CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r. 86/09)

Canada

A TRANSPORT LAYER DESIGN FOR LOCAL AREA NETWORKS

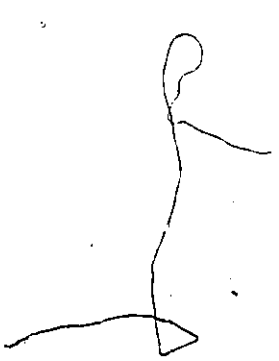
by

ENG HWA TEO

A thesis
presented to the University of Ottawa
in partial fulfillment of the
requirements for the degree of
MASTER OF APPLIED SCIENCE
in
THE DEPARTMENT OF ELECTRICAL ENGINEERING

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The University of Ottawa requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.



ABSTRACT

As one of the seven layers defined in the ISO Reference Model, the Transport Layer exists to provide data delivery service between session entities. Presently, most of the studies on the Transport Layer are focused on long haul networks. This thesis presents a design of the Transport Layer for local area networks. The service offered to the Transport Layer is assumed to be an unacknowledged connectionless service. The Transport Layer shall provide a reliable, in-sequence connection-oriented service to the Session Layer. Messages of two priority levels are provided. To bridge the gap between the above two services, the Transport Layer performs a set of functions, such as error recovery, flow control, duplicate detection and sequencing etc.. The transport protocol provides the means for establishing, maintaining and releasing transport connections. To avoid ambiguous interpretation, the transport protocol described in this thesis is specified by using a Formal Description Technique.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my supervisor, Dr. N. D. Georganas, for his patience and encouragement throughout the preparation of this thesis, and also for his financial assistance.

I would like to express my thanks especially to Jackson Chan, James Kozlowski, Philip Mui and Hasan Ural for their tremendous efforts in helping me to solve many of the problems encountered during the preparation of this thesis.

Finally, I want to thank all the staff of the Department of Electrical Engineering.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CIP	- CLOSE-IN-PROGRESS
CSMA/CD	- Carrier-Sense-Multiple-Access with Collision-Detection
FDT	- Formal Description Technique
ICI	- Interface-Control-Information
IDU	- Interface-Data-Unit
ISO	- International Standards Organization
LAN	- Local Area Network
LSAP	- Link-Service-Access-Point
LSI	- Large-Scale Integration
OIP	- OPEN-IN-PROGRESS
OSI	- Open System Interconnection
PCI	- Protocol-Control-Information
PDU	- Protocol-Data-Unit
SAP	- Service-Access-Point
SDU	- Service-Data-Unit
SPDU	- Session-Protocol-Data-Unit
TCEP	- Transport-Connection-Endpoint
TCEP-ID	- Transport-Connection-Endpoint-Identifier
TIDU	- Transport-Interface-Data-Unit
TPDU	- Transport-Protocol-Data-Unit
TSAP	- Transport-Service-Access-Point
TSDU	- Transport-Service-Data-Unit
VLSI	- Very-Large-Scale Integration

Chapter I

INTRODUCTION

The modern business workplace is filled with increasingly 'intelligent' machines, such as computer systems and workstations, which assist in carrying out day-to-day tasks and communication. In order for a group of people in an organization to operate as an efficient and well-integrated unit, the machines that they work on must be able to communicate and exchange information quickly, easily and reliably. A generally accepted rule of thumb holds that about 80 percent of communication takes place within the local environment. Therefore, the need to connect these systems for communications within a local working environment has become an important problem.

On the other hand, due to the advance of LSI and VLSI technology, the cost and size of machines with a high level of functionality continue to drop. The widespread use and low cost of these devices (which can easily be interfaced to a computer) have prompted network designers to look for a simple way of interconnecting them. The LOCAL AREA NETWORK (LAN) [1][2][3] is particularly applicable to small sites such as an office block, a factory or an university campus.

A local area network is best described in terms of the purpose it serves rather than in terms of how it functions. A local area network is primarily a data communication system intended to link computers and associated devices within a restricted geographical region. The key characteristic of a local area network is the fact that the whole network is confined to one site and is usually under the complete control of one organization.

Since a local area network is confined to a small area, it is possible to employ vastly different communication devices from those commonly used in other telecommunication systems. Inexpensive network interface devices can be employed instead of the relatively complex modems used in long haul networks, e.g. packet switching networks such as DATAPAC. High data transmission speed can be achieved by utilizing the advantages of short distances and the advances in electronics. Thus, local area networks are typified by:

1. Short distances, up to 10 km.
2. A high transmission rate, 0.1 to 20 Mbps.
3. A low error rate.

Today, a considerable variety of means of designing local area networks is available which can be classified in the following ways:

1. By network topology, e.g. star, ring, bus and mesh.

2. By transmission medium, e.g. twisted pair wire, coaxial cable, radio and fibre optics.
3. By transmission technique - the method in which data is transported within the network, e.g. digital baseband and broadband systems.
4. By sharing technique - the way in which network bandwidth is allocated to the users, e.g. dedicated (non-shared), time or frequency division multiplexing, statistical multiplexing and contention.

Networking systems are designed and constructed in functional layers. Each layer performs a specific set of functions. Together, these layers interact with one another to provide total end-to-end network operation. The Reference Model of Open System Interconnection developed by the Internation Organization for Standardization (ISO) consists of seven layers [11]: Application (the highest layer), Presentation, Session, Transport, Network, Data Link and Physical (the lowest layer). This thesis will concentrate on the Transport Layer.

The Transport Layer exists to provide, in association with the layers below it, a reliable and efficient end-to-end data delivery service between processes rather than just between machines. (For example, a single machine may execute several processes each of which may wish to

communicate with processes on other machines.) The Transport Layer relieves these processes from any concern with the detailed way in which transfer of data is achieved. The Transport Layer provides service by performing a specific set of functions. The operation of the Transport Layer is governed by a set of rules and formats known as transport protocol.

Recent technological development and most of the current discussions on local area networks are focused on two layers - the Physical and Data Link layers. They provide transport of information between machines on the network over a physical medium. Although many studies have been done on the Transport Layer, almost all of them are concentrated on long haul networks [4-8]. Since long haul networks have different network characteristics such as large network delay and high error rate, the Transport Layer of these networks would not be efficient enough when used on local area networks.

Presently, many different LAN designs exist (e.g. Ethernet, Mitrenet), and many more are possible. The majority of those in use today were designed for limited applications such as tying together a cluster of word processors or enhancing local communication between processors of a particular manufacturer. To reach the universal level of communication, there must be standards

for LAN that are widely accepted by manufacturers. ISO and the Institute of Electrical Engineers (IEEE) are presently engaged in developing standards. Up to now, the efforts for LAN standardization were focused mainly on the Physical and Data Link Layer. A number of standards for these two layers have also been developed [12][34][35]. Standards for higher layers have not yet been developed. This thesis shall study the operation and propose a design for the Transport Layer in the LAN envirnoment.

The organization of this thesis is as follows :

Chapter 2 describes the network architecture of LANs. This include a discussion on the principles of network layering. A local area network architecture will be proposed.

Chapter 3 discusses the design issues of the Transport Layer. This includes a study on the impact of LAN characteristics on the transport service. Functions that will be performed by the Transport Layer in providing the service are also discussed.

Chapter 4 describes the services provided by the Transport Layer.

Chapter 5 describes the protocol which determines the operation of the Transport Layer. The Formal Description Technique developed by ISO is used to specify the protocol.

Chapter II

THE NETWORK ARCHITECTURE

2.1 INTRODUCTION

The architecture of a computer network defines the structure of the network, identifies the physical and logical elements of the system and specifies the interconnections and interactions among these elements. In this chapter, the basic principles of a structuring technique used to describe network architecture will be presented. A local area network architecture will also be proposed.

Present day network architectures are complex systems. To facilitate understanding of such systems for purposes of design, implementation, and maintenance; current network architectures have been designed in terms of 'layers' (see Figure 1) [9][10][11]. The basic idea of layering is that each layer adds value to services provided by the set of lower layers. These services are offered to the highest layer to run distributed applications. Another basic principle of layering is to ensure independence of each layer by defining services provided by a layer to the next higher layer, independent of how these services are

performed. This permits changes to be made in the way a layer or a set of layers operate, provided they still offer the same service to the next higher layer.

Layering has several benefits. First, it provides a convenient partitioning of functions and therefore provides a well-structured network design. Second, layering helps to reduce functional overlap, thus minimizing network complexity. Third, layering establishes well-documented and well-defined interfaces.

2.2 PRINCIPLES OF LAYERING

2.2.1 GENERAL CONCEPTS

Layering is a structuring technique which permits the network to be viewed as logically composed of a succession of layers, as shown in Figure 1 . An equivalent illustration of layering frequently used in describing network architecture is given in Figure 2 . Successive layers are represented in a vertical sequence, with the physical media for interconnecting the systems at the bottom.

According to this technique, each system in Figure 2 is viewed as being composed of an ordered set of subsystems. A subsystem is an element in a hierachical division of a system which interacts directly only with elements in the next higher division and the next lower division of that

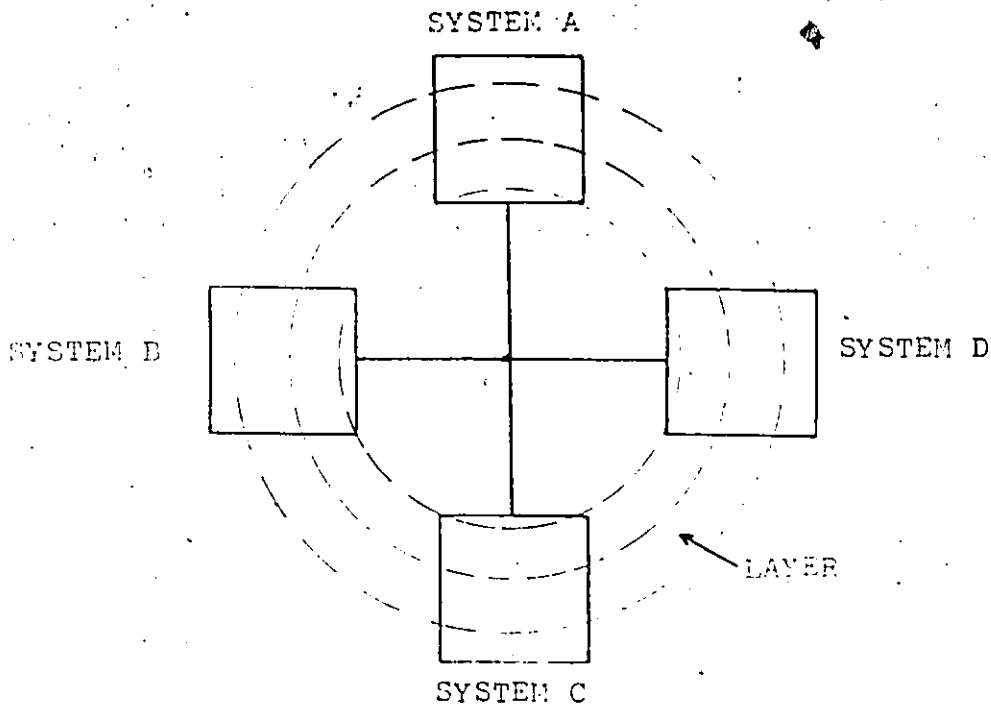


Figure 1: Network Layering

system. The (N)-layer is made up of subsystems of the same rank (i.e., (N)-subsystems) of all interconnected systems. An (N)-subsystem consists of one or several active elements known as (N)-entities. Therefore, each layer is made up of entities; and entities in the same layer are termed peer-entities. Entities in adjacent subsystems communicate through their common boundaries, and entities in the lowest subsystems are assumed to communicate directly with their peers via the physical media connecting them.

