type, then it is assumed that the application is still composing the response, so the retry is discarded. If the original message was a non-idempotent request, the response is reissued and the retry is discarded. If the original message was an idempotent request, then the retry APDU is en-crypted using the checksum function C(). The result

	is compared with that saved from the original message. If they do not match, the retry is marked as not authenticated. In any case, the retry is delivered to the application;
not authenticated	the authentication exchange has completed without successful verification. The action taken is much like that for the “authenticated” state, except no encryption/comparison is done for idempotent requests.

Algorithm 9.13:

```
Procedure C(PDU) ->C[3];        {yields a 3 byte checksum}
    var R[8]; { 8 byte work space}
    begin {algorithm 9.13}
        R := E(K,PDU.APDU);  { K is the 6 byte encryption key ; E() is algorithm 9.12}
        C[0] := R[0];
        C[1] := R[4];
        C[2] := R[7];
    end; {algorithm 9.13}
```

9.14 Random Number Generation

The random number generator used by the authentication protocol has the following properties:

(i) R, the number generated need not be mathematically random but it must be truly unpredictable; this means that the generator period should be as long as possible;

(ii) The generator does not generate predictable values after events such as power failure or rebooting.

9.15 Using Authentication

The LonTalk authentication scheme must be correctly used to provide maximum security. One problem of which the user should be aware is the transportation of authentication keys in the open using a network management command. This problem can be overcome by using the increment authentication key network management command rather than the network management command which provides an absolute value for the key.

10. PRESENTATION/APPLICATION LAYER

10.1 Assumptions

The Application protocol makes no assumptions apart from relying on the Transaction Control sublayer for correct TPDU/SPDU sequencing and duplicate detection. The provided functions of the presentation layer are specified as a part of the APDU header. In particular, when the APDU header indicates that the APDU is a network variable update, the header has presenation information encoded within it because it tells the node how to interpret the APDU data.

10.2 Service Provided

The Presentation/Application layer provides 5 services:

- Network Variable Propagation. This service sends messages which are interpreted by the receiver(s) as a network variable updates. If the receivers are using the LONWORKS application programming model, these network variable updates are handled for the application in a special way. See the Neuron C Programmer's Guide; A special two byte header is used to convey the presentation layer information that the APDU is to be interpreted as a network variable.

- Generic Message Passing. An application may construct an arbitrary message, addressed using any of the addressing modes;

- Network Management Messages. These messages are described in detail in chapter 11;

- Network Diagnostic Messages. These messages are typically initiated by a network management tool to test which nodes are fully operational, and to take corrective action around problem areas. These messages are described in detail in chapter 11;

- Foreign Frame Transmission. These messages originate external to the LonTalk environment, and are destined for nodes also external to the LonTalk environment. This function is provided as a means to use the LonTalk protocol as a gateway between two such external nodes, or to tunnel other protocols through the LonTalk protocol.

10.3 Service Interface

For the purpose of this protocol specification it is assumed that the service interface to the application program has the form shown in figure 10.1.

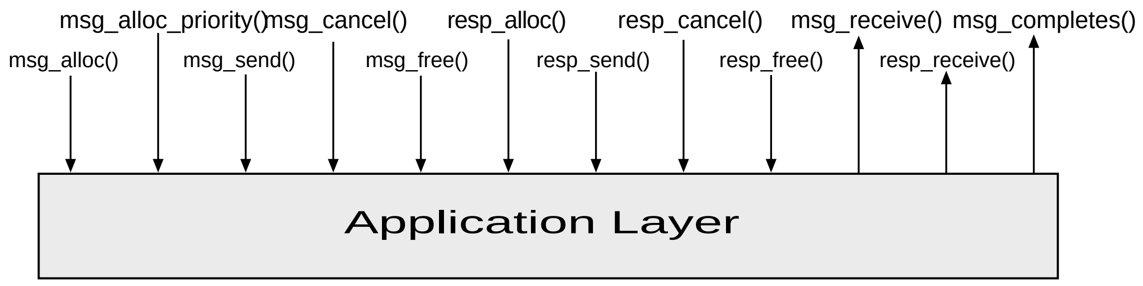


Figure 10.1 Application Layer Interface

The syntax of the service interface primitives is given below. Most of the service primitives require no parameters. Instead they operate on the data structures msg_out, msg_in, resp_out and resp_in.

msg_alloc() -> (true/false)
msg_alloc_priority() -> (true/false)
msg_send(msg_out)
msg_cancel()
msg_free()
resp_alloc() -> (true/false)
resp_send(resp_out)
resp_cancel()
resp_free()
msg_receive(msg_in)
resp_receive(resp_in)
msg_completes() -> (failed, succeeded, completed)

10.4 APDU Types and Formats

The APDU consists of a header followed by the application data. The header is a single byte which is followed by a second byte only if the header specifies that network variable information is to follow. The data structure for the APDU is given below:

```
struct message
{
      int destin_type;
      int data[];
}
```

where data is an open ended array and destin_type is one of the following:

```
00xxxxxx  generic application message (64 codes)
1dxxxxxx  a network variable message; “d” indicates direction: 1 for
          outgoing, 0 for incoming. The remaining code bits are
          combined with the first data byte to form a 14 bit network
          variable selector.
011xxxxx  a network management message (32 codes)
0101xxxx  a diagnostic message (16 codes)
0100xxxx  foreign frame (16 codes)
```

The rest of the APDU is defined with the first byte received as leftmost and the last byte received as rightmost. Any long or quad quantities stored in the APDU are stored with the most significant bit on the left. Arrays are stored with the lowest numbered element on the left. Structure fields are also stored left to right. Every LonTalk node must be able to receive, as a minimum, an APDU of 16 bytes of data plus the destin_type.

The application physical data unit (APDU) has the following format:

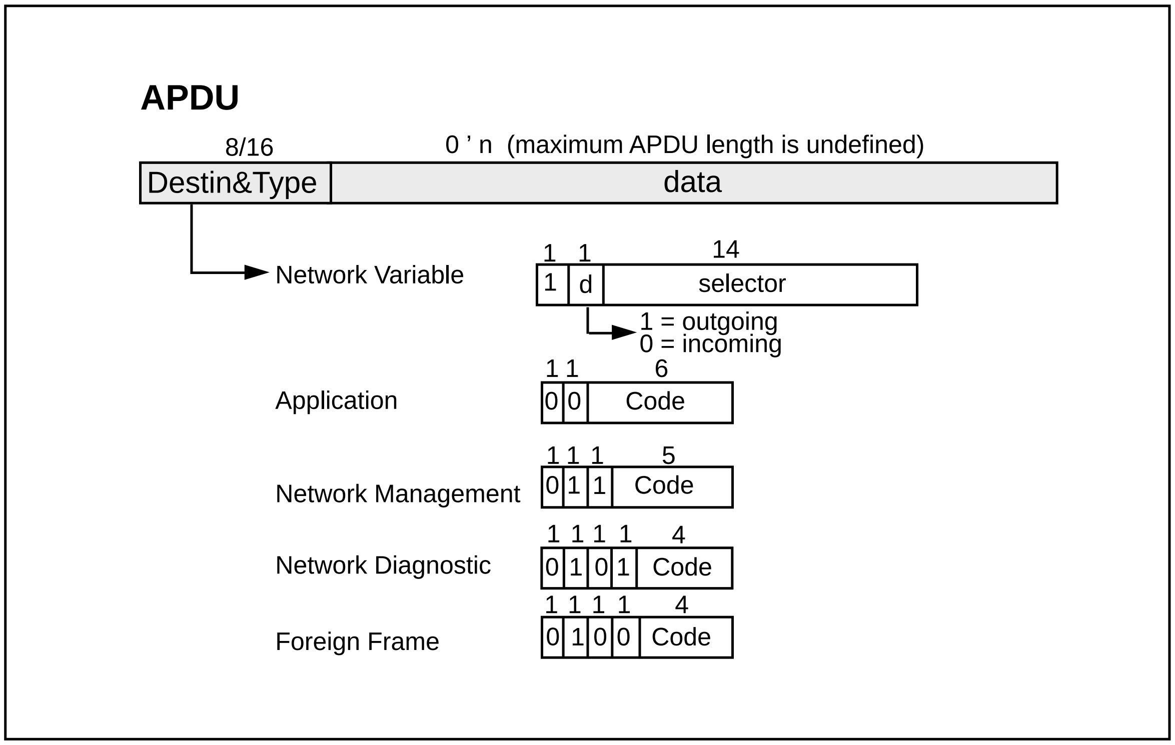


Figure 10.2 LonTalk APDU Format

10.5 Protocol Diagram

The diagram below shows a Non-Idempotent multicast transaction with a loss of both the initial APDU and the ACK TPDU.