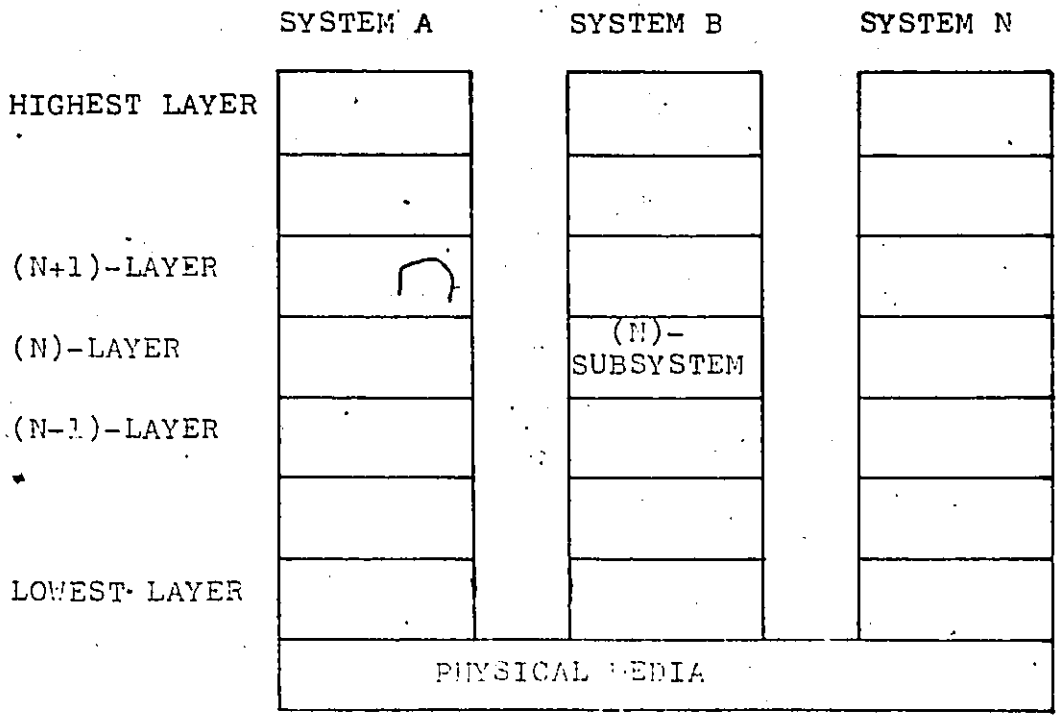


Figure 2: Representation of Network Layering

2.2.2 COMMUNICATION BETWEEN ADJACENT LAYERS

Entities in adjacent subsystems communicate with each other to provide and use a set of services. The (N)-layer (and the layers beneath it) provides (N)-service to the (N+1)-entities through an (N)-service-access-point at the boundary between the (N)-layer and the (N+1)-layer, as shown in Figure 3 .

An (N)-address identifies a particular (N)-service-access-point to which an (N+1)-entity is attached. When the

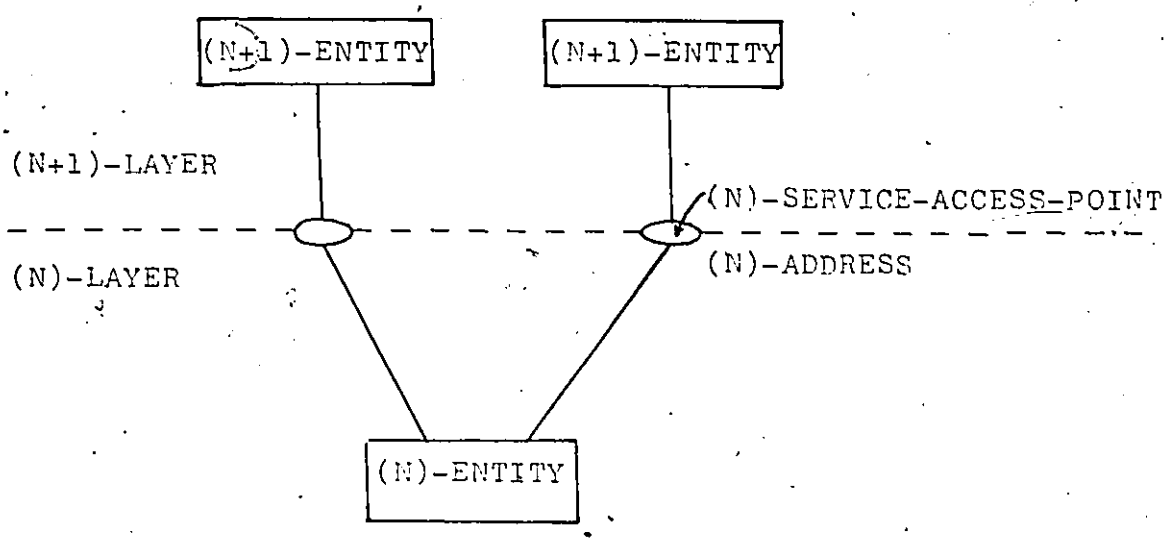


Figure 3: Entities and Service-access-point

(N+1)-entity is detached from the (N)-service-access-point, the (N)-address no longer provides access to the (N+1)-entity. If a permanent attachment between the (N+1)-entity and the (N)-service-access-point can be assured, then the same (N)-address could be used to identify the (N+1)-entity at all times. A service-access-point (SAP) has the following properties:

1. An (N+1)-entity requests (N)-service through an (N)-SAP which allows the (N+1)-entity to interact with an (N)-entity.

2. An (N+1)-entity may be attached to one or more (N)-SAPs concurrently, but only one (N+1)-entity is attached to an (N)-SAP.
3. An (N)-entity may be attached to one or more (N)-SAPs concurrently, but only one (N)-entity is attached to an (N)-SAP.
4. Both the (N)- and (N+1)-entities attached to an (N)-SAP are in the same system.
5. Unless the attachment is permanent, an (N)-SAP may be detached from an (N+1)-entity and re-attached to the same or another (N+1)-entity.

In a real system, a SAP is realized by an interface which is subject to a local system environment. The unit of information transferred between entities in a single interaction through an (N)-SAP is known as (N)-interface-data-unit. It is comprised of :

1. (N)-interface-control-information, which provides the means of managing the (N)-interface; and
2. (N)-interface-data, which is the information from an (N+1)-entity or an (N)-entity for transmission across the (N)-interface.

A given amount of (N)-interface-data would constitute an (N)-service-data-unit whose identity is preserved from one (N)-SAP to another.

2.2.3 COMMUNICATION BETWEEN PEER ENTITIES

Communication between peer-entities of the (N)-layer is provided and managed by the lower layers in the form of (N-1)-service, as illustrated in Figure 4 . This path is in fact made up of a set of logical and physical associations.

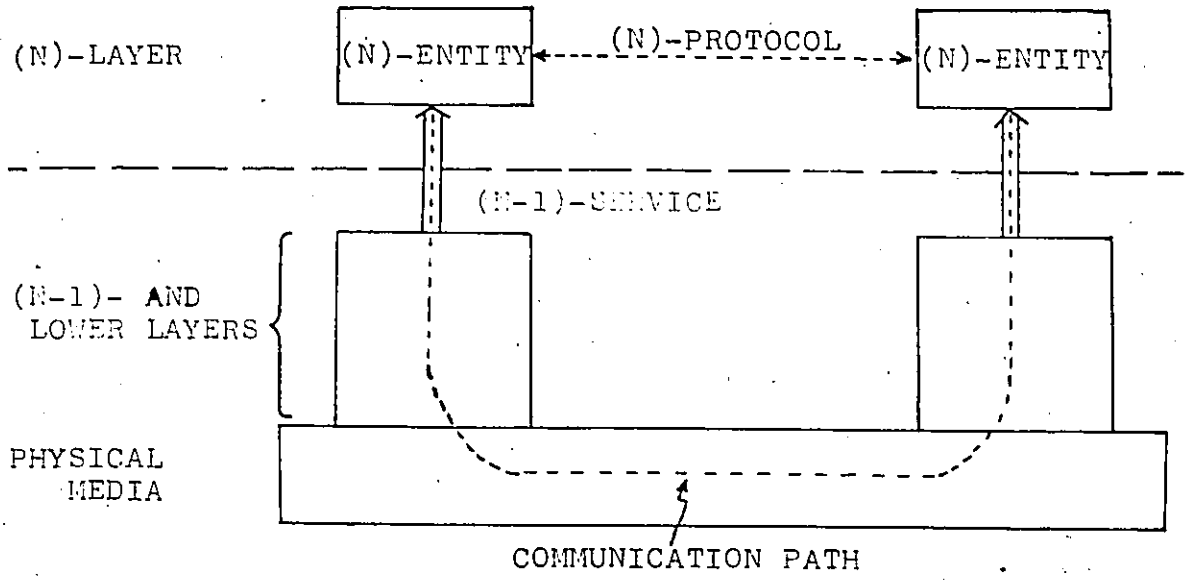


Figure 4: Communication Between Peer Entities

An (N)-connection is an association established for communication between two or more (N+1)-entities, identified by their (N)-addresses. It is offered as a service by the (N)-layer, so that information may be exchanged between the (N+1)-entities. An (N)-connection has the following properties:

1. An (N+1)-entity may have the following (N)-connections simultaneously: single or multiple connections with other (N+1)-entities, or single or multiple connections with itself.
2. An (N)-connection is established by referencing an (N)-address for the source (N+1)-entity and an (N)-address for each of one or more destination (N+1)-entities.
3. An (N)-connection-endpoint is constructed for each (N)-address when an (N)-connection is established.
4. An (N)-connection has two or more (N)-connection-endpoints.
5. An (N)-connection-endpoint is not shared by (N+1)-entities or (N)-connections.
6. An (N)-connection-endpoint relates three elements : (a) an (N+1)-entity, (b) an (N)-entity and (c) an (N)-connection.
7. An (N)-connection-endpoint has an identifier, called an (N)-connection-endpoint-identifier, which is unique within the scope of the (N+1)-entity which is bound to the (N)-connection-endpoint.
8. An (N+1)-entity references an (N)-connection using its (N)-connection-endpoint-identifier.

The (N)-protocol is a set of rules and formats which specifies the communication behaviour between (N)-entities. The information exchange between these entities is known as

(N)-protocol-data-unit. An (N)-protocol-data-unit consists of (N)-protocol-control-information and (N)-user-data. The (N)-protocol-control-information allows the (N)-entities to co-operate through joint operations. The (N)-user-data, on the other hand, is the data transferred between (N)-entities on behalf of their users.

The relationships and the mapping between data units that have just been discussed are summarized in Table 1 and Figure 5 . There is no overall architecture limit to the size of data units. But there may be other size limitations at specific layers. Figure 5 assumes that neither segmenting nor blocking of (N)-SDU is performed. An (N+1)-PDU may be mapped one-to-one into an (N)-SDU, buf many-to-one mapping (i.e. concatenation) is also possible [11]. Figure 5 does not imply any positional relationship between PCI and user-data in PDU, and ICI and interface-data in IDU.

TABLE 1

Relationships Between Data Units

	CONTROL	DATA	CONTROL AND DATA COMBINED
(N)-(N) PEER-ENTITIES	(N)-PROTOCOL-CONTROL-INFORMATION	(N)-USER-DATA	(N)-PROTOCOL-DATA-UNITS
(N+1)-(N) ADJACENT LAYERS	(N)-INTERFACE-CONTROL-INFORMATION	(N)-INTERFACE-DATA	(N)-INTERFACE-DATA-UNITS

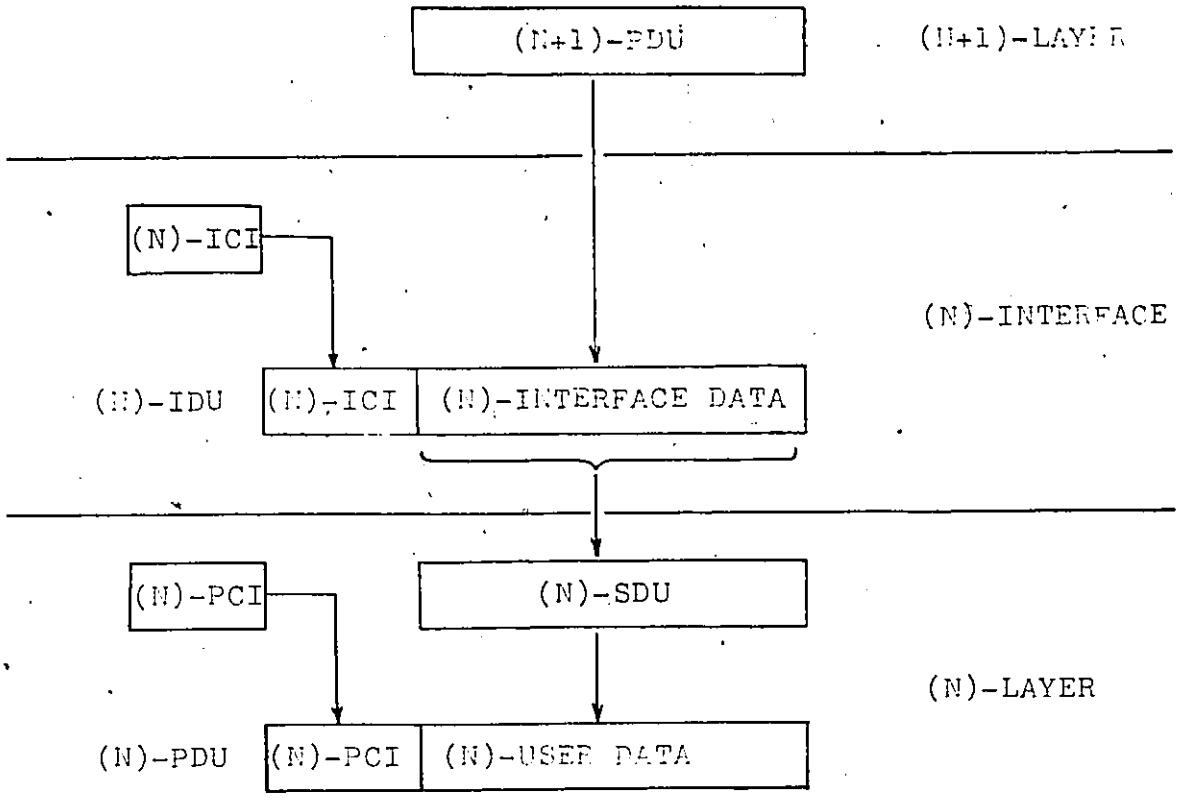


Figure 5: Mapping Between Data Units in Adjacent Layers

2.3 A LOCAL AREA NETWORK ARCHITECTURE

Network architecture assists in providing a uniform framework for communication within a heterogeneous computing environment, and is described by a specific set of layers each of which performs well-defined functions. Some of these functions are :

1. data transmission,
2. data manipulation (reformatting),
3. detection and recovery from transmission errors,
4. designation of boundaries between data sets, and
5. address recognition.

The problem of defining the specific set of layers in a network architecture, and of determining the boundaries between the layers (i.e. the partitioning of function), is fairly complicated. Therefore, instead of designing a completely new network architecture, the International Standard Organization's (ISO's) Reference Model of Open System Interconnection (OSI) is used as a guideline for the development of a local area network (LAN) architecture. It should be noted that the OSI architecture is mainly intended for describing long haul networks. As such, some of the functions ascribed to certain layers in the OSI architecture would not be efficient enough in the LAN envirnoment. This is especially true for those layers that are concerned with the use of the communication subnetwork.

The OSI architecture is made up of seven layers [11], as shown in Figure 6 . Not all systems in the OSI architecture provide initial source or final destination of data. When the physical media do not link all system directly, some systems act only as a relay system by passing data to other systems. Without explaining how these seven layers were chosen, the purpose of each layer is briefly described in Appendix A.

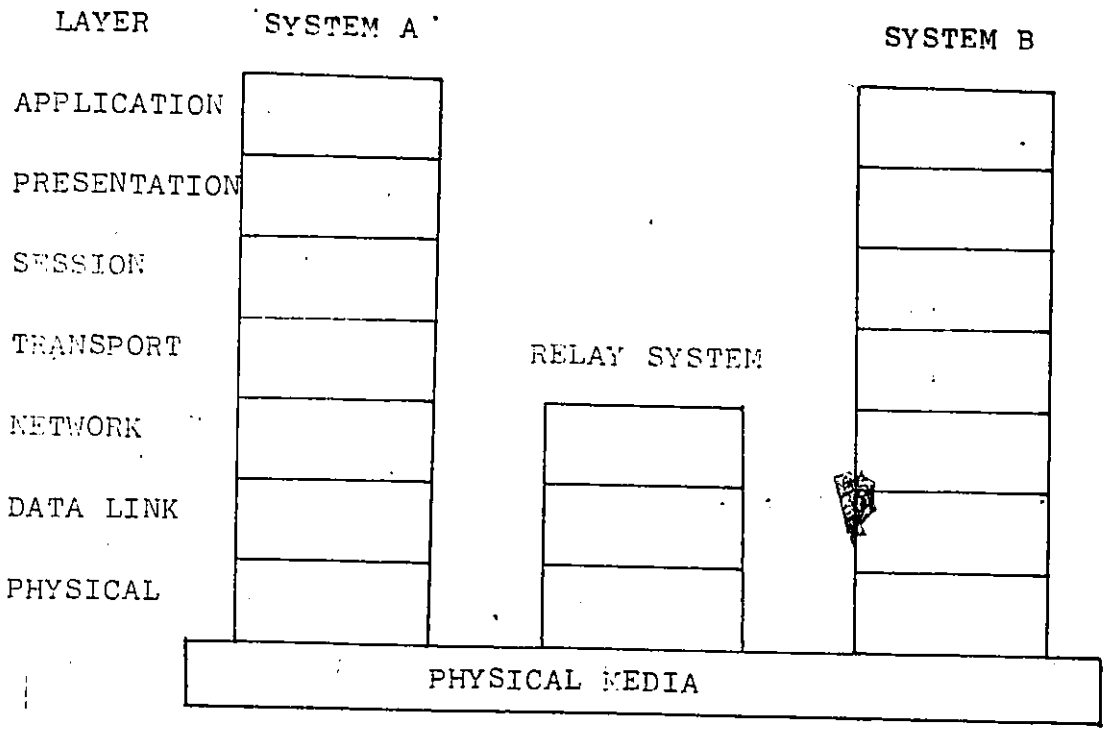


Figure 6: The OSI Architecture

A list of services and functions for each of the seven layers is given in Appendix B. These functions can be roughly classified as processing-oriented functions and

communication-oriented functions. The processing-oriented functions are concerned with the ability of two systems to exchange information and to understand it. These functions are found in the Application, Presentation and Session Layers. The communications-oriented functions as performed by the Transport, Network, Data Link and Physical Layers are concerned with the use of the communications subnetwork in transferring information between two systems.

The local area network (LAN) architecture evolves from the OSI architecture. Since local area networks are but a class of network systems, the unique network characteristics will certainly impact those layers that are concerned with the use of the communications subnetwork. By examining the functions performed by the Network Layer as described in Appendix B, certain functions appear to be irrelevant in the LAN environment; for example, the functions of routing and relaying. Unlike long haul networks which employ sophisticated routing functions, local networks have a very simple routing function because there is no way to selectively route a message in a bus or ring networks. In fact, this type of routing, which involves only adjacent systems, has been taken care of by the Data Link Layer. Therefore, functions of the Network Layer which appear to be irrelevant for the local network may be omitted. Other functions such as error detection and recovery become redundant since they could be performed by the Data Link or

the Transport Layers. As such, the Network Layer could be excluded from a LAN architecture.

Figure 7 shows a possible architecture for local area networks. It consists of six layers which are similar to those defined in the OSI architecture. The Data Link Layer, which is of particular concern in this design, consists of two sublayers: logical link control and media access control. The logical link control sublayer provides the usual data link protocol functions of framing, addressing and formatting in a form which is independent of the access mechanism, topology and physical media in use. The media access control sublayer which is topology-dependent specifies the techniques used in accessing the physical media. Presently, the two popular access techniques are :

1. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) [12], which allows stations to contend for use of the physical media and schedule retransmission for the collided packets;
2. Token Passing [13], which is a mechanism whereby each station, in turn and in a predetermined order, receives and passes the right to use the physical media.

CSMA/CD is applicable only to the bus topology networks, but Token Passing can be used on bus or ring.

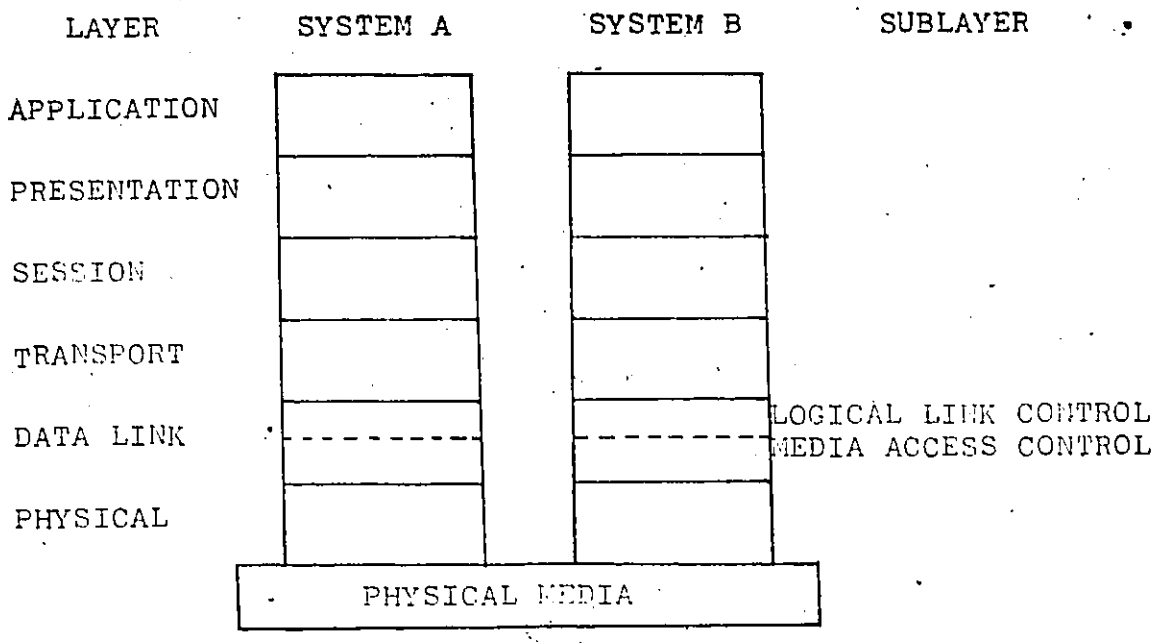


Figure 7: A Local Area Network Architecture

The above LAN architecture does not consider the interconnection of networks. The discussions of LAN interconnection may be found in [2][36]. Of course, the removing of the Network Layer from the LAN architecture will make the Transport Layer dependent on the link level [37][38]. This is especially true on the data segmentation performed by the Transport Layer. Since the Data Link Layer does not provide data segmentation, every data unit provided by the Session Layer must be segmented into a predetermined size imposed by the link level. Other studies on the LAN architecture which included the Network Layer may be found in [39][40].

Chapter III

DESIGN CONSIDERATIONS FOR THE TRANSPORT LAYER

3.1 INTRODUCTION

In the previous chapter, a layered architecture for local area networks has been discussed. This architecture is made up of six layers, each of which provides a certain subset of services to the overall set of network functions. As one of the six layers, the Transport Layer exists to provide a reliable and efficient end-to-end data delivery service between session entities. Throughout the remainder of this thesis, the discussions will be concentrated on the design of the Transport Layer for local area networks.

Figure 8 shows the structure of the Transport Layer and the relationship between it and its adjacent layers. The Transport Layer requires a set of services provided by the Data Link Layer, so that it could in turn offer a set of services to the Session Layer. Obviously, a description of the Transport Layer should include the following elements [11]: (1) the services it provides to the Session Layer, (2) the services it obtains from the Data Link Layer, (3) the protocol which determines the interactions between the transport entities. In this chapter, the discussion will

focus on the design considerations for the Transport Layer. This begins by looking into some design issues of the transport service, and then follows by identifying the functions that will be used by the transport protocol.