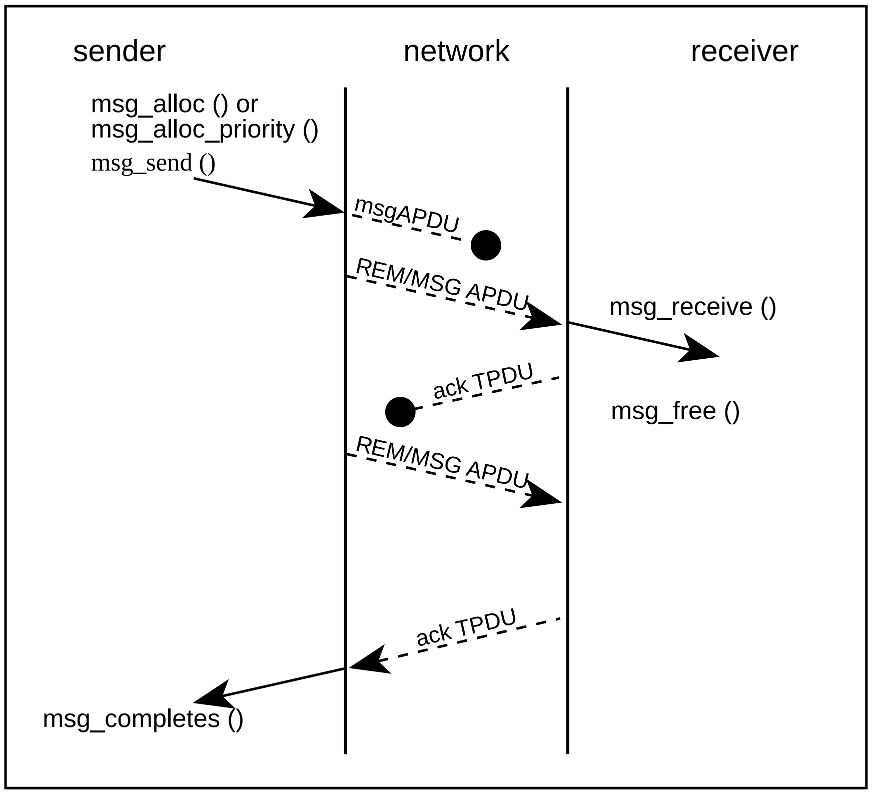


Figure 10.3 Application Protocol Diagram

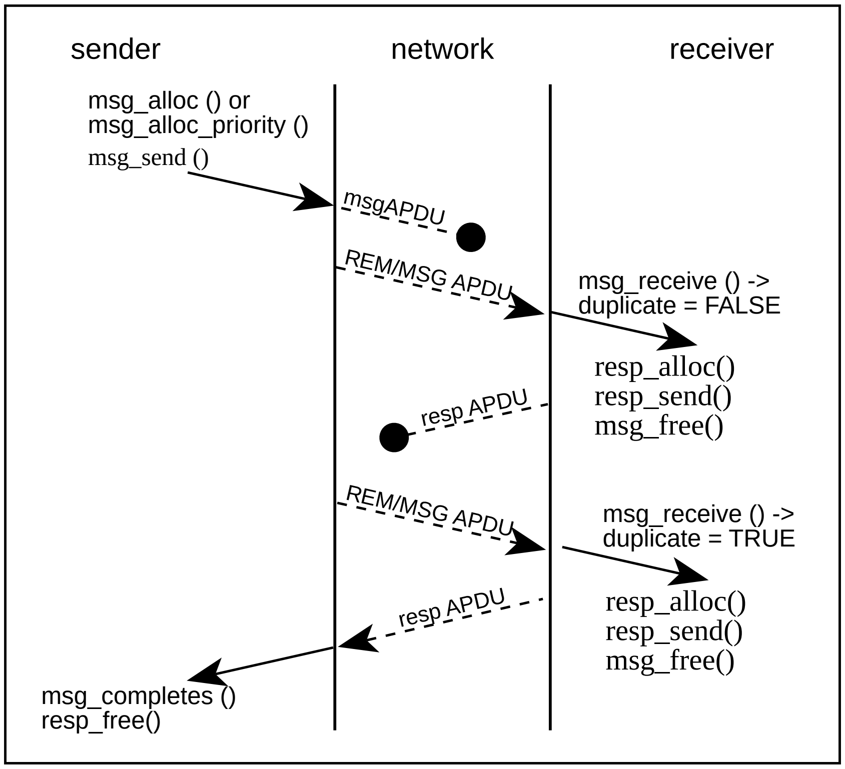


Figure 10.4 Application Protocol Diagram

Figure 10.4 shows an Idempotent Multicast Request/Response transaction with a Loss of Both the Request and Response

10.6 Application Protocol State Variables

The address format data structures are listed below. These address formats are used by the msg_out, msg_in, resp_out, and resp_in structures to direct messages to their destinations.

```
#define Neuron_ID_LEN   6

typedef  enum  addr_type  {  UNBOUND,  SUBNET_NODE,  NEURON_ID,  BROADCAST  }
addr_type;

typedef struct group_struct {
          unsigned    type          : 1;    /* 1 => group                    */
          unsigned    size          : 7;    /* group size (0 => huge group)*/
          unsigned    domain        : 1;    /* domain index                  */
          unsigned    member        : 7;    /* member num                    */
          unsigned    rpt_timer     : 4;    /* unackd_rpt timer              */
          unsigned    retry         : 4;    /* retry count                   */
          unsigned    rcv_timer     : 4;    /* receive timer index           */
          unsigned    tx_timer      : 4;    /* transmit timer index          */
          unsigned    group;                /* group ID                      */
} group_struct;

typedef struct snode_struct {
          addr_type   type;                 /* SUBNET_NODE                   */
          unsigned    domain        : 1;    /* domain index                  */
          unsigned    node          : 7;    /* node number                   */
          unsigned    rpt_timer     : 4;    /* unackd_rpt timer              */
          unsigned    retry         : 4;    /* retry count                   */
          unsigned                  : 4;
          unsigned    tx_timer      : 4;    /* transmit timer index          */
          unsigned    subnet;               /* subnet ID                     */
} snode_struct;

typedef struct bcast_struct {
          addr_type   type;                 /* BROADCAST                     */
          unsigned    domain        : 1;    /* domain index                  */
          unsigned                  : 1;
          unsigned    backlog       : 6;    /* backlog override value        */
          unsigned    rpt_timer     : 4;    /* unackd_rpt timer              */
          unsigned    retry         : 4;    /* retry count                   */
          unsigned                  : 4;
          unsigned    tx_timer      : 4;    /* transmit timer index          */
          unsigned    subnet;               /* subnet ID (0 = domain bcast */
} bcast_struct;


typedef struct nrnid_struct {
          addr_type   type;
          unsigned    domain        : 1;
          unsigned                  : 7;
          unsigned    rpt_timer     : 4;
          unsigned    retry         : 4;
          unsigned                  : 4;
          unsigned    tx_timer      : 4;
          unsigned    subnet;
          unsigned    nid [NEURON_ID_LEN];
} nrnid_struct;


typedef   union msg_out_addr {
          addr_type          no_address;  /* UNBOUND 0 if no address     */
          group_struct       group;       /* Defined above               */
          snode_struct       snode;       /* Defined above               */
          nrnid_struct       nrnid;       /* Defined above               */
          bcast_struct       bcast;       /* Defined above               */
} msg_out_addr;

/* Typedef for 'msg_in_addr', which is the type of the field 'msg_in.addr' */

typedef struct msg_in_addr {
          unsigned   domain         : 1;
          unsigned   flex_domain    : 1;
          unsigned   format         : 6;  /* NOT the 'addr_type' enum.   */
                                          /* INSTEAD: 0 => Bcast, 1 =>   */
                                          /* Group, 2 = > Subnet/Node,   */
                                          /* 3 => Neuron-ID              */
          struct     {
                     unsigned       subnet;
                     unsigned              : 1;
                     unsigned       node   : 7;
          } src_addr;
          union {
                     unsigned bcast_subnet;
                     unsigned group;
                     struct         {
                                    unsigned    subnet;
                                    unsigned
          : 1;
                                    unsigned    node
          : 7;
                     } snode;
                     struct {
                                    unsigned    subnet;
                                    unsigned    nid [NEURON_ID_LEN];
                     } nrnid;
          } dest_addr;
} msg_in_addr;

/* Typedef for 'resp_in_addr', the type of the field 'resp_in.addr'       */


typedef struct resp_in_addr {
          unsigned    domain              : 1;
          unsigned    flex_domain         : 1;
          struct      {
                      unsigned    subnet;
                      unsigned    is_snode : 1; /* 0=>group resp,         */
                                                /* 1=>snode resp          */
                      unsigned    node     : 7;
          } src_addr;
          union {
```

```
                    struct {
                          unsigned   subnet;
                          unsigned          : 1;
                          unsigned   node   : 7;
                    } snode;
                    struct {
                          unsigned   subnet;
                          unsigned          : 1;
                          unsigned   node   : 7;
                          unsigned   group;
                          unsigned          : 2;
                          unsigned   member : 6;
                    } group;
           } dest_addr;
} resp_in_addr;

struct {
       int            code;          /* message code                      */
       int            len;           /* length of message data            */
       int            data[];        /* message data                      */
       boolean        authenticated; /* TRUE if message was authenticated */
       service_type   service;       /* Service type used to send the msg */
       boolean        duplicate;     /* TRUE if message is a duplicate    */
       unsigned       rcvtx;         /* index to the transaction record   */
       msg_in_addr    addr;          /* destination address (see above)   */

       } msg_in;

struct  {
       boolean        priority_on;     /* TRUE if a priority message      */
       msg_tag        tag;             /* to correlate completion codes   */
       int            code;            /* message code                    */
       int            data[];          /* message data                    */
       boolean        authenticated;   /* TRUE if to be authenticated     */
       service_type   service;         /* service type used to send the msg*/
       msg_out_addr   dest_addr;       /* destination address (see above) */

       } msg_out;

struct  {
       int            code;            /* message code                    */
       int            len;             /* message length                  */
       int            data[]'          /* message data                    */
       resp_in_addr   addr;            /* destination address (see above) */

       } resp_in;                      /* struct for receiving responses  */

struct  {
       int            code;            /* message code                    */
       int            data[];          /* message data                    */

       } resp_out;                     /* structure for sending responses */
```

10.7 Interactions Between the Offline State and Request - Response

When using the request/response mechanism either explicitly, or implicitly as with a network variable poll, it is possible to issue a request to a node where the application program is offline, and thus unable to respond. When this condition occurs, what happens depends on whether the response is to a network variable poll or to any other message. If the response is to an application message, the destin_type in the APDU shall be set to 63. When the response is to a foreign frame, the destin_type in the APDU shall be set to 79. When the response is to a network variable poll the response shall contain the network variable selector, but shall not have any associated data. This is also the response received if one attempts to poll a network variable on a node which has no matching selector.

10.8 Error Notification to the Application Program

Regardless of whether the application program is using network variables or sending messages, it always has available to it the status of the last transaction.