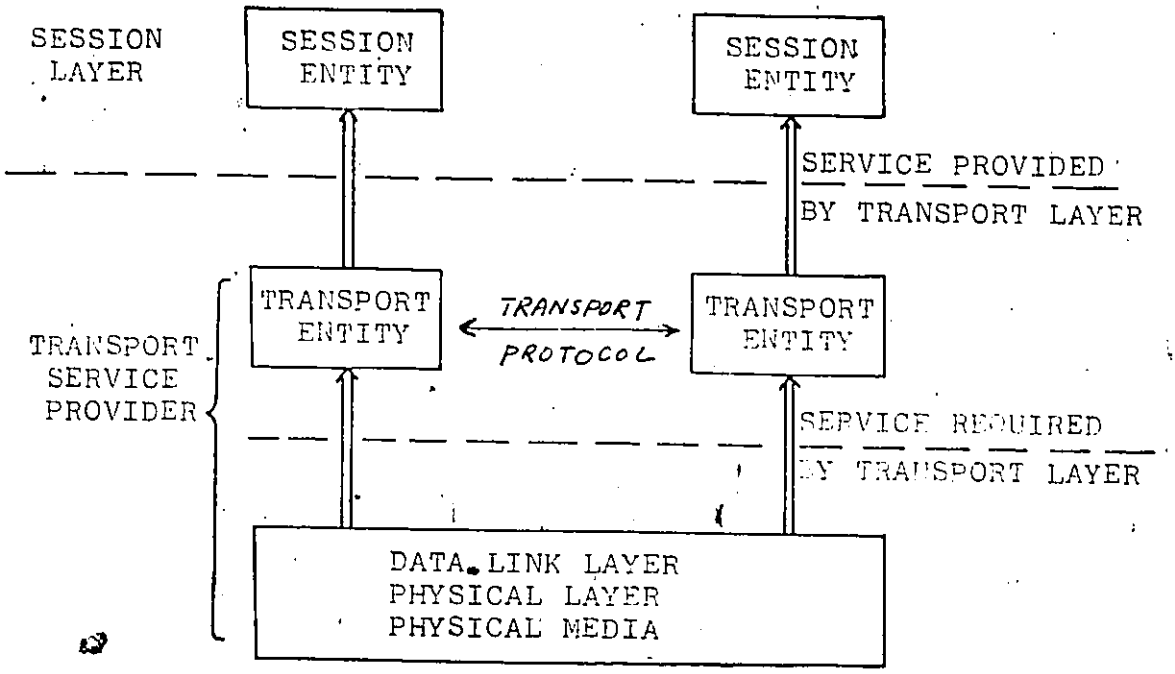


Figure 8: The Transport Layer Structure

3.2 DESIGN ISSUES OF THE TRANSPORT SERVICE

The primary objective of the transport service is to provide data transfer between session entities, and to relieve these entities from any concern with the detailed way in which efficient and reliable transfer of data is achieved. The transport service shall be transparent, in other words, it should not impose any restrictions on the

content, format or coding of user information, nor should it need to understand its structure or meaning. The transport service is usually described by a set of service features [14]. These service features are derived from user requirements and can be classified into two major categories: type of service and grade of service.

3.2.1 TYPE OF SERVICE

Basically, the two types of service that could be offered to a session entity are the connection-oriented service and the connectionless service [15].

A connection-oriented service provides the session entities with the means of establishing and terminating transport connections. These connections are offered as a service by the Transport Layer, so that information may be exchanged between session entities. In the following discussion, communication on a transport connection is assumed to be full-duplex and each connection has only two connection-endpoints. A transport connection has to pass through three phases which are distinct to the session entity: an establishment phase, a data transfer phase and a termination phase. Before a data transfer phase could begin, the three parties (namely the two session entities that wish to communicate and the transport service provider) must first agree on their mutual willingness to establish a

connection. There is no possiblity of data transfer through an unwilling service to an unwilling partner. After a connection has been established, the three parties must continue to agree on the acceptance of each other's data units transferred over the connection. A graceful connection termination should not interfere with the usual orderly transfer of data.

A connectionless service provides session entities with the means of conveying data units without establishing, maintaining and releasing a transport connection. The most visible characteristic of connectionless service toward a session entity is the single service access. In other words, all the information required to deliver the data is presented to the transport service provider, along with the data, in a single interaction that is not related in any way to the prior or subsequent interactions. Unlike the connection oriented service, which requires the establishment of a three-party agreement, the connectionless service involves only two-party agreement. For each data transfer, individual agreement between each session entity and the transport service provider is required; but there is no agreement between the involved session entities. Typical examples of connectionless service are: (1) transaction service, which entails in sending a data message from one session entity to another; (2) broadcast service, which allows sending a single data message to all listening

session entities; and (3) multicast service, which permits sending a message to all of a specified list of session entities.

In this design, the Transport Layer shall provide only connection-oriented service. The reasons for selecting this type of service are as follows.

First, the connection-oriented service requires that the two communicating session entities and the service provider are bounded by an agreement which they have reached during the establishment phase. This agreement permits them to reserve a set of resources (such as buffers and programs) for their exclusive use throughout the lifetime of their associations, so as to maintain orderly communication and status information. If each party must allocate resources that required to carry out data transfer operations, negotiation provides opportunity to scuttle the establishment of connection if the resources that would be required to support it cannot be obtained. This negotiation process also allows a variety of access-control, security, accounting, and identity-verification to be carried out to establish the willingness of the three parties to communicate. In addition, when more than one protocol is defined for the Transport Layer, the negotiation process provides an opportunity to select the one best suited to the current circumstances.

Second, once a connection has been established, it may be used to transfer successive data units until the connection is released. These data units are related to each other simply by virtue of being transferred in the context of a particular connection. Since data units transferred over a connection are related, out-of-sequence, missing and duplicated data units can easily be detected and recovered. The data unit relationship maintained by a connection also enables the use of flow control techniques to ensure that the peer-to-peer data transfer is synchronized.

Third, transaction service can be easily integrated into a connection-oriented service [17]. This could be done by beginning the three-party handshakes for connection establishment and termination in the single segment, which will also contain the user data. The transport interface, which usually involves several interactions in the case of the connection-oriented service, could be easily extended to allow such a transaction from a single transport service request.

One of the reasons for not selecting connectionless service is that the service provides no negotiation between the involved entities. Since no agreement needs to be made between a pair of session entities before data units can be transmitted between them, no buffers or other resources are reserved. If a data unit arrives at a session entity that

has no resources for buffering the incoming information, or if the entity is busy due to some reasons, the data unit will be lost. On the other hand, a data unit transmitted by connectionless service is completely unrelated to any other data units. This data unit independence implies that a series of data units handed one after another to a connectionless service for delivery to destination will not necessarily be delivered to the destination in that order. In order to provide a reliable transport service, the Transport Layer requires more complex procedures for transmission error recovery and flow control etc.. Presently, the connectionless service is not as widely understood as the connection-oriented service in terms of providing reliable service. This is especially true with respect to broadcast and multicast service, since the usual acknowledgement scheme becomes unpalatable when a single broadcast or multicast would cause hundreds or thousands of acknowledgements [16].

Another reason for not selecting connectionless service is that the service represents a greater overhead for each transmission. Data units transmitted using connectionless service are entirely self-contained. All information required to transmit the data unit must be included in each transmission. Therefore, the corresponding layer average data unit size usually represents a greater overhead for each transmission than is incurred during data transfer phase of a connection.

Because of the above reasons, the connection-oriented service is chosen in this design, with the recognition that the transaction service is the most desirable of the additional features.

3.2.2 GRADE OF SERVICE

The delivery of data by the transport service provider should agree with the grade of service requested by the users. In general, this grade of service would define acceptable error and loss levels, desired delay, priority levels, security and other considerations.

Different grades of service are meaningful to the users of an unreliable subnetwork who are willing to accept an unreliable service at time/cost savings in return for dispensing with the extra processing performed by the Transport Layer. This is especially true in long haul networks where transmission costs and propagation delay are high. However, geographic restriction permits local networks to utilize low-cost but very high-bandwidth transmission media. Inexpensive interface devices are usually used to connect each of the hosts to the transmission media. As such, the cost of transmitting data in a local network is low compared with the cost of the hosts themselves. On the other hand, high-speed data transmission in a small geographic area results in an

insignificant propagation delay. Therefore, in local area networks, transmission cost and network delay should not be the factors that will affect the transport user's decision in requesting a particular grade of service. As a result, the transport users may request grades of service according to the types of application. For example, file transfer requires a grade of service that defines an error-free and in-sequence delivery with some network delay. However, in the case of real-time voice communication, because of the required timing constraints, it is better to accept small levels of error and loss in the transmission than to wait for retransmission of an entire message.

In this design, the Transport Layer will provide a reliable, sequenced service in which messages could be transmitted at two different priority levels. These two levels of service are normal and expedited data deliveries. The expedited data delivery is particularly useful in the situation where some data submitted to the transport service provider may supercede data previously submitted. This service allows transport users to exchange such high-priority information even when normal data flow is temporary blocked.

3.2.3 TRANSPORT INTERFACE

Transport interface provides the means for communication between transport and session entities. The details of this interface are heavily dependent on the local computer environment, but certain properties of the interface must be specified. In here two properties will be mentioned. First, the transport interface should have a mechanism for flow control. This will prevent session and transport entities from swamping one another with data. Second, a transport interface should allow the session entity to be notified of the unrecoverable error detected by the transport service provider.

3.3 DESIGN ISSUES OF THE TRANSPORT PROTOCOL

The transport protocol is a set of rules and formats which determine the communication behavior of transport entities in the performance of transport functions. Its main objective is to allow transport entities to provide reliable transport service requested by the session entities. To achieve this objective, the protocol must invoke the necessary transport functions to bridge the gap between the service available from the lower layer and those to be offered to the session entities. The design of the transport protocol begins by determining the behavior of the Data Link Layer and the services it provides. This is then

followed by a discussion on the transport functions required to provide the reliable service.

Another area in protocol design is the protocol performance. The two main aspects of protocol performance are efficiency and reliability [18]. Efficiency takes into account the throughput, delay, buffering requirements, overhead and other quantitative measures. Reliability, on the other hand, is concerned with the correct operation of the protocol in managing connections and transfering data. In this thesis, only the problems of protocol reliability will be addressed.

3.3.1 THE NETWORK ENVIRONMENT

According to the LAN architecture defined in Chapter 2, the nature of the Data Link Layer to which the Transport Layer interfaces will have considerable impact on the operation of the layer. This is especially true with respect to the reliability of the Data Link Layer. If the Data Link Layer provides reliable and in-sequence data delivery, then the Transport Layer does not require error-checking and sequencing. Otherwise, the Transport Layer must itself assume the responsibility of making the transport service reliable by performing the necessary functions.

In this design, the Data Link Layer is assumed to provide the following services [19]:

1. a best-effort station-to-station data delivery;
2. transmission error detection; and
3. provision of means for transport entities to uniquely identify their correspondents.

In other words, the Data Link Layer does not assume the responsibility for reliable data transfer. Although each data frame is protected from error by a cyclic redundancy checking error-detection-code, no acknowledgement is sent at data link level for each frame correctly received. Therefore, the Transport Layer must assume the burden of providing a reliable data delivery between session entities.

Network characteristics also have an impact on the design of transport protocol. The low transmission delay inherent in LANs, as well as their high data rate, can eliminate the need for complex buffer management and flow control. For example, the receiver will not buffer the data which is out of sequence. As noted in Section 3.2.2, bandwidth is inexpensive in LANs. Therefore, there is little motivation to be concerned with protocol features designed to reduce the size of the header sent with each message. This is in contrast to protocols developed for long haul networks which assume that bandwidth is expensive. Examples of ways in which the extra header space can be used to simplify processing include [20]:

1. Having a single standard header format with fields in fixed locations, rather than having optional fields or multiple packet types. As such, field extraction at the host can be optimized, thus reducing processing time.
2. Using addresses that directly translate into addresses of processes at the receiver without table lookup.

3.3.2 THE TRANSPORT FUNCTIONS

As noted in the previous section, the Data Link Layer does not provide reliable service. As such, the Transport Layer must perform transmission error recovery, duplicate detection and sequencing so as to ensure intact and in-order delivery. A flow control mechanism is needed to keep the communication synchronized between transport entities. The Transport Layer must also provide a method for session entities to address their correspondents. Functions for establishing and terminating transport connections are required. Because of the limited frame size imposed in the Data Link Layer, data segmentation is needed in the Transport Layer. A discussion of these functions follows.

3.3.2.1 ADDRESSING

Transport address is the means by which transport entities uniquely identify their users (i.e. session entities). It is also used by the session entity to identify their counter-parts. Figure 9 shows a common configuration where a transport entity provides service to a number of session entities. The transport entity is uniquely identified by a host address provided by the Data Link Layer.