10.8.1 Error Notification for Messages

Messages have three events associated with them: completion, success, or failure. Completion means that the transaction has completed (either successfully or not). For acknowledged transactions, success is defined as all acknowledgments having been received. For request/response transactions, success is defined as a response having been received from all of the intended recipients. In this case, it is up to the application to check the response code to see that the intended node was not offline. For unacknowledged transactions, the transaction succeeds when the transaction completes (when the message is sent). For unacknowledged-repeated transactions the transaction completes and succeeds when the message has been sent the requested number of times.

Unacknowledged transactions never fail. Acknowledged transactions provide failure notification to the application when the expected number of acknowledgments are not received. Request-response transactions fail when one or more of the responses are not received.

10.8.2 Error Notification for Network Variables

For network variables, the completion event is posted when the transaction completes. This is identical to messages. Again, as with messages, unacknowledged updates never post a failure event. For acknowledged network variable updates, success is defined as all expected acknowledgments having been received. Failure is then defined as one or more expected acknowledgments not being received. Network variables always use the request-response mechanism when they are polled. Polled network variable updates succeed when all of the values have been returned by the target nodes. A failure event is posted to the application when either: 1) all of the responding targets did not have valid data (no matching network variable or offline), or 2) one or more of the expected responses did not arrive.

11. NETWORK MANAGEMENT AND DIAGNOSTICS

11.1 Assumptions

Network Management and Network Diagnostic (NM/ND) protocols are application level protocols running on top of the Session layer. This means that network management and diagnosis is only possible when the Session layer (and all the underlying layers) are functioning properly.

NM/ND operations interact intimately with the network nodes, and their precise semantics are somewhat node dependent. To facilitate explanation, therefore, reference is made to the Neuron Chip implementation in this section, although it is necessary for non-Neuron Chip hosts to process some of these NM/ND commands when using Neuron Chips as communications chips.

With a few exceptions, all NM/ND commands either examine or modify memory locations in one fashion or another. A portion of the various data elements that reside in EEPROM, such as address assignment, are supported with their own NM/ND commands for reporting and updating, allowing a more controlled execution of these operations within the Neuron Chip. Other areas can be read or written using special addressing modes of the read and write memory commands. Thus, users of move and change types of commands need not concern themselves with the physical layout of the EEPROM. Those needing to download applications must understand the physical layout of EEPROM (although even this information can be wholly contained within a download file).

11.2 Services Provided

The Network Management and Diagnostics application supports the following services:

- Address Assignment: the assignment of all address components (with the exclusion of the Neuron_ID);
- Node Query: the querying of node status and essential statistics;
- Configured router table maintenance.

With a few minor exceptions, network management operations are implemented as remote procedure calls on top of the underlying request-response service provided by layer 5. Sections 11.4 and 11.5 specify the details.

11.3 Network Management and Diagnostics Application Structure

The Network Management application is a distributed application with multiple *clients* and multiple *servers*. Server functions must be supported on all nodes, whereas client functions need only be supported on nodes used as network management devices.

11.4 Node States

A Neuron Chip can be in one of four states. These states are maintained in EEPROM and have the values listed in parenthesis after the state name. These states are reported in the response to the Query Status network diagnostic command (see 11.8.1).

Applicationless: (3) no application yet loaded, application in process of being loaded, or application deemed bad due to application checksum error or signature inconsistency. No application runs in this state. The service LED (a diagnostic aid optionally available in Neuron Chip-based nodes) is on continuously.

Unconfigured: (2) application loaded but configuration either not loaded, being reloaded, or deemed bad due to configuration checksum error. A program can make itself unconfigured by calling the "go_unconfigured()" function. It is determined by the program whether or not an application runs in this state. The service LED flashes at a one second rate.

Hard-offline: (6) application loaded but not running. The configuration is considered valid in this state; the network management authentication bit is honored. The service LED is off.

Configured: (4) normal node state. The application is running and the configuration is considered valid. This is the only state in which messages for the application layer are received. In all other states, they are discarded. The service LED is off.

The configured state has an additional modifier which is the online/offline condition. This condition is <u>not</u> maintained in EEPROM. The states and online/offline condition are controlled via different mechanisms. However, they are reported together in the status command.

Note that there is a subtle distinction between being in the configured or unconfigured states and being *configured* or *unconfigured*. A node in the configured or unconfigured state is as described above. However, a node is referred to as *configured* if it is either in the hard offline or configured states (having valid configuration in either case). A node is referred to as *unconfigured* if it is either in the applicationless or unconfigured states (no valid configuration in either case).

11.5 Using the Network Management Protocol

Most Network Management PDUs (NMPDUs) are conveyed within Session layer requests and/or responses. By default, an NMPDU inherits either the request or the response attribute of the enveloping SPDU. However, some requests, such as Query_ID (), are conveyed within NPDUs rather than SPDUs.

When configured to do so, most NM/ND transactions must be authenticated in order to take effect. Authentication is not possible for messages which are addressed using Neuron ID addressing where the server is not in the same domain as the one that the client used to initiate the request. Commands which do not require authentication to be executed are so noted.

11.5.1 Addressing Considerations

The transmit transaction timer value of the client node must be extended to handle the lengthy delays involved with any command that alters EEPROM. When Neuron ID addressing is used, the server node automatically extends the non-group receive transaction timer to about 8 seconds. This allows this timer to be tuned for normal application traffic without concern for lengthy network management transactions.

The recommended addressing mode for initially using these commands on a node is Neuron_ID. Once the node has been assigned an adequate non-group receive timer value (for duplicate detection) and a domain, subnet, and node field then subnet/node addressing is recommended.

Neuron ID addressed messages are received regardless of the domain in which they are sent. Unconfigured nodes shall also accept any subnet or domain wide broadcast regardless of the domain. In both of these cases, acknowledgments and responses are returned on the domain in which the message was received with a source subnet/node pair of 0/0. Messages received in a domain in which the node is not a member (either because the node is unconfigured or not in the domain) are termed as being received on a flexible domain. Some commands are not permitted under these circumstances and are noted below.

A significant advantage of using Neuron ID addressing for network management commands is that if a node accidentally becomes unconfigured (e.g., due to a checksum error resulting from a power failure while changing configuration), the network management tool does not lose its ability to communicate with the node.

11.5.2 Making Configuration Changes

The paradigm for making configuration changes is as follows:

1. Alter the node state or condition (optional);
2. Perform the change or changes;
3. Update the configuration checksum (only necessary if not done in step 2);
4. Return to step 2 if more needs to be done;
5. Restore the node state or condition if changed in step 1;
6. Reset the node if communication parameter changes were made and it is desired that they take effect.

11.5.3 Downloading An Application Program

The paradigm for downloading applications is as follows:

1. Take the node offline;
2. Alter the node state to applicationless;
3. If node went bypass offline[1] (in step 1), reset the node;
4. Perform a sequence of write memory commands to load the application;
5. Reset the node[2]
6. Compute the application checksum.
7. Enter the unconfigured state.

At this point, the configuration can be loaded. Note that when loading an application followed by loading of the configuration, a node comes up in the offline condition.

11.5.4 Error Handling Conditions

There are several classes of errors worth considering:

Transaction Failures: If a transaction fails (i.e., the desired acknowledgment or response is not received), it is best to attempt to get to a known state rather than simply retry the transaction. If network management authentication is turned on, returning to a known state should include attempting authenticated transactions using different keys (e.g., the current key, the previous key, etc.) until success is achieved.

Node Resets or Power Cycles: if a node resets while a network management command is in progress, the reset will likely manifest itself as either a communication problem or a transaction failure. When EEPROM writes are involved, there is a significant probability that the location being modified at the time of the reset will

[1]Bypass offline is defined by the application checking for "offline" events directly and invoking “offline_confirm” to effect the offline condition. Normally, going offline is handled by the scheduler.

[2]Note that node resets can take quite a while. The slower a node’s input clock, the longer the reset. The more offchip EEPROM or RAM, the longer the reset. Durations up to 18 seconds are possible in the worst case.

become corrupted (most likely with the erase pattern of 0xFF). This adds considerable complexity to the *update key* command, as the authentication key could become corrupted. Thus, in addition to trying the current and previous keys, it may be appropriate to try variants of the key where each single byte is replaced with the erase pattern.

A reset during a *memory refresh* command could result in the corruption of the configuration or program. Either could be catastrophic depending on the scope of knowledge in the network management tool. An option here is to put early power down detection on the network management tool and only issue refresh commands (with no retries) when the power appears stable (assuming the client and server share a common power source).

Read/Write Protect Violations: if a node is read/write protected, attempts to write to the application code area are denied. The client can verify that a write memory attempt failed for this reason by reading the read_write_protect field of the read_only_data structure.

Other adverse effects, such as address table and domain changes, need to be carefully handled and understood by the client. A request can produce a response in a new domain, for example.

Every network node maintains two checksums, one over the configuration and one over the application. Following the completion of any of the network addressing commands that alter the configuration image, a new configuration checksum is calculated and updated. This results in added time to the execution of these commands, and the client should take this into account before sending the next message to the target node. This delay should always take into account the EEPROM write time multiplied by the number of bytes altered. The delay per byte can be as long as 20 ms. Therefore an *update address* command should have a transmit transaction timeout of at least 20*5 + 30, or 130 ms.

Those commands that automatically update checksums are noted.

11.6 Using Router Network Management Commands

The router shall follow the normal Neuron Chip “states” and must be in the Application/Configured state in order to operate fully as a router.

All of the commands that affect the routing tables affect only a single router half. The NM Node Mode command for OFFLINE, ONLINE, and RESTART shall automatically affect BOTH router halves.

For a router, ONLINE means that the router shall operate normally as described. OFFLINE means that the router performs no forwarding; all packets not addressed to the router that appear in the packet buffer circular lists are dropped. Other than the dropping of these packets, an OFFLINE router continues to perform normally.