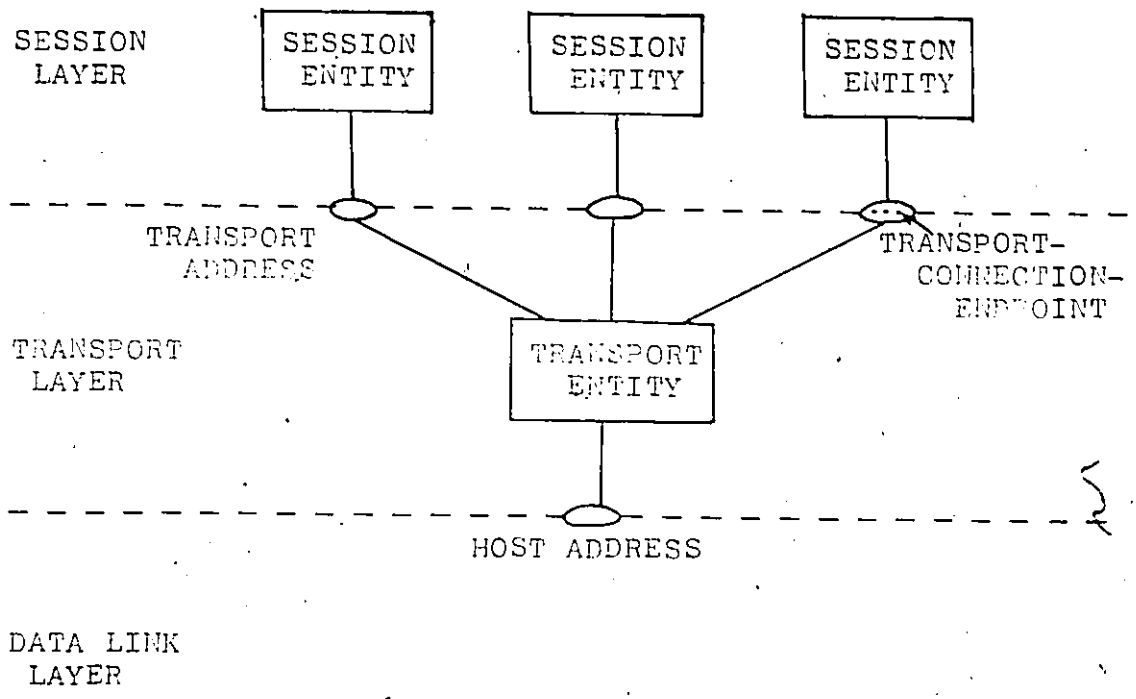


Figure 9: Hierarchical Mapping of Transport Address

When a pair of session entities wish to communicate with one another, the respective entities are required to know

each other's location (i.e. host address). That is, a transport entity should be able to determine the remote host address from a remote transport address given by its user. Typical choices for an addressing scheme are:

1. hierarchical address-mapping, in which local names are concatenated with a global address to form a unique address; and
2. address-mapping by table, in which a defined set of global addresses is distributed among all processes to which network access is required, and the local system maps these global addresses onto the corresponding local names.

Since transport addresses (see Figure 9) are always mapped into only one host address, hierarchical construction of addresses will be used. In this case, a transport address consists of two parts: a host address and a port number. The hierarchical structure is shown in Figure 10 . A transport address can uniquely identify a session entity because the host address is unique within the network, and the port number makes the transport-service-access-point uniquely identifiable within the scope of the host address. A hierarchical structure of addresses simplifies the address-mapping functions because of the permanent nature of the mapping it presupposes. Instead of keeping a list of addresses (e.g. address-mapping by table lookup), the hierarchical address-mapping functions require only to

recognize the hierarchical structure of a transport address and extract the host address it contains. Therefore, this addressing scheme provides the means for each station to use addresses that directly translate into addresses of processes at the receiving end without table lookup.

Since more than one connection are possible for a given pair of transport-service-access-points (TSAP), local identifiers are introduced to distinguish among these connection-endpoints within a TSAP. A transport-connection-endpoint-identifier (TCEP-ID) may be used to uniquely identify each of these connection-endpoints within a local area network. In this case, a TCEP-ID consists of a transport address and a local identifier as shown in Figure 11 .

In this design, the service provided by a transport entity through a TSAP is identical to the servie provided through another TSAP. Therefore, it is sufficient just to describe the operation at a particular TSAP. The transport and the session entities of the same system communicate with one another through the use of local identifiers. A transport entity identifies a transport connection by referencing the local identifier at the local connection-endpoint and the TCEP-ID at the remote connection-endpoint.

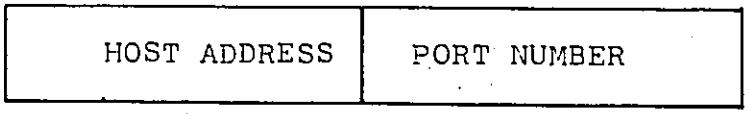


Figure 10: The Hierarchical Structure of Transport Address

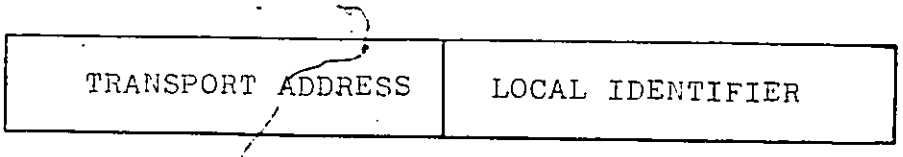


Figure 11: The Hierarchical Structure of TCEP-ID

3.3.2.2 CONNECTION ESTABLISHMENT AND TERMINATION

The connection establishment functions are responsible for setting up transport connections between session entities. These functions include the selection of local identifiers, the initialization of state variables associated with the connection and the mapping of transport addresses onto host addresses.

Each transport entity is responsible for selecting a local identifier to form a TCEP-ID which the partner will use. This mechanism is symmetrical and therefore avoids the need to assign a status of master or slave to partners and avoids call collision. This mechanism also provides identification of the transport connection.

The connection termination function is used to provide both graceful and non-graceful release of transport connection, regardless of the current activity.

3.3.2.3 TRANSMISSION ERROR RECOVERY

The Transport Layer employing an unreliable data link service must ensure that data accepted for delivery is delivered correctly within the desired grade of service. Transmission error recovery is required to guarantee the integrity of each data units delivered. This mechanism consists of three functions: error detection, acknowledgement and retransmission.

The first step towards transmission error recovery is error detection. Since the Data Link Layer provides error checking service, the Transport Layer may use this service instead of performing its own error detection. If a transport-protocol-data-unit (TPDU) is received correctly, a positive acknowledgement may be returned to the sender. This acknowledgement provides a positive confirmation to the sender that the TPDU was received without error. The error recovery is done by means of a timeout. A TPDU transmitted starts a timer, and if a positive acknowledgement is not received before the timeout occurs, the TPDU is retransmitted. If a positive acknowledgement is not received after some number of retransmissions, the sending transport entity may take some special actions, such as to close the connection and to notify the session entities. Detailed discussions on the procedures for acknowledgement and retransmission is given in section 3.3.2.5.

3.3.2.4 SEGMENTATION

All packet-switching systems impose a limit on the packet size they can transmit. Thus every transport-service-data-unit (TSDU) provided by the session entity must be segmented into sets of sequential TPDUs before being transmitted. Since the Data Link Layer does not provide data segmentation, the TPDU size must not exceed the data unit size imposed by the Data Link Layer. In this

design, the transport protocol shall provide a record-oriented service interface to the session entities [18]. With this interface, the session entity has a means of signalling logical breakpoints in the source of data. The transport protocol conveys this information when it delivers the TPDUs. In this case, a breakpoint indicates the end of a TSDU.

3.3.2.5 DUPLICATE DETECTION AND SEQUENCING

Retransmission until positive acknowledgement is received guarantees that every TPDU transmitted has been received by the receiver. Whenever an acknowledgement is lost, a duplicated TPDU will be retransmitted to the receiver. Therefore, the transport protocol must have a means of differentiating between duplicates and new TPDUs at the receiving end.

This mechanism consists of an unique identifier attached to each TPDU by the sender. At the receiving end, the protocol must keep track of the TPDU identifiers it has successfully received. The identifier of each received TPDU is checked against this information and appropriate action is taken. Identifiers are normally assigned sequentially by the transmitter and play a role of both an unique identifier and a sequence number. Using sequence numbers for identifiers allows the receiver to remember a single number

which identify the TPDU and all its predecessors which have been received. In this design, a receiver will not acknowledge for every TPDU it has received correctly. The acknowledgement is sent only at the request from the sender. In this case, only the transmitted TPDU which contains a request for acknowledgement shall start a timer at the sender. If a positive acknowledgement is not received before a timeout, the TPDU is retransmitted. If a TPDU is detected to be a duplicate at the receiver, it will be discarded. On the other hand, if a TPDU is received out of order, it will also be discarded but an acknowledgement which indicates the expected TPDU sequence number will be sent. The sender shall retransmit the TPDU indicated by the expected TPDU sequence number.

3.3.2.6 FLOW CONTROL

Mechanisms that limit the rate of information transfer fall under the category of flow control. Flow control permits more efficient use of network resources while avoiding buffer overflow at the destination. At the transport level, ususally it is the receiver who wishes to limit the activity of the sender. An effective flow control mechanism for the transport protocol is based on granting 'credit' for transmission. The receiver grants credits for a number of TPDUs to the sender so that both sides know exactly how many TPDUs will be exchanged.

Credits may be passed from receiver to sender in a special control TPDU dedicated to that purpose. Credits may be expressed relative to the sequence number of TPDUs [5]. The number of credits provided by the receiver is frequently called the WINDOW SIZE. The receiver indicates the window of acceptable TPDUs by specifying the low sequence number and the number of TPDUs beyond it that the sender is allowed to transmit.

A window-based flow-control mechanism may be implemented efficiently by using the acknowledgement sequence number both for error-control and flow-control purposes. This number then provides an acknowledgement for all lower-numbered TPDUs and the base for the window size information. It is important to note that acknowledgement for error-control purpose is not identical to granting new flow-control credits. A TPDU may be acknowledged without granting new credits by advancing the acknowledgement sequence number and reducing the window size by a corresponding amount. New credits may be granted without acknowledging any TPDU received by increasing the window size but leaving the acknowledgement sequence unchanged.

3.3.3 ISSUES IN PROTOCOL RELIABILITY

Several kinds of potential protocol 'traps' in packet communication systems are deadlocks, message ping-ponging,

unspecified receptions and nonexecutable interactions. [21][22][23]. A deadlock condition arises when two communication entities wait indefinitely for the other either to complete or to initiate an action. A message ping-ponging condition occurs when two communication entities are in a loop such that, upon receipt of a message from one entity, the other responds with message which in turn causes the first to repeat its own message. An unspecified reception occurs when a reception that can take place is not specified in the design. A nonexecutable interaction is present when a design includes message transmissions and receptions that cannot occur under normal operating conditions.

These types of protocol traps are the result of deficiencies in the design of communications procedures. When they occur, the throughput between two communicating entities becomes effectively zero. Therefore, if possible, they should be prevented in the design stage by defining the protocol in a way that they will not occur.

One of the possible causes of protocol trap is due to the existence of certain 'home states' in the protocol. Such a state may be the 'steady state' of the protocol after initialization, or it may be used for error recovery after failure. Therefore, it is critical that they are reachable from all system states that are reachable from the initial

state. As a result, there exists a loop in the state transition diagram, starting at a home state and heading back to it. If this loop is not properly designed, then the execution of the protocol will not produce any useful results. During the designs, a technique of perturbation [22] based on a reachability analysis was used to detect design errors due to deadlocks, message ping-ponging, unspecified receptions and nonexecutable interactions (see Appendix C).

Some of the deadlock conditions that could be encountered in this design are: (1) loss of control information during transmission, (2) zero flow control windows, and (3) interaction between window size and fragmentation. They are discussed as follows.

The first condition is the loss of control information during transmission. When a transport entity sends a command to another entity, it is committed to the interaction. Therefore, the sending entity needs to have a timeout arrangement to prevent being locked in case the command is lost or the receiver is not ready. Also, the number of retransmission attempts must be limited.

The second condition is the zero flow control windows. Consider a system employing a window-based flow-control mechanism as discussed in Section 3.3.2.4. In this case, it is possible for the receiver to expand or shrink the window

size dynamically, according to its willingness to provide buffer space. In particular, when the receiver is unwilling to accept data, it can reduce its window size to zero. When the window is closed, a deadlock can arise. The problem is how to reopen a window of size zero. To handle this, the sender may use a special command to request credit. At the receiver, this command is handled immediately without regard for flow control restrictions.

The third condition is the interaction between window size and fragmentation. When the length of a transport-service-data-unit (TSDU) exceeds the maximum size of a transport-protocol-data-unit (TPDU), the TSDU has to be fragmentated into set of sequential TPDUs for transmission. Suppose that the window-based flow-control mechanism is used and that the number of TPDUs in the TSDU exceeds the window size. That is, the TSDU length exceeds the product of TPDU size and window size. Then, after the sender has transmitted a number of TPDUs equal to the window size, it has to wait for an acknowledgement from the receiver before further transmission can proceed. Consider the scenario that a TSDU is not processed by the receiver, e.g. an acknowledgement is not generated, until after the TSDU is completely received. In this case, a deadlock occurs since the sender is waiting for an acknowledgement in order to continue sending the remaining TPDUs. To prevent this kind of deadlock, intermediate processing of the partially

received TSDU should be allowed to proceed, so that acknowledgement can be returned to the sender.

Chapter IV

DEFINITION OF THE TRANSPORT SERVICE

4.1 INTRODUCTION

The transport service is usually defined in an abstract manner in the sense that it describes the types of command and their effects, but leaves open the exact format and mechanisms for conveying them [24][25]. In practice, these formats and mechanisms are greatly affected by the properties of the local system environment. Therefore, they are considered as a matter of local implementation and will not be discussed here.

The definition of the transport service presented here is in terms of transport service primitives. These service primitives, each of which represents a single interaction, describes the operation at the transport interface through which the service is provided. Service primitives are defined conceptually, which means they would not be prescribed a syntactical realization. However, parameters that are vital to the operation should be specified. The execution of transport service primitives is associated with the exchange of parameters between the transport entity and the session entity. This execution of service primitives

must also conform to a predetermined order to render the service meaningful to the users. A service primitive should be defined in terms of:

1. the specific task it performs,
2. the parameters associated with the interaction, and
3. the identity of the party (i.e. the service provider or the user) that initiates the interaction.

The transport service primitives and the parameters applicable to the connection-oriented service are presented in Table 2 . These service primitives provide the communicating session entities with the means to :

1. establish a connection,
2. transfer data,
3. transfer expedited data,
4. control the flow of data,
5. terminate a connection, and
6. reset a connection.

TABLE 2

Transport Service Primitives

PHASE	SERVICE PRIMITIVES	PARAMETERS
CONNECTION ESTABLISHMENT	T_CONNECT_Request	LOCAL IDENTIFIER CALLED ADDRESS USER CONTROL DATA
	T_CONNECT_Indication	LOCAL IDENTIFIER CALLING ADDRESS USER CONTROL DATA
	T_CONNECT_Response	LOCAL IDENTIFIER CALLING ADDRESS USER CONTROL DATA
	T_CONNECT_Confirm	LOCAL IDENTIFIER CALLED ADDRESS USER CONTROL DATA
DATA TRANSFER	T_DATA_Request	LOCAL IDENTIFIER USER DATA
	T_DATA_Indication	LOCAL IDENTIFIER USER DATA
	T_EX_DATA_Request	LOCAL IDENTIFIER USER DATA
	T_EX_DATA_Indication	LOCAL IDENTIFIER USER DATA
	T_EX_DATA_Response	LOCAL IDENTIFIER
	T_EX_DATA_Confirm	LOCAL IDENTIFIER
	T_PURGE_Request	LOCAL IDENTIFIER
	T_PURGE_Indication	LOCAL IDENTIFIER
CONNECTION TERMINATION	T_DISCONNECT_Request	LOCAL IDENTIFIER USER CONTROL DATA
	T_DISCONNECTION_Indication	LOCAL IDENTIFIER REASON USER CONTROL DATA

4.2 CONNECTION ESTABLISHMENT SERVICE

The connection establishment service provides the session entities with the means of establishing a transport connection. Four transport service primitives are used to describe this service: T_CONNECT_Request, T_CONNECT_Indication, T_CONNECT_Response, and T_CONNECT_Confirm. The order in which these service primitives are invoked to establish a transport connection is shown in Figure 12 .

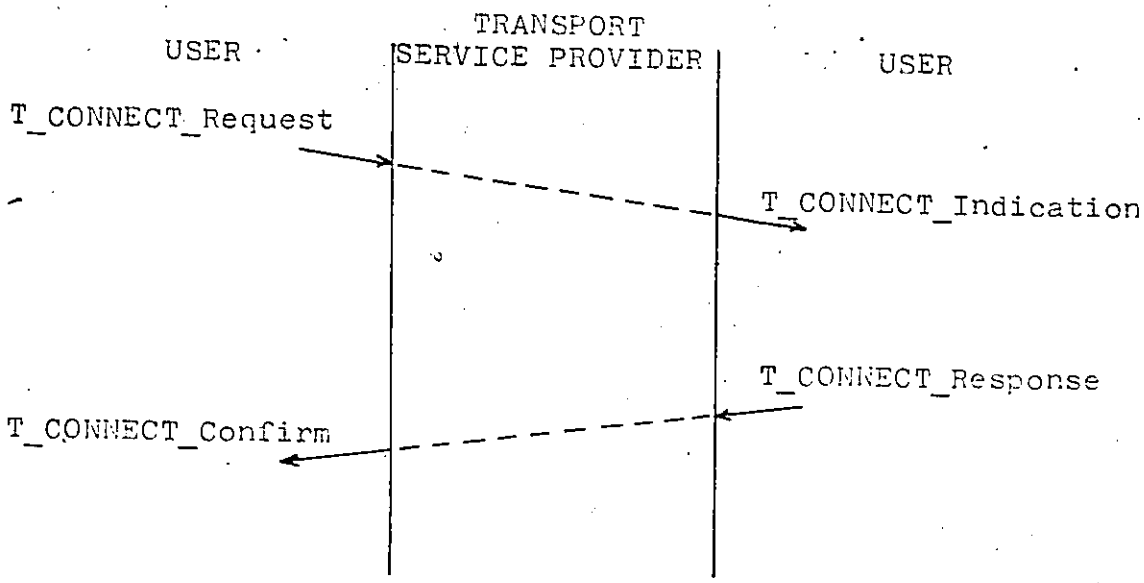


Figure 12: Connection Establishment Service

When a session entity wishes to establish a transport connection with a remote session entity, it will invoke the connection establishment service by passing the T_CONNECT_Request to the local transport entity. This request is then sent, in a different form, to the called station. T_CONNECT_Request may convey the following information:

1. a local identifier, which will be used by the local transport and session entities to identify the connection;
2. the called transport address, which is the remote session entity address; and
3. a limited amount of optional session control information which is considered to be a complete transport-service-data-unit.

As a result of the request by the calling session entity, the called session entity receives the T_CONNECT_Indication from the corresponding transport entity. This service primitive contains a local identifier, the calling transport address and, if any, the control information of the calling session entity.

If the called session entity wishes to participate in establishing the transport connection, it will then acknowledge the corresponding transport entity with the T_CONNECT_Response. At this point, the called session

entity is in the data transfer phase although the connection establishment is not yet completed. The parameters associated with this service primitive are (1) the local identifier, which is the same identifier used in T_CONNECT_Indication, (2) the calling transport address, and (3) a limited amount of optional control information belonging to the called session entity.

Upon receiving the response from the called station, the calling session entity will be informed by the local transport entity through the use of the T_CONNECT_Confirm. The connection establishment is now completed, with the calling transport entity in the data transfer phase. The T_CONNECT_Confirm may contain the same local identifier used in T_CONNECT_Request, the called transport address and, if any, the control information of the called session entity.

So far, only ideal connection establishment has been discussed. It is possible, however, that a connection request from a session entity could be rejected either by the transport service provider or the remote session entity for a number of reasons, e.g. a lack of resources. The discussion of these issues will be presented in Section 4.4.

4.3 DATA TRANSFER SERVICE

When a transport connection is established, it represents a two-way simultaneous data path provided for the exchange of transport-service-data-units. The data transfer service is offered to the attached session entities for as long as the transport connection exists. The participating service primitives in the data transfer services can be grouped into the following three areas:

1. Normal transfer service: T_DATA_Request and T_DATA_Indication.
2. Expedited transfer service: T_EX_DATA_Request, T_EX_DATA_Indication, T_EX_DATA_Response and T_EX_DATA_Confirm.
3. Purge service: T_PURGE_Request and T_PURGE_Indication.

4.3.1 NORMAL TRANSFER SERVICE

The transport service provider delivers transport-service-data-units (TSDUs) in the same order in which they were submitted by the session entity, from one end of the transport connection to the other. The service provider does not limit the size of TSDUs. Each TSDU has a distinct beginning and ending which are known to each local transport entity. A transport-service-data-unit can be transferred across the transport interface in one or more separated

transport-interface-data-units (TIDUs), as illustrated in Figure 13 . The size of a TIDU is not necessarily the same at each end of the transport connection, but each TIDU contains a service primitive together with its associated parameters. Figure 14 shows that either session entity may initiate data transfer by sending T_DATA_Request to the corresponding transport entity. As soon as the data arrives at the receiving station, the local session entity will be informed of the incoming data through the use of T_DATA_Indication. Local indentifier and user data are the parameters associated with each service primitive.

Flow control exists at both transport interfaces. It regulates the rate at which transport-interface-data are passed between a session entity and a transport entity. It is usually used to reduce congestion at the local transport entity. At the same time it may also provide synchronizaion between a sending session entity and a receiving session entity. With synchronized data transfer, a session entity can dynamically control the rate at which it receives transport-interface-data. This flow control condition may eventually be propagated to the remote transport interface, so as to control the rate at which the sending transport entity will accept transport-interface-data from the corresponding session entity. When the transport service provider delivers TSDUs, it is assumed by its users to provide a reliable data delivery. Therefore, acknowledgment