A router shall ignore a certain group of Network Management commands. These are the broadcast Node Mode commands. This is to prevent a broadcast RESTART or OFFLINE command from stopping the router and preventing the same broadcast command from reaching destinations of the other side of the router. Routers therefore must be RESTARTed or taken OFFLINE individually when desired. A single service pin must exist for the router, the two router halves shall be electrically connected via diodes to a grounding switch. Grounding this point shall send out unique service pin messages on both sides of the router.

11.7 NMPDU Formats and Types

This section lists LonTalk Network Management Protocol Data Units (NMPDUs), using a notation similar to C structure definitions. The bit and byte ordering rules defined in Appendix A apply, with the most significant bit of each byte being transmitted first; the first byte of a record is considered the least significant byte of that record. In the value section of the descriptions, the value corresponds to a command number or a response code for that message.

The first byte of all NM message APDUs contains the Destination/Type data which, for NM requests and commands, is always (binary):

011xxxxx

The <xxxxx> field contains the command code.

Responses that have been generated by the execution of these NM commands are directed to the Application, as specified by the first byte of the APDU:

00pxxxxx

The <p> field is set to one if the operation succeeded, or zero if it failed. Failures are usually due to range errors (table boundaries) or EEPROM write failures. The <xxxxx> field echoes the original NM command code.

The first byte of all ND message APDUs contains the Destination/Type data of:

0101xxxx

The <xxxx> field contains the command code.

ND responses have the following format, where <p> is the same as in NM responses and <xxxx> mirrors the original command:

00p1xxxx

The implications of this are that all NM/ND requests are delivered to the NM/ND layer in the LonTalk protocol, while all NM/ND responses are delivered to the Application Layer. It is assumed that the responses are to be processed at the Application Layer.

In this document, only the command field value is described. The <p> field and the destination code are not included but are assumed to be in place.

Single byte responses are provided for NM operations that are considered non-idempotent.

It is important to note that this document uses a nomenclature of "byte" for 8-bit items and "int" for 16-bit items. This is in contrast to Neuron C, which classifies "int" as an 8-bit item.

11.7.1 Query ID

Query_ID() requests a node, or a set of nodes, to report its Neuron_ID to the requester. Typically this request is addressed on a single domain as a subnet-wide broadcast, implying that the client has knowledge of at least the domain and may be taking orderly probes at subnet addresses in order to interrogate a set of nodes.

To query unconfigured nodes, the selector value within the Query_ID_Request record is set to 0. In order to query nodes whose "respond to query" bit is set (see following command), the selector value is '1'. Either the subnet or domain wide broadcast addressing mode is typically used. This command never requires authentication to be executed.

If supplied, the address and data fields are used as additional qualifiers. The address mode and address field are used to form an address (see "read memory" for a description of this process); "count" bytes (1–11) of data starting at that address are then compared with the supplied data. Only if they match does the query id proceed (as specified by the "selector").

For this command only, read protect is assumed to be always on. If the address and count fall in a read protected area (such as where the authentication keys are stored), no response is returned.

```
struct query_id_request {
      byte command;          /* value = 1 */
      byte selector;         /* 0 = unconfigured nodes
                                1 = respond to query set
                                2 = respond to query set and unconfigured */
      byte address_mode;     /* See "read memory" command */
      byte address_hi;
      byte address_lo;
      byte count;
      byte data;             /* “count” bytes of data */
};

struct query_id_response {
      byte command;          /* value = 1 */
      byte Neuron_ID[6];
      byte program_id_string[8];
};
```

11.7.2 Respond to Query

This command sets or clears the “respond to query” bit in the target node(s). When set, the target shall respond to *query id* requests that have a selector of ‘1’. It shall continue in this mode until the node is reset or its bit is cleared via command. This command is used for network topology interrogation. The “on” version is usually addressed as subnet broadcast, using the unacknowledged-repeated service. The “off” version is addressed to a specific node once it has been interrogated. This command never requires authentication to be executed.

```
struct respond_to_query_cmd {
      byte command;          /* value = 2 */
      byte mode;             /* 1 => ON; 0 => OFF */
};

struct respond_to_query_resp {
      byte command;          /* value = 2 */
};
```

11.7.3 Update Domain

This command updates one of the domain entries in the server. Note that the most significant bit of the node field must be set. Execution of this command updates the configuration checksum. If a node can only be in a single domain, attempts to assign domain index ‘1’ shall return an error. If the domain to be updated is the same as the domain in which the modify message was sent, and the node is in the “configured” state, then the response shall come back on the new domain and thus shall not be received by the sender.

Since the encryption key is propagated in the open, this request should only be used when physical network security can be guaranteed (or security is achieved through other means).

```
struct update_domain_request {
      byte command;          /* value = 3 */
      byte domain_index;     /* 0 or 1 */
      byte domain_id[6];
      byte subnet;
      byte node;              /* msb must be set */
      byte did_length;        /* 0, 1, 3, or 6 */
      byte encrypt_key[6];
};

struct update_domain_response {
      byte command;           /* value = 3 */
};
```

11.7.4 Leave Domain

The node must honor this command even if it still has addresses assigned within this domain. Internally, the node's domain length is set to 0xFF and the subnet, node addresses are set to zero. Also, the authentication key is cleared. The configuration checksum is updated during the execution of this command. If the domain to be left is the domain on which the request was received and the node is *configured*, no response is sent. If the domain being left is the last domain in which the node is configured, the node automatically enters the unconfigured state and re-sets.

```
struct leave_domain_request {
      byte command;           /* value = 4 */
      byte domain_index;      /* 0 or 1 */
};

struct leave_domain_response {
      byte command;           /* value = 4 */
};
```

11.7.5 Update Key

This command is used for updating encryption keys. The domain to be used is specified in the message. The encrypt_key bytes are added to the existing key in a bytewise fashion (no carry). The configuration checksum is updated by this command.

```
struct update_key_request {
      byte command;           /* value = 5 */
      byte domain_index;      /* 0 or 1 */
      byte encrypt_key[6];
};

struct update_key_response {
      byte command;           /* value = 5 */
};
```

11.7.6 Update Address

This command is executed by index. If the address table entry does not exist, the <p> bit in the response shall be zero. The configuration checksum is updated by this command. No cross checking for duplicate addresses or groups is performed.

The form of each address entry in the address table is:

```
struct group_s {
      byte field1;            /*    b7: 1
                                    b0-b6: group size, 0 for huge */
      byte field2;            /*    b7: domain ref
                                    b0-b6: member number, 0 for huge */
      byte field3;            /*    b4-b7: unackd_rpt timer */
                              /*    b0-b3: retry count */
      byte field4;            /*    b4-b7: receive timer index
                                    b0-b3: transmit timer index */
      byte group;             /* group id. */
};

struct snode_s {
      byte type;              /* 1 */
      byte field2;            /*    b7: domain ref set by target
                                    b0-b6: node number */
      byte field3;            /*    b4-b7: unackd_rpt timer */
                              /*    b0-b3: retry count */
      byte field4;            /*    b0-b3: transmit timer index */
      byte subnet;            /* subnet */
};

struct bdcst_s {
      byte type;              /* 3 */
      byte field2;            /*    b7: domain ref set by target */
                              /*    b0-b5: backlog; 0 == unknown */
      byte field3;            /*    b4-b7: unackd_rpt timer */
                              /*    b0-b3: retry count */
      byte field4;            /*    b0-b3: transmit timer index */
      byte subnet;            /* subnet */
};

struct ta_s {                 /* Turnaround entry */
      byte type;              /* 0 */
      byte ta;                /* 1 */
      byte field3;            /*    b4-b7: unackd_rpt timer */
                              /*    b0-b3: retry count */
      byte field4;            /*    b0-b3: transmit timer index */
};

struct empty_s {              /* Empty entry */
      byte type               /* 0 */
      byte empty;             /* 0 */
};
```

```
struct update_addr_request {
      byte command;           /* value = 6 */
      byte index;             /* 0–14 */
      union {
            struct group_s;
            struct snode_s;
            struct bdcst_s;
            struct ta_s;
            struct empty_s;
      } address;
};

struct update_addr_response {
      byte command;           /* value = 6 */
};
```

11.7.7 Query Address

This command reports an entry within the Neuron Chip's address table, given an index.

```
struct query_addr_request {
      byte command;           /* value = 7 */
      byte index;             /* 0–14 */
};

struct query_addr_response {
      byte command;           /* value = 7 */
      union {
            struct group_s;
            struct snode_s;
            struct bdcst_s;
            struct ta_s;
            struct empty_s;
      } address;
};
```

11.7.8 Query Network Variable Configuration

This command reports the entry in the node's nv_config table, by index number; the entry must exist in the table.

```
struct query_nv_cnfg_request {
      byte command;           /* value = 8 */
      byte nv_index;
      [int nv_index16;]       /*    16-bit index iff nv_index == 255 */
                              /*    Host nodes only */
};
```

```
struct query_nv_cnfg_response {
      byte command;           /* value = 8                        */
      byte field1;            /*    b7: priority                  */
                              /*    b6: direction                 */
                              /*    b0-b5: net var selector—msb   */
      byte idlo;              /*    net var selector—lsb          */
      byte field2;            /*    b7: turnaround                */
                              /*    b5-b6: service                */
                              /*    b4: authenticated             */
                              /*    b0-b3: address table index    */
};
```

11.7.9 Update Group Address

This command is used to update a group entry in an address table; it is typically addressed by group. The group size, timer indices, and retry count are updated. The group member field is left unchanged. The entry is updated based on the domain in which the command was received. Therefore, this command is disallowed for flexible domains. This command updates the configuration checksum.

```
struct update_group_addr_request {
      byte command;           /* value = 9 */
      struct group_s;
      } address;
};

struct update_group_addr_response {
      byte command;           /* value = 9 */
};
```