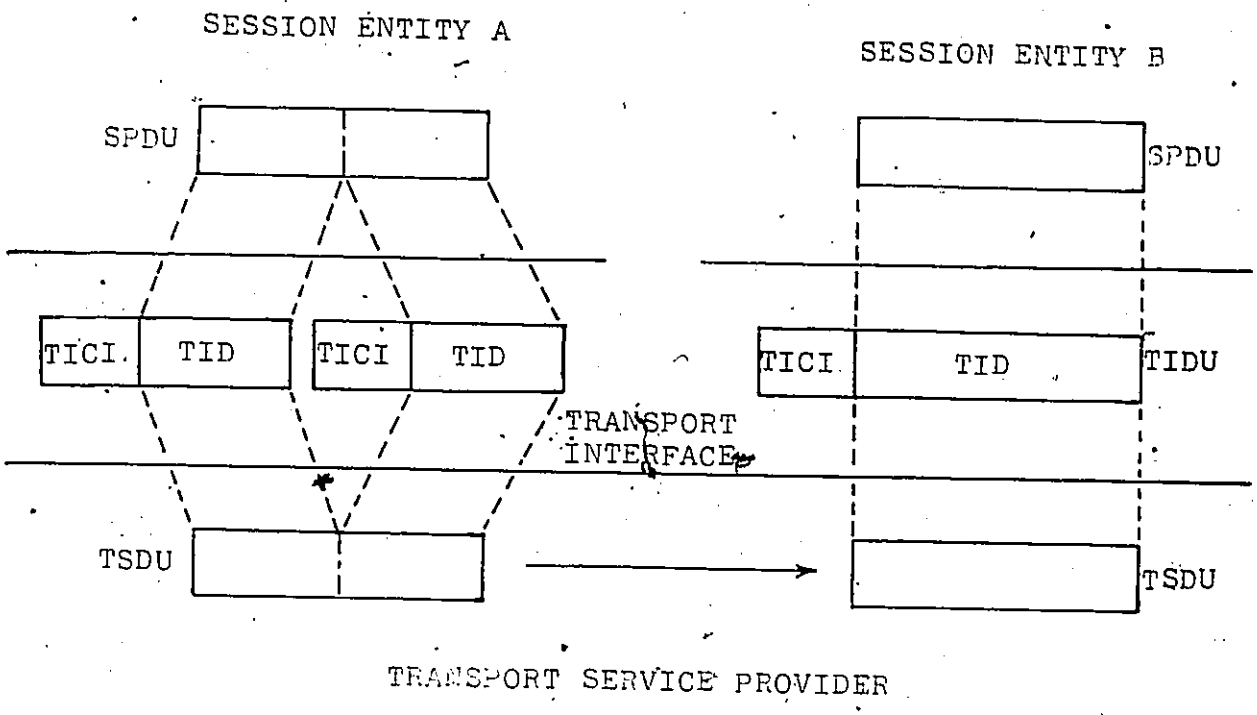


Figure 13: The Mapping Between TSDU and TIDUs

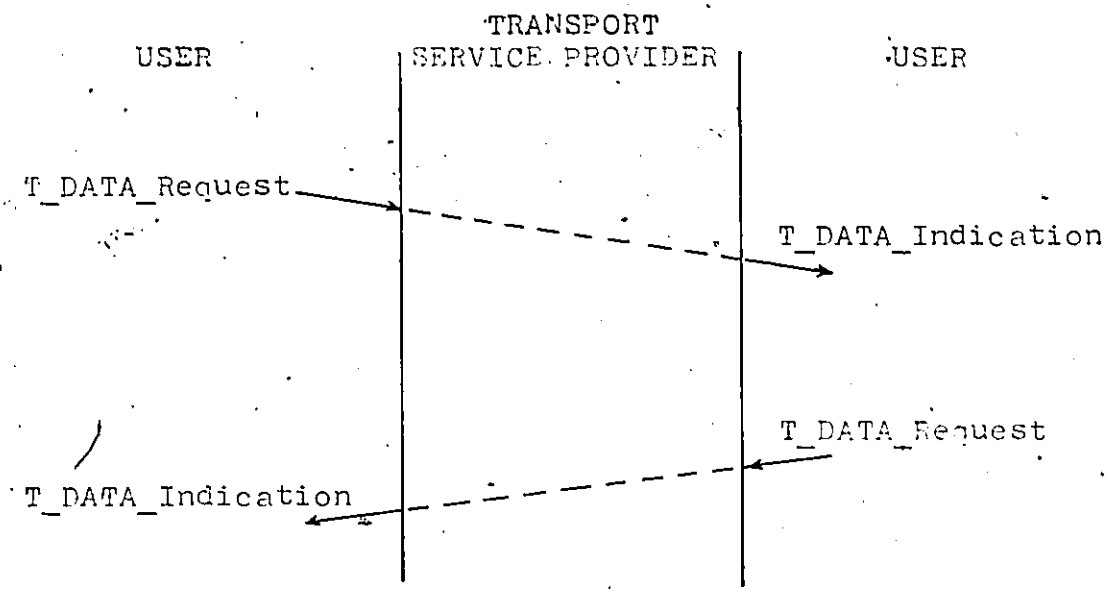


Figure 14: Normal Transfer Service

of a successful delivery of TSDUs is not required in the communication between the transport service provider and its users. Session entities, however, should be notified of the errors from which the transport service provider can not recover. This could be done through the use of the connection termination service which will be discussed in Section 4.4.

4.3.2 EXPEDITED TRANSFER SERVICE

Expedited data transfer is an additional form of data delivery which is not subject to the normal data flow control. The order of executing the involved service primitives is illustrated in Figure 15 . The expedited transfer service is invoked when a session entity sends a T_EX_DATA_Request to the corresponding transport entity along with the parameters: local identifier and user data. As soon as the expedited-TSDU arrives at the receiving station, the local session entity will be informed by the local transport entity through the use of a T_EX_DATA_Indication. The session entity will then acknowledge this indication with a T_EX_DATA_Response. As a result of the acknowledgement, the sending session entity shall receive the T_EX_DATA_Confirm which indicates that the expedited transfer has been completed.

User data, one of the parameters, usually contains session-level commands or responses and is considered to be a complete TSDU. In this design, an expedited-TSDU may contain up to 16 octets of information. It should be noted that T_EX_DATA_Response and T_EX_DATA_Confirm are used for acknowledgement purposes and therefore they convey only local identifiers.

The flow control used in the expedited data transfer is fully controlled by the sender. A session entity is not

allowed to send the next expedited-TSDU until it has explicitly received the acknowledgement (i.e. T_EX_DATA_Confirm) for the previous expedited-TSDU it has transmitted.

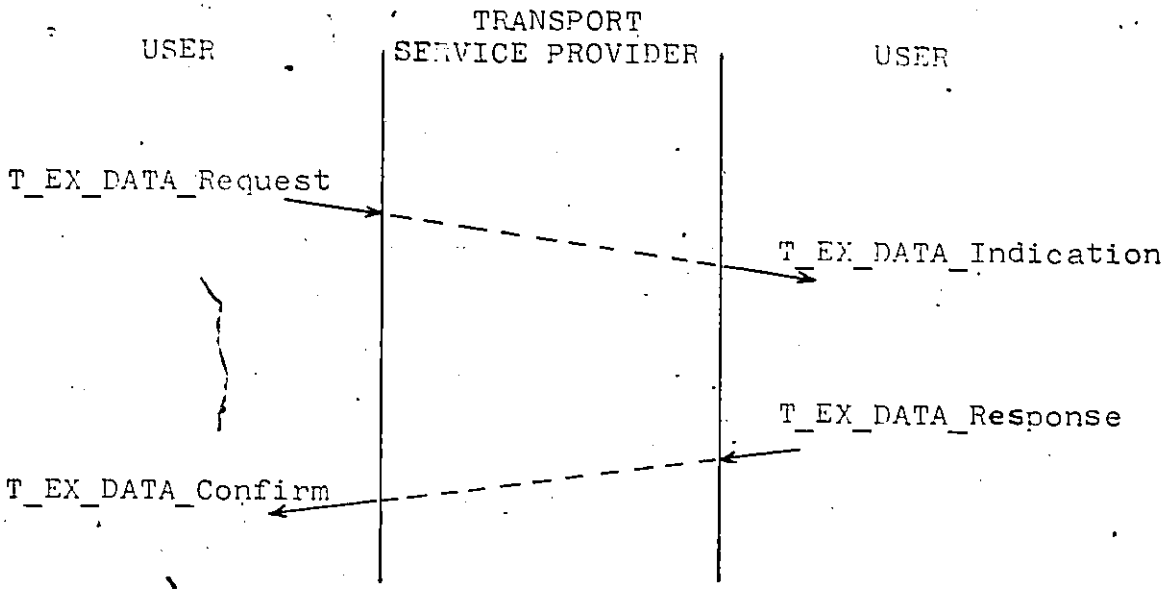


Figure 15: Expedited Transfer Service

4.3.3 PURGE SERVICE

The purge service (available during the data transfer phase only) is requested by transferring T_PURGE_Request across the transport interface. Upon receiving this request, the transport service provider will remove all TSDUs, including expedited-TSDU, on the transport connection and reset it to the predetermined state. Then, the remote session entity will be informed through the use of T_PURGE_Indication as shown in Figure 16 . Of couse, the invocation of purge service will result in the loss of TSDUs. Therefore it is the responsibility of the session entities to provide the necessary recovery mechanism.

The purge service is usually requested in the following situations:

1. When a session entity wants to resynchronize with the corresponding session entity and clear out all TSDUs on the connection.
2. When there is a need to recover from a blocked flow of normal TSDUs on the connection.

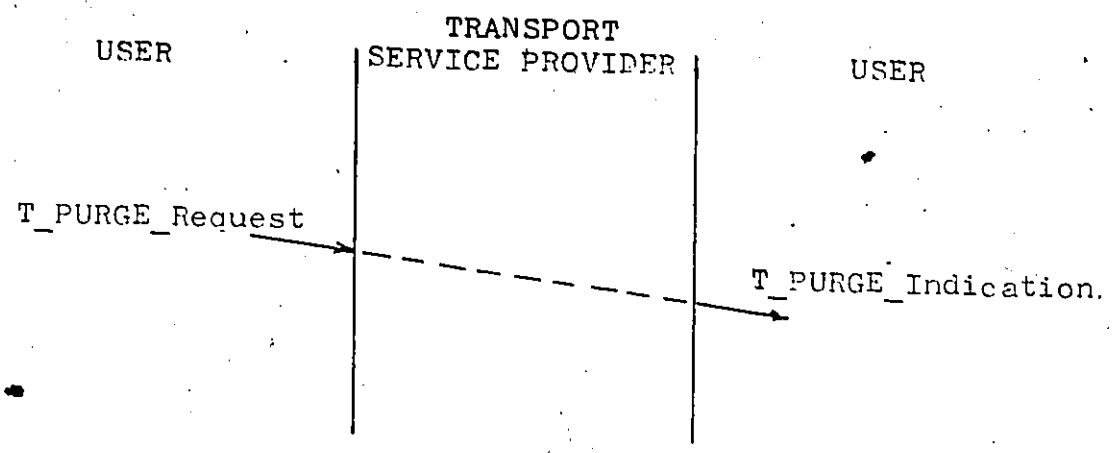


Figure 16: Purge Service

4.4 CONNECTION TERMINATION SERVICE

Termination of a transport connection may be initiated at the transport interface either by the session entity issuing a T_DISCONNECT_Request or by the local transport entity transferring a T_DISCONNECT_Indication across the transport interface. A transport connection may be terminated in any phase. Termination of a transport connection may result in aborting some current data transfers. Therefore, it is the responsibility of the Session Layer to provide the means for achieving orderly termination of a session before initiating termination of the corresponding transport connection.

Figure 17 shows the interaction of the involved service primitives in terminating a transport connection. Either session entity may initiate termination of a transport connection. Upon receipt of a T_DISCONNECT_Request, the corresponding transport entity will release the resource and terminate the connection immediately without making sure that data units submitted prior to the T_DISCONNECT_Request has been delivered to the remote entity.

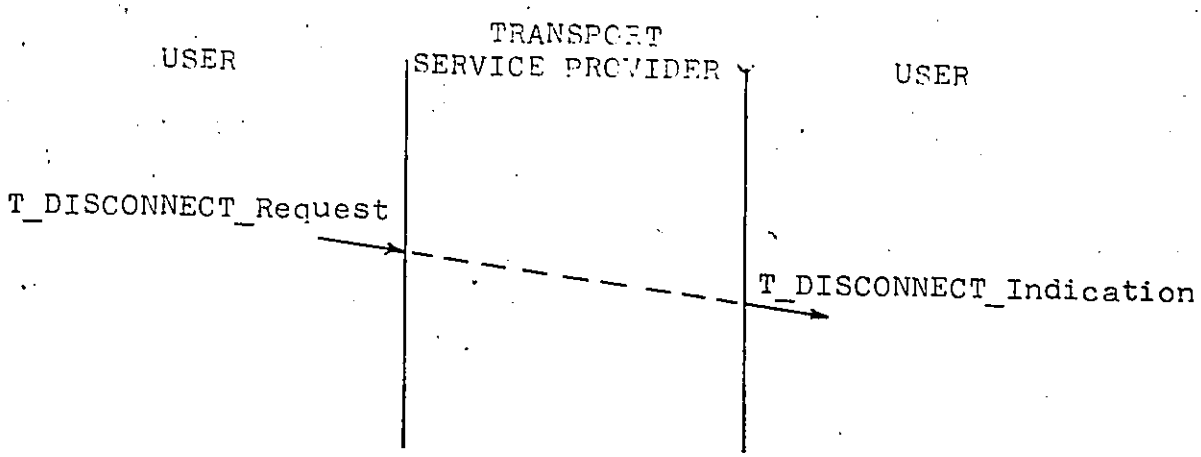


Figure 17: Connection Termination Service

A T_DISCONNECT_Indication will be issued by the local transport entity in the following circumstances:

1. As a result of a T_DISCONNECT_Request at the remote transport interface, indicating that the connection termination is initiated by the remote session entity.
2. Due to an unrecoverable error detected by the service provider, implying that the requested service can no longer be provided.
3. In response to a T_CONNECT_Request because of a lack of resources.