11.7.10 Query Domain

This command is used to retrieve the domain information for one of the two domains in a node. If the second domain is requested and room for only one domain exists, an error is returned.

```
struct query_domain_request {
      byte command;           /* value = 10 */
      byte index;             /* Domain index, 0 or 1 */
};

struct query_domain_response {
      byte command;           /* value = 10 */
      byte domain_id[6];
      byte subnet;
      byte node;
      byte did_length;        /* 0,1,3, or 6 */
      byte encrypt_key[6];
};
```

11.7.11 Update Network Variable Configuration

This command is used to add or modify entries to or from the node's nv_cnfg table, by an index number. There must be free space in the nv_cnfg table for the entry. The address table index may be set to 0–15, where 15 indicates that no address table entry is associated with the network variable. The configuration checksum is updated by this command. For a network variable with index X, a network variable selector with the value 0x3FFF-X implies that the network variable is not in a logical connection (i.e., is not bound).

```
struct update_nv_cnfg_request {
      byte command;           /* value = 11                              */
      byte nv_index;          /*    0-255                                */
      [int nv_index16;]       /*    16-bit index iff nv_index == 255     */
                              /*    Host nodes only                      */
      byte field1;            /*    b7: priority                         */
                              /*    b6: direction                        */
                              /*    b0-b5: net var selector—msb          */
      byte idlo;              /* net var selector—lsb                    */
      byte field2;            /*    b7: turnaround                       */
                              /*    b5-b6: service                       */
                              /*    b4: secure                           */
                              /*    b0-b3: address table index           */
};

struct update_nv_cnfg_response {
      byte command;           /* value = 11 */
};
```

11.7.12 Set Node Mode

This request instructs the application scheduler to enter either the offline or online condition, to reset the entire node via an internal reset, or to change the state of a node. When offline, the application program is halted and NM/ND commands continue to be processed. The online request instructs the application scheduler to leave the offline condition and resume operation of the application. One use of the offline condition is for suspending the application during application EEPROM downloading.

The service type used for this command varies. For online and offline, no response is ever returned so request-response cannot be used. Confirmation of the change in condition is achieved via issuance of a *status request*. For state changes, request-response should be used. Since the state is part of the application, the application checksum is updated. For reset, only the unack'd (or ack'd if authentication is required) service type is used. This message is confirmed with a *status request*. Note that failure to confirm the reset may indicate that the initial unack'd message was lost, necessitating a retry of the exchange.

```
struct set_node_mode_request {
      byte command;           /* value = 12         */
      byte state;             /*    0: offline      */
                              /*    1: online       */
                              /*    2: reset        */
                              /*    3: change state */
      byte state_data;        /* Iff state=3; see “Node States” for values */

struct set_node_mode_response {
      byte command;           /* value = 12   Note, no response is provided*/
                              /*  for offline, online or reset commands    */
};
```

11.7.13 Read Memory

This command is used to read memory. If read / write protect is on, the program can only read the read_only_data, config_data (see access.h), SNVT table, and RAM or EEPROM data areas.

The “count” field contains the number of bytes to be read. This number should not exceed 16 unless the target node has buffers sufficiently large to accommodate the additional data. All images except the Neuron 3120 Chip image ensure that the count is not too large for the buffer space available in the node. Attempts to access too much data result in a failed response.

```
struct read_memory_request {
      byte command;           /* value = 13 */
      byte address_mode;      /* See below */
      byte address_hi;
      byte address_lo;
      byte count;
};

struct read_memory_response {
      byte command;           /* value = 13 */
      byte data[count];
};

/* where “address_mode” determines the physical address as follows:     */
/*    0: address = address_hi*256 + address_lo                          */
/*    1: address = EEPROM read-only address+address_hi*256+address_lo   */
/*    2: address = EEPROM configuration address + address_hi*256 +      */
/*    address_lo                                                        */
```

11.7.14 Write Memory

There are two forms of this command: one form resets the Neuron Chip after writing, the other does not. Confirmation of the reset form must be performed by reading back memory using *read memory*. The non-reset form produces a response and is conveyed via request-response. The reset form is conveyed using unacknowledged or unacknowledged-repeated service. The configuration check-sum is optionally updated. Note that any single write command should not cross the boundary of the EEPROM memory.

If read / write protect is on, only the config_data_struct (see access.h) can be written. The byte count should be limited to 11 bytes unless the target node has buffers that are sufficiently large to handle more. When writing to EEPROM, the byte count should be limited to 38 to avoid watchdog timeouts on the target.

```
struct write_memory_request {
      byte command;           /* value = 14 */
      byte address_mode;      /* See comment in read_memory command above */
      byte address_hi;
      byte address_lo;
      byte count;
      byte form;              /* 0: no reset, no checksum
                                 1: recalculate both checksums
                                 4: recalculate just configuration checksum
                                 8: reset
                                 9: reset, recalc both checksums
                                12: reset, recalc configuration checksum */
      byte data[count];
};

struct write_memory_response { /* used only for non-reset writes */
      byte command;            /* value = 14 */
};
```

11.7.15 Checksum Recalculate

This command forces the Neuron Chip to compute and store a new configuration or application checksum. It should be used at the end of any Network Management sessions that alter the configuration EEPROM image or application EEPROM/RAM image (unless those commands have specifically performed this operation already, as with the address commands).

```
struct checksum_request {
      byte command;           /* value = 15 */
      byte which;             /*    1: do application and configuration
                              /*    4: do configuration only */
};

struct checksum_response {
      byte command;           /* value = 15 */
};
```

11.7.16 Install

This command is used during installation for a variety of purposes. The primary purpose is to perform the wink function. This forces the Neuron Chip to execute a special "when" clause in the application program called the "wink" clause. This clause shall execute even if the node is in an unconfigured state. This way, an installer can signal to a node to do something distinctive (such as blink a light) for physical identification purposes. The install command always performs a wink on non-host[1] nodes (request-response cannot be used for this command with non-host nodes).

For host nodes, the install command has an additional subcommand. If the subcommand is not present, wink is assumed; if it is present, it invokes a command as follows:

0:	wink
1:	send service pin data
2–255:	reserved

If the subcommand is “send service pin data”, the subdata field contains the LONWORKS network interface number. The host node responds with the service pin data for that LONWORKS network interface. The command field of the service_pin_message structure in the response contains ‘0’ if the LONWORKS network interface is functional, and a non zero value otherwise. This command can be used to methodically process each of the LONWORKS network interfaces attached to a host.

```
struct install_msg {
      byte command;           /* value = 16 */
      byte subcommand;        /* see above (host node only) */
      byte subdata;           /* see above (host node only) */
};

struct install_response {
      byte command;           /* value = 16 */
      union {
            struct service_pin_message;   /* if subcommand == 0 */
      } data;
};
```

11.7.17 Memory Refresh

This command causes the Neuron Chip to rewrite the existing contents of EEPROM at a specified address for a specified number of bytes. This can be used to periodically rewrite the contents of on-chip or off-chip EEPROM to extend the retention time of the memory contents. Note that the Neuron ID is write protected and thus cannot be rewritten.

An error is returned if a refresh of off-chip memory is requested and none exists. Also, if “offset” falls beyond the end of the EEPROM area, an error is returned. In this way, the sender of these commands can simply increment the offset until an error is returned. The count should be limited to 8 if the target is online and could be as high as 38 if the target is offline.

[1]A host node is a microprocessor-based node that uses the Neuron Chip as a communication chip only. A host node may have one or more LONWORKS network interfaces (LNIs).

```
struct memory_refresh_request {
      byte command;           /* value = 17 */
      int offset;             /* 16 bit offset from start of EEPROM */
      byte count;             /* Number of bytes to write */
      byte offchip;           /* 1=> refresh off chip memory */
};

struct memory_refresh_response {
      byte command;           /* value = 17 */
};
```

11.7.18 Query Standard Network Variable Type

This command is used to retrieve Standard Network Variable Type (SNVT) data from a host node. An error is returned if the Neuron Chip application program is not configured as a LONWORKS network interface. The requester need know nothing about physical addresses on the host; instead the requester only deals with offsets. The requester starts with an offset of 0 and then bumps the offset for each subsequent request. The byte count should be limited to 16 unless the target node has sufficiently large buffers to handle larger counts.

```
struct query_SNVT_request {
      byte command;           /* value = 18 */
      int offset;             /* 16 bit offset into micro’s SNVT table */
      byte count;             /* number of bytes to return (up to 16) */
};

struct query_SNVT_response {
      byte command;           /* value = 18 */
      byte data[count];       /* return data (count bytes) */
};
```

11.7.19 Network Variable Value Fetch

This message is used to poll network variables. It has two advantages over the network variable poll message: it uses the network variable selector and is thus independent of configuration, and it also obtains the value regardless of the node's online/offline condition.

```
struct nv_fetch_request {
      byte command;           /* value = 19 */
      byte index;             /* Network Variable selector */
      [int index16;]          /* 16-bit index, used only if index == 255 */
                              /* This format supported only by host nodes*/
};

struct nv_fetch_response {
      byte command;           /* value = 19 */
      byte index;             /* Network Variable selector */
      [int index16;]          /* Iff index == 255; host nodes only */
      byte data[NVLEN];       /* return data (based on NV length) */
};
```

11.7.20 Service Pin Message

This message is unlike the other network management messages in that it is an unsolicited message. It is sent over the network from a node when that node's service pin is depressed. The message is sent as a domain-wide broadcast on domain length 0 with a source subnet 0 and node address of 0.

```
struct service_pin_message {
      byte command;     /* value = 31 */
      byte neuron_id[6];
      byte program_id_string[8];
      };
```

11.7.21 Network Management Escape Code

One of the network management command codes is reserved as a escape. The value of the escape code is 0x7D. Sending the escape code as the network management command causes the first two bytes of the APDU to be interpreted as additional command codes. This capability allows the network management protocol to be extended in product specific ways. For example, this mechanism is used in the serial LonTalk Adapter and the PC LonTalk adapter products to query product specific information and to configure application specific information.

If a node responds to network management messages which use the network management escape code, then that node shall always respond to the Product Query command. All other commands are product specific and are documented with the products. The response to this command has two forms. The short form contains only a single byte to specify the product. The complete form contains the response code, the product byte (as in the short form), a two byte field for the model number, a single byte for the firmware version, a byte for the configuration of the device and a byte for the transceiver type. In the complete response form, the value of the configuration byte returned is zero unless the device can be put into several modes or configurations. In this case, the byte contains the current mode or configuration of the device.

Success or failure is reported on the escape code rather than on the subcode or command.

```
struct product_query_request {
      byte command;           /* Destination: NM, code: 0x7D (escape)   */
      byte data[2];           /* Product query subcode = 0x01           */
                              /* Product query command = 0x01           */
};
```

```
struct product_query_short_response {
      byte command;     /* success = 0x3D, failure = 0x1D         */
      byte product_ID;  /* product identifier                     */
}

struct product_query_complete_response {
      byte response;          /* success = 0x3D, failure = 0x1D         */
      byte product;           /* product identifier                     */
      byte model[2];          /* model number                           */
      byte version;           /* version number                         */
      byte config;            /* current configuration or mode of device*/
      byte Xcvr_ID;           /* transceiver type                       */
}
```

11.7.22 Router Mode

This request instructs the router to perform one of several router related tasks. The "resume" command returns the router from the "all flood" state. The "init router tables" command copies all routing tables from EEPROM into the RAM tables (if a configured router) or sets all RAM tables to flood (if a learning router); this is the same action that occurs after node reset. The "mode all flood" command causes the router to forward all packets in the domain. The Router Mode command affects both router halves, and is conveyed via the Request-Response protocol. Note that the normal Network Management Node Mode request may be used to take the entire router offline and online.

```
struct router_mode_request {
      byte command;           /* Destination: NM, code: 20 */
      byte mode;              /*    0: resume, 1: init subnet table,
                                    2: mode flood */
};

struct router_mode_response {
      byte command;           /* Destination: APPL, code: 20 */
}
```

11.7.23 Router Clear Group or Subnet Table

This request is used to clear all entries in either the group or subnet routing table for a single domain for a *single router half*. The command is segmented to cover 8-byte sections in order to prevent lengthy EEPROM write operations. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated.

```
struct router_table_clear_request {
      byte command;           /* Destination: NM, code: 21  */
      byte field1;            /* b7: 1 = group, 0 = subnet  */
                              /* b6: domain ref             */
                              /* b0-3: 8x index             */
}
```

```
struct router_group_clear_response {
      byte command;           /* Destination: APPL, code: 21 */
}
```

11.7.24 Router Group or Subnet Table Download

This request is used to configure the entire group or subnet table in EEPROM for the specified domain for a single router half. The download function is broken into 8-byte sections. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated.

```
struct groupsubnet_table_download {
      byte command;           /* Destination: NM, code: 22  */
      byte field1;            /* b7: 1 = group, 0 = subnet  */
                              /* b6: domain ref             */
                              /* b0-3: 8x index             */
      byte table[8];          /* l.s. bit is l.s. group/subnet # */
}

struct groupsubnet_table_download_response {
      byte command;           /* Destination: APPL, code: 22 */
}
```

11.7.25 Router Group Forward

This request sets the forwarding flag in the routing table for a given group in the specified domain. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated if changed.

```
struct group_forward_request {
      byte command;           /* Destination: NM, code: 23 */
      byte field1;            /* b0: 0: RAM only, 1: RAM + EEPROM */
                              /* b6: domain ref             */
      byte group;             /* 0-255 */
}

struct group_forward_response {
      byte command;           /* Destination: APPL, code: 23 */
}
```

11.7.26 Router Subnet Forward

This request sets the forwarding flag in the routing table for a given subnet in the specified domain. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated if changed.

```
struct subnet_forward_request {
      byte command;           /* Destination: NM, code: 24 */
      byte field1;            /* b0: 0: RAM only, 1: RAM + EEPROM */
                              /* b6: domain ref             */
      byte subnet;            /* 1-255 */
}

struct subnet_forward_response {
      byte command;           /* Destination: APPL, code: 24 */
}
```

11.7.27 Router Do Not Forward Group

This request clears the forwarding flag in the routing table for a given group in the specified domain. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated if changed.

```
struct group_noforward_request {
      byte command;           /* Destination: NM, code: 25 */
      byte field1;            /* b0: 0: RAM only, 1: RAM + EEPROM */
                              /* b6: domain ref             */
      byte group;             /* 0-255 */
}

struct group_noforward_response {
      byte command;           /* Destination: APPL, code: 25 */
}
```

11.7.28 Router Do Not Forward Subnet

This request clears the forwarding flag in the routing table for a given subnet in the specified domain. This command is conveyed via the Request-Response protocol. The configuration checksum in EEPROM is updated if changed.

```
struct subnet_noforward_request {
      byte command;           /* Destination: NM, code: 26 */
      byte field1;            /* b0: 0: RAM only, 1: RAM + EEPROM */
                              /* b6: domain ref             */
      byte subnet;            /* 1-255 */
}

struct subnet_noforward_response {
      byte command;           /* Destination: APPL, code: 26 */
}
```

11.7.29 Router Group or Subnet Table Report

This request is used to report the current settings of either group or subnet tables in EEPROM or RAM for the specified domain for a single router half. The report function is broken into 8-byte sections. This command is conveyed via the Request-Response protocol.

```
struct groupsubnet_table_report_request {
      byte command;           /* Destination: NM, code: 27  */
      byte field1;            /* b7: 1 = group, 0 = subnet  */
                              /* b6: domain ref             */
                              /* b0-3: 8x index             */
}

struct groupsubnet_table_report_response {
      byte command;           /* Destination: APPL, code: 27      */
      byte table[8];          /* l.s. bit is l.s. group/subnet #  */
}
```

11.7.30 Router Status

This request is used to report the router configuration and flood/normal modes. It is conveyed via the Request-Response protocol.

```
struct router_status_request {
      byte command;           /* Destination: NM, code: 28  */
}

struct router_status_response {
      byte command;           /* Destination: APPL, code: 28        */
      byte router_cnfg;       /* type: 1 = learning, 0 = configured */
                              /* 2 = bridge, 3 = bridge_repeater    */
      byte mode;              /* 0 = normal, 2 = flood              */
}
```

11.7.31 Router Half Escape Code

Although this is not in itself a network management command, it is included in this section for completeness. When this code is placed at the start of the APDU and is followed by any Network Management or Network Diagnostic command, that command shall be passed over to the *other* router half for processing. Any responses shall be returned in the normal manner.

```
      byte command;           /* Destination: NM, code: 30 */
```

11.8 DPDU Types and Formats

Most Diagnostic PDUs (DPDUs) are conveyed within Session layer Requests and/or Responses. By default, a DPDU then inherits either the request or the response attribute of the enveloping SPDU.

This section lists LonTalk Diagnostic Protocol Data Units. The bit and byte ordering rules defined in Appendix A apply, with the most significant bit of each byte being transmitted first; the first byte of a record is considered the least significant byte of that record. In the value section of the descriptions, the value corresponds to a command number or a response code for that message. In addition, the string "+ pass/fail" means that a single bit flag is set in the high order bit of the response to indicate that the command was either successful or that it failed.

11.8.1 Query Status

This command gives a snapshot of a node's health. It conveys error statistics, reset information, the node state, the error log, the system image version, and the Neuron model number. The error statistics, reset cause, and error log can all be cleared via the "clear status" command. Note that the statistics are also cleared whenever the node resets. This command never requires authentication to be executed.

```
struct query_status_request {
      byte command;                 /* value = 1 */
};

struct query_status_response {
      byte command;                 /* value = 1 */
      struct status_response data;  /* see below */
};

struct status_response {
      int transmission_errors;
      int transaction_timeouts;
      int receive_transaction_full;
      int lost_messages;
      int missed_messages;
      byte reset_cause;
      byte node_state;
      byte version;
      byte error_log;
      byte model_number;
};
```

These fields are defined as follows:

transmission_errors: This is a count of the number of transmission errors that have occurred on the network. A transmission error is detected via a CRC error during packet reception. This could result from a collision, a noisy medium, signal attenuation, etc.;

transaction_timeouts: This is a count of the number of timeouts that have occurred in attempting to carry out acknowledged or request/response transactions. A time-out occurs when a node fails to receive all the expected acknowledgments or responses after retrying the configured number of times at the configured interval;

receive_transaction_full: This counter reflects the number of times an incoming unackd_rpt, ackd or request message was lost because there was no more room in the receive transaction database. The size of this database is configured via a compile time pragma;

lost_messages: This is the number of messages that were addressed to the node which were thrown away because there was no application buffer available for the message. The number of application buffers is configured via a compile time pragma;

missed_messages: This is the number of messages that were on the network but could not be received because there was no network buffer (packet buffer) available for the message or the network buffer was too small to receive the message. The number and size of network buffers are configured via a compile time pragma;

reset_cause: This byte contains the reset cause information. This identifies the source of the most recent reset. The values for this byte are as follows (X => don’t care):

```
Power-up reset             0bXXXXXXX1
External reset             0bXXXXXX10
WDT reset                  0bXXXX1100
Software-initiated reset   0bXXX10100
```

node_state: This contains both the node state and node condition (as defined in the “Node State” section in the network management section). Values are as follows:

```
Unconfigured               0x02
Unconfig/App-less          0x03
Configured/online          0x04
Configured/hard-offline    0x06  /* Permanent offline          */
Configured/offline         0x0C  /* Non-reset retained offline */
                                 /* (note this is actually an  */
                                 /* encoding of the node state */
                                 /* of “configured” and the    */
                                 /* offline condition)         */
Configured/bypass          0x8C  /* Non-reset retained bypass- */
                                 /* mode offline Like config-  */
                                 /* ured/offline except that   */
                                 /* the application went off-  */
                                 /* line in bypass mode        */
```

version: The version number reflects the ROM version and may be used by a network management tool for computing addresses to EEPROM data fields not supported by the standard NM address assignment/reporting commands. It is also needed by the linker for resolving references to system functions in the application. The version number is 1 to 127 for system images and 128 to 255 for custom images. 255 is a special escape version that means more version number information is available via other commands (this mechanism is not currently implemented);

error_log: The error log contains the most recent error logged by the system. A value of 0 indicates no error. An error in the range 1..127 is an application error and unique to the application. An error in the range 128..255 is a system error. These errors are documented in the *“Neuron C Programmer’s Guide;”*

model_number: The model number is the Neuron Chip model (e.g., 3150, 3120, etc.). The codes for this are as follows:

8:	3120
0:	3150

The model and version numbers together can be used to determine the exact firmware image in use by a node.