Chapter V

THE TRANSPORT PROTOCOL

5.1 INTRODUCTION

As discussed in the earlier chapter, the Transport Layer shall provide a reliable, sequenced connection-oriented service in which messages are transmitted at two priority levels. This requires that the transport protocol must provide the facilities needed for connection establishment and termination (with error recovery), and data transfer (with error recovery, flow control, segmentation, sequencing, duplicate detection, purge and expedited transfer). The description of a transport protocol consists of the following three elements [26]:

1. Syntax - the structure of commands and responses in either field-formatted (i.e. header bits) or character-string form.
2. Semantics - the set of requests to be issued, action to be performed, and responses to be returned by either party.
3. Timing - the specification of ordering of events.

The execution of the protocol is associated with the exchange of commands and responses between the involved

transport entities. This transport protocol, which is concerned with reliability, is based on the following two assumptions:

1. The executions of commands and responses, and the transfer of data within each individual system are fault free.
2. The order of transmitting the commands and responses at a sender and the order of receiving them at the receiver are the same.

As the protocol is developed, detailed specification must be provided. The specification of the protocol using informal techniques can lead to an ambiguous interpretation, and consequently an incorrect implementation. To avoid misinterpretation, the Formal Description Technique (FDT) developed by the ISO/TC97/SC16/WG1 is used to specify the transport protocol formally. Detailed studies of the FDT are presented in [27-30]; and examples of the use of this technique in specifying computer network protocols and services may be found in [31][32].

In this chapter, the format and meaning of the commands and responses used in the transport protocol are first described. Then, an overview of the transport protocol is given. Finally, a detailed protocol specification is presented in FDT.

5.2 FORMAT OF COMMANDS AND RESPONSES

This section describes the transport protocol commands and responses, a list of which is given in Table 3 .

1. Connection Request (CR) : This command is sent to the called transport entity to indicate the desire of the calling transport entity to establish a transport connection.

STRUCTURE : See Figure 18

CODE : Connection Request code (1110)

CDT : Credit allocation (value indicates the maximum number of TPDU of maximum size that the receiver could accept)

DST-ADDRESS : Transport address of the called entity

DST-ID : 0000 0000

SRC-ADDRESS : Transport address of the calling entity

SRC-ID : Local identifier of the transport connection at SRC-ADDRESS

DATA : Optional user control information (max. size = 80 octets)

2. Connection Confirm (CC) : This response is used to acknowledge the Connection Request if the called transport entity is willing to participate in establishing the transport connection.

STRUCTURE : See Figure 18

CODE : Connection Confirm code (1101)

TABLE 3

Protocol Commands and Responses

COMMANDS AND RESPONSES	ABBREV.	CODE
CONNECTION REQUEST	CR	1110
CONNECTION CONFIRM	CC	1101
DISCONNECT	DISC	0011
DISCONNECT REQUEST	DR	1000
DISCONNECT CONFIRM	DC	1100
DATA TRANSFER	DT	1111
EXPEDITED DATA TRANSFER	EDT	0001
DATA ACKNOWLEDGEMENT	ACK	0110
EXPEDITED DATA ACK	EACK	0010
PURGE REQUEST	PR	1010
PURGE CONFIRM	PC	1001
CREDIT ALLOCATION REQUEST	CAR	0101
CREDIT ALLOCATION ACK	CAA	0111

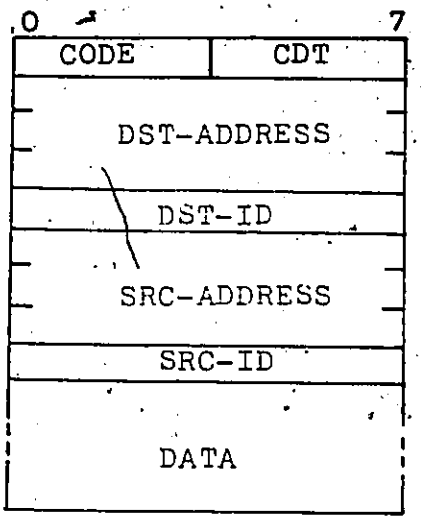


Figure 18: The Structure for Connection Request and Connection Confirm

CDT : Credit allocation

DST-ADDRESS : Transport address of the calling entity

DST-ID : Local identifier of the transport connection at DST-ADDRESS

SCR-ADDRESS : Transport address of the called entity

SRC-ID : Local identifier of the transport connection at SRC-ADDRESS

DATA : Optional user control information (max. size = 80 octets)

3. Disconnect (DISC) : This response is used by the called transport entity to reject the request for establishing a transport connection.

STRUCTURE : See Figure 19

CODE : Disconnect code (0011)

DST-ADDRESS : Transport address of the calling entity

DST-ID : Local identifier of the transport connection at DST-ADDRESS

SCR-ADDRESS : Transport address of the called entity

SRC-ID : 0000 0000

REASON : 128 - Disconnect initiated by remote session entity

129 - Disconnect initiated by remote transport entity

130 - Congestion at remote transport entity

255 - Unknown reason

DATA : Optional user control information (max. size = 80 octets)

4. Disconnect Request (DR) - This command is used to indicate a request for terminating a transport connection.

STRUCTURE : See Figure 19

CODE : Disconnect Request code (1000)

DST-ADDRESS : Transport address of the receiving entity

DST-ID : Local identifier of the transport connection at DST-ADDRESS

SRC-ADDRESS : Transport address of the sending entity

SRC-ID : Local identifier of the transport connection at SRC-ADDRESS

REASON : See DISCONNECT (DISC)

DATA : Optional user control information (max. size = 80 octets)

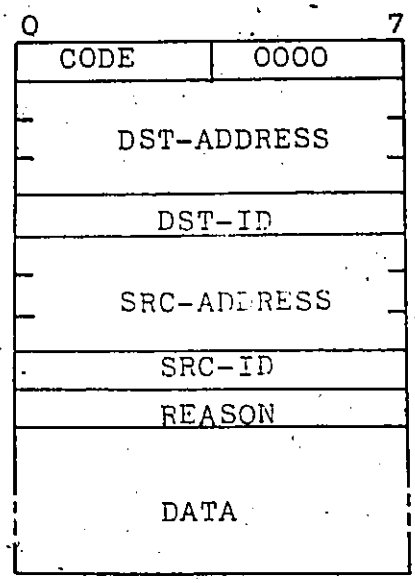


Figure 19: The Structure of Disconnect and Disconnect Request

5. Disconnect Confirm (DC) : This response is used to acknowledge the Disconnect Request.

STRUCTURE : See Figure 20

CODE : Disconnect Confirm code (1100)

DST-ADDRESS :

DST-ID : } See Disconnect Request (DR)

SCR-ADDRESS :

SRC-ID : See Disconnect Request (DR)

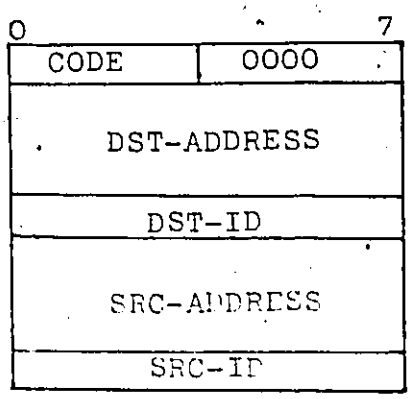


Figure 20: The Structure for Disconnect Confirm

6. Data Transfer (DT) : This command indicates that the transport-protocol- data-unit (TPDU) contains user data.

STRUCTURE : See Figure 21

CODE : Data Transfer code (1111)

DST-ADDRESS : Transport address of the receiving entity

DST-ID : Local identifier of the transport connection at DST-ADDRESS

SEND-SEQUENCE (bits 0-5) : Sequence number of TPDU

X(bit 6) : Request ACK - When set to ONE, requests Data Acknowledgement from the receiving entity

Y(bit 7) : End of TSDU - When set to ONE, indicates the data TPDU contains the last data unit of a TSDU

DATA : Transport user data (max. size = depends on frame size)

7. Expedited Data Transfer (EDT) : This command indicates that the TPDU contains expedited data.

STRUCTURE : See Figure 21

CODE : Expedited Data Transfer code (0001)

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

SEND-SEQUENCE (bits 0-5) : Sequence number of expedited TPDU

X(bit 6) : Not applicable, nominally 0

Y(bit 7) : Not applicable, nominally 0

DATA : Expedited transport user data (max. size = 16 octets)

8. Data Acknowledgement (ACK) : This response is used to acknowledge the Data Transfer.

STRUCTURE : See Figure 22

CODE : Data Acknowledgement code (0110)

CDT : Credit Allocation

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

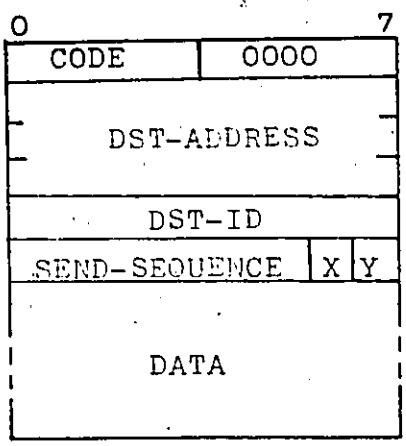


Figure 21: The Structure for DT and EDT

EXPECTED-SEQ-NUMBER (bits 0-5) : The next expected sequence number of TPDU

9. Expedited Data Acknowledgement (EACK) : This response is used to acknowledge the Expedited Data Transfer.

STRUCTURE : See Figure 22

CODE : Expedited Data Acknowledgement code (0010)

CDT : Not applicable, nominally 0

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

EXPECTED-SEQ-NUMBER (bits 0-5) : Not applicable, nominally 0

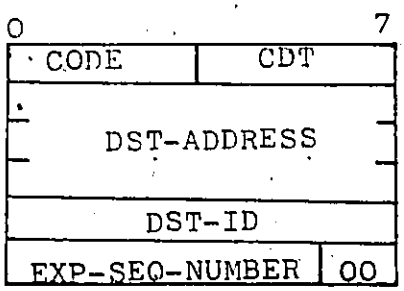


Figure 22: The Structure for ACK and EACK

10. Purge Request (PR) : This command is used to request the remote transport entity to flash all data and expedited data in both directions of the transport connection.

STRUCTURE : See Figure 23

CODE : Purge Request code (1010)

CDT : Credit Allocation

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

11. Purge Confirm (PC) : This response is used to acknowledge the Purge Request.

STRUCTURE : See Figure 23

CODE : Purge Confirm code (1001)

CDT : Credit allocation

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

12. Credit Allocation Request (CAR) : This command is used to request credit allocation from the receiver.

STRUCTURE : See Figure 23

CODE : Credit Allocation code (0101)

CDT : Not applicable, nominally 0

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

13. Credit Allocation Acknowledgement (CAA) : This response is used to acknowledge the Credit Allocation Request.

STRUCTURE : See Figure 23

CODE : Credit Allocation Acknowledgement code (0111)

CDT : Credit allocation

DST-ADDRESS :
DST-ID : } See Data Transfer (DT)

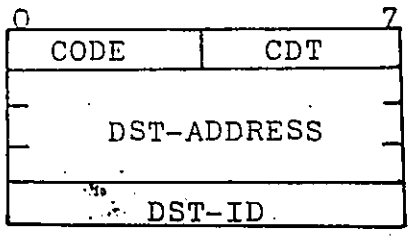


Figure 23: The Structure for PR, PC, CAR and CAA

5.3 AN OVERVIEW OF THE PROTOCOL

The transport protocol provides the means of establishing, maintaining and terminating transport connections by using the commands and responses defined in the previous section. Figure 24 shows the state transition diagram of a transport connection. Each transport connection has five major states : CLOSE, OPEN, OPEN-IN-PROGRESS-CALLING (OIP-CALLING), OPEN-IN-PROGRESS-CALLED (OIP-CALLED), and CLOSE-IN-PROGRESS (CIP). Transitions are specified from a major state to another major state. These transitions depend on an input such as a command, a response or a service primitive. Associated with

each transition is an operation to be executed as part of the transition. During the execution, the transition may initiate an output (see Figure 24).

After a transport entity has transmitted a Connection Request (CR) command to the remote transport entity, it should enter OIP-CALLING