11.8.2 Proxy Status

This command can be used to deliver a command to one or more target nodes via an agent node. The proxy command is sent to an agent along with a target address in the APDU. The agent node relays the command to the target and then relays the response back to the original requester. The proxy command can only be used to relay a status request or a query id (*unconfigured*) request. If the original request is a priority request, it is relayed as a priority request. Although this command never itself requires authentication to be executed, if the original request is marked to be authenticated, the relayed request to the target shall also be so marked.

The original requester specifies the target address via data in the form of an address table entry in the APDU. The agent node uses this to determine the destination address and the retry/timeout values used during the transaction. There is one exception—the domain bit is ignored (the message is always relayed in the same domain as which it was received). Note that the retry/timeout values supplied in the target address should result in a shorter transaction duration than those used by the original requester. In general, this command works best if the agent and target are on the same channel.

```
struct proxy_command {
      byte command;           /* value = 2 */
      byte sub_command;       /* 0=> query id (unconfigured); 1=> status
                                 2=> transceiver status request */
      union {
            struct group_s;
            struct snode_s;
            struct bdcst_s;
            struct nid_s;
      } address;              /* Address structures as defined in the network
                               * management modify address command and below
*/
};

struct nid_s {
      byte type;              /*    2 */
      byte field2;            /*    b7: domain ref set by target */
      byte field3;            /*    b0-b3: retry count */
      byte field4;            /*    b0-b3: transmit timer index */
      byte subnet;            /*    subnet */
      byte nid[6];            /*    Neuron ID */
};

struct proxy_response {
      byte command;           /* value = 2 */
      byte resp_data;         /* data and length are functions of the
sub_command */
```

The response data received by the original requester shall be identical to that which would have been received as a result of a direct request to the target. Note that in the case of a query id (*unconfigured*) broadcast, a direct message could result in multiple responses, whereas a proxy command sent to a single agent with a broadcast target would result in only a single response to the original requester.

This command is disallowed if the agent receives the request on a flexible domain.

Finally, if the agent node is in the process of sending outgoing transactions, it may not be able to deliver the relayed request immediately. Depending on how long this is delayed, the response may not be received by the original requester in time, even after several retries. Therefore, a transaction failure for a proxy command is more likely than for other transactions; this should be taken into account when drawing conclusions from same. Also, in order for the proxy command to work, the agent node must have at least two application input buffers.

11.8.3 Clear Status

This command clears a subset of the information in the status response. This includes the statistics information, the reset cause register and the error log. A controller that does a status request on a periodic basis may choose to use a clear command following each successful status response.

```
struct clear_status_command {
      byte command;           /* value = 3 */
};

struct clear_status_response {
      byte command;           /* value = 3 */
};
```

11.8.4 Query Transceiver Status

This command retrieves the status register information from a transceiver. It fails if there is no transceiver on the node, or if communication with the transceiver fails. It returns seven registers' worth of data regardless of the number of registers that the transceiver actually supports. It is up to the controller to know how many registers are valid.

```
struct query_xcvr_status_command {
      byte command;           /* value = 4 */
};
```

```
struct query_xcvr_status_response {
      byte command;           /* value = 4*/
      byte data[7];           /* register values */
};
```

12. BEHAVIORAL CHARACTERISTICS

12.1 Channel Capacity and Throughput

In defining the key performance parameters, the following notations are used:

bps	raw channel speed in bits/sec
Beta	randomizing slot size (bits)
w	size of the MAC randomizing window: 16 slots
$D_{mean}= w/2$	mean busy channel interpacket spacing: 8 slots
$p=(1/2w+p_e)$	probability of packet loss from collisions and transmission errors
p_e	probability of loss due to transmission error
AvgPktSize	length of the packet including preamble
C_{cost}	collision cost (2 packets)

Assuming zero collisions and zero transmission errors, the number of LonTalk frames that can be transmitted while using randomizing window $w = 2*D_{mean}$ is as follows:

$$\textbf{frames} = \textbf{bps} / (\textbf{Beta2}*(D_{mean}+ \#\textbf{pri slots}) + \textbf{Beta1} + \textbf{AvgPktSize}) \; [\textbf{pkts/sec}]$$

Some frames are lost because of collisions and some because of transmission errors. The packet bandwidth *Net_L3* available to the L3 Network Service is then

$$\textbf{Net\_L3} = \textbf{frames} * (1- p) \, [\textbf{pkts/sec}]$$

When packets are lost due to collisions, the L4 and L5 protocols must retransmit, thus further reducing the effective bandwidth. The *net bandwidth available to the L4 and L5 protocols* when the probability of packet loss is p, is Net_L4

$$\textbf{Net\_L4} = \textbf{Net\_L3} * (1-C_{cost}*p)[\textbf{TPDUs/sec}]$$

Net_L4 is maximized by choosing the proper value of w. This “net bandwidth” defines the maximum *Transaction Rate*. With the exception of authenticated transactions, a LonTalk transaction within a group of N requires N PDUs to be transmitted. Assuming that all groups have the same size, this rate is

$$\textbf{Trans\_Rate} = \textbf{Net\_L4} / \textbf{group\_size} \; [\textbf{trans/sec}]$$

Table 12.1 shows the key throughput parameters, using the following assumptions: error free channel, correct backlog estimation, physical NPDU length of 120 bits, all Neuron Chips running at 10 MHz clock rate, no priority slots configured on the channel. The number of transactions per second assumes acknowledged service. The 120 bit packet length assumes a domain id length of 1 byte, and average user data size of 3–4 bytes.

		Channel Speed	
		10 kbits/s	78 kbits/s
Capacity (Net TPDUs/Second)		43 36	388 329

L4 or L5
Transactions/Second
(for Group Size N)
	N = 2
	N = 4
	N = 8
	N = 16
		~ 18
		~ 9
		~ 4
		~ 2
			~ 164
			~ 82
			~ 41
			~ 20

Table 12.1 LonTalk Protocol—Key Throughput Parameters
(For Single Error Free Channel @ 10 kbits/s, and @ 78 kbits/s)

12.2 Network Metrics

For a *single channel* with backlog BL, the expected *mean Network Delay* is

$$N_{delay} = (BL/2 + 1) * busy\_cycle$$

where

busy_cycle = Beta2*(Dmean+ #pri slots) + Beta1 + AvgPktSize

In general case, the delay over k hops is

$$N_{delay} = SUM \{ (BL_i/2 + 1) * busy\_cycle \}_{i = 1,...,k}$$

What is the worst case delay ?

For a channel or a network where the load exceeds throughput (i.e., BL = 1 is never true), the worst case N_{delay} is unbounded. This may happen with some very small probability but it may happen. When BL = 1 at least occasionally, N_{delay} is bounded. Its worst case value is then determined by the arrival rate distribution—the *more uniform* the distribution, the *less* is the worst case *delay*.

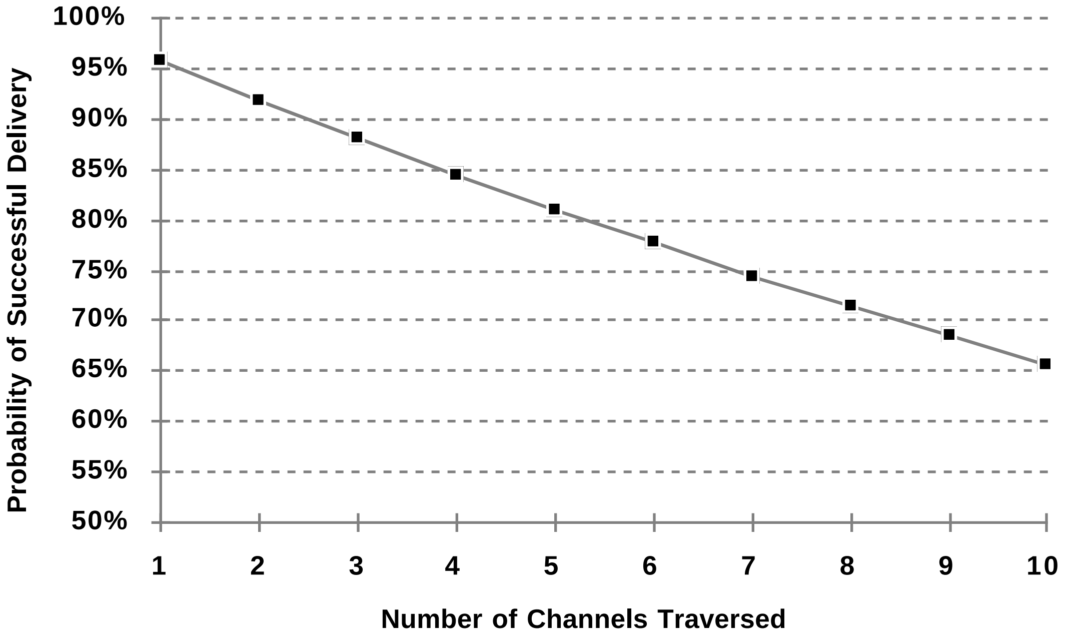
Assuming constant transmission error probability p_e, and a constant randomizing window w over each of the k hops, the *probability of successful delivery* is

$$\mathbf{P\ (L3)\ =\ (1-\ p)^{**}k}$$

where

$p=(1/2w + p_e)$	is the probability of packet loss
1/2w	is the probability of loss due to collision
p_e	is the probability of the packet being corrupted by a transmission error

Graph 12.1 plots the probability of successful delivery (P) of a single packet over k hops where p_e is 1% and w is 16. This yields a probability of error over a single channel of 4.13%. The probability of a failure in delivery is 1 - P. The probability of failure is used to calculate the optimal number of attempts to send the packet since the probability of failure is given by $(1\text{-}P)^{**attempts}$.



Graph 12.1: Probability of Successful Delivery Over k Hops

12.3 Transaction Metrics

The *Transaction Completion Time* in the single channel, single transaction environment is

$$\mathbf{T_{time} = group\_size * busy\_cycle + T_{np}}$$

where T_{np} is the transaction processing time at the destination node. This time is negligible for transport transactions, and it may vary considerably for Request-Response transactions.

Above, collisions (or transmission errors) were ignored. When collisions and multiple transactions are considered, this time increases.

$$\mathbf{T_{time} = x * L4\_timer + y * busy\_cycle + T_{np}}$$

where

$x = 0, 1, ..., L4\_retries$
$y \le \max(group\_size, BL)$

Given that the combined collision + error probability is p, the *probability* of a transaction in a group of n *completing within k retries* is

$$P\{\text{no retry}\} = (1-p)^n$$

$$P\{\le k \text{ retries}\} = \sum_{i=0}^{k} \binom{k+1}{i} p^i (1-p)^{k-i+1} (1-p)^{n-1} \qquad k \ge 0,\ n \ge 2$$

where p is the probability of packet loss over the channel.

P{≤ k retries} is computed as a product of the following two probabilities:

P {lose i messages in k+1 attempts}
P {successfully receive all n-1 ACKs given k-i+1 successful msgs}

The first probability is the binomial probability represented by the first three terms in the equation; the second probability is given by the last term. The above probabilities are plotted in graph 8.1.

12.4 Boundary Conditions — Power-Up

Power-up has an impact on the Transaction Control sublayer: no duplicate detection is done for the first transaction following the reset (transaction number 0). For this reason, the first application operation after a reset should be idempotent.

Rebooting a learning router also has an effect: messages to a specific (subnet, node) address are routed by flooding until the router learns about the location of that subnet. This normally happens with the first acknowledgment from that subnet.

12.5 Boundary Conditions — High Load

Loads exceeding channel capacity will result in increased delays for some transactions. This may lead to timeouts and transactions being aborted with failure or only partial success.

As long as the estimated and the real backlogs match, channel capacity will stay constant at the level(s) defined in table 12.1. Backlog mismatch will always reduce channel capacity. The negative effect of underestimating tends to produce excessive collisions and loss of throughput. Overestimating the backlog by a small amount has a relatively small impact on channel throughput, so backlog calculations in the LonTalk protocol tend to overestimate rather than underestimate the backlog.

13. APPENDIX A— PDU SUMMARY

13.1 Formats and Notation

This appendix provides a summary description of all LonTalk Protocol Data Units (PDUs). PDU syntax is specified pictorially (see figure, following page), with the following explanatory comments:

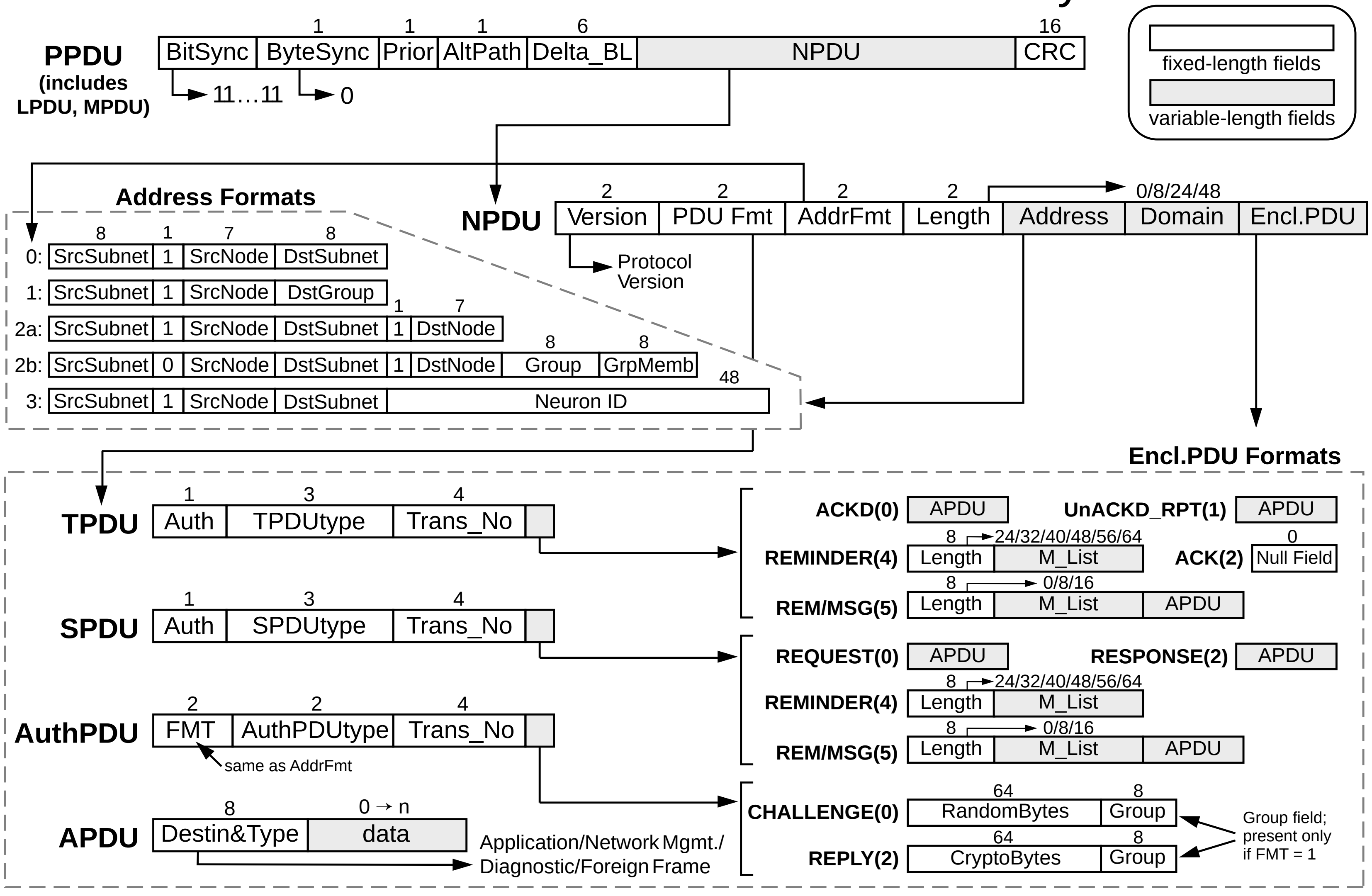
- Field Size. The number above each field specifies the field size in bits;

- Field Values. In order to facilitate the description of semantics, most field values are defined as symbolic ranges. A symbolic range (S0, S1, S2, ..., Sn) always maps onto a numeric range (0, 1, 2, ..., n) in the order specified;

- Bit Ordering. Bit transmission order within a byte is “least significant first”, meaning that the least significant bit is transmitted first. In the attached figures, the least significant bit is the *rightmost* bit of a byte;

- Byte Ordering. Byte transmission order is also “least significant first”, meaning that the least significant byte of a field is transmitted first. In the attached figures, the least significant byte is the *leftmost* byte of a field.

LonTalk® Protocol Data Unit Summary



PPDU (includes LPDU, MPDU)
BitSync
1 ByteSync
1 Prior
1 AltPath
6 Delta_BL
NPDU
16 CRC
11…11
0
fixed-length fields
variable-length fields
NPDU
2 Version
2 PDU Fmt
2 AddrFmt
2 Length
Address
Domain
Encl.PDU
0/8/24/48
Protocol Version
Address Formats
0: 8 SrcSubnet | 1 1 | 7 SrcNode | 8 DstSubnet
1: SrcSubnet | 1 | SrcNode | DstGroup
2a: SrcSubnet | 1 | SrcNode | DstSubnet | 1 1 | 7 DstNode
2b: SrcSubnet | 0 | SrcNode | DstSubnet | 1 | DstNode | 8 Group | 8 GrpMemb
3: SrcSubnet | 1 | SrcNode | DstSubnet | 48 Neuron ID
TPDU: 1 Auth | 3 TPDUtype | 4 Trans_No
SPDU: 1 Auth | 3 SPDUtype | 4 Trans_No
AuthPDU: 2 FMT | 2 AuthPDUtype | 4 Trans_No
same as AddrFmt
APDU: 8 Destin&Type | 0 → n data
Application/Network Mgmt./ Diagnostic/Foreign Frame
Encl.PDU Formats
ACKD(0) APDU
UnACKD_RPT(1) APDU
REMINDER(4) 8 Length | 24/32/40/48/56/64 M_List
ACK(2) 0 Null Field
REM/MSG(5) 8 Length | 0/8/16 M_List | APDU
REQUEST(0) APDU
RESPONSE(2) APDU
REMINDER(4) 8 Length | 24/32/40/48/56/64 M_List
REM/MSG(5) 8 Length | 0/8/16 M_List | APDU
CHALLENGE(0) 64 RandomBytes | 8 Group
REPLY(2) 64 CryptoBytes | 8 Group
Group field; present only if FMT = 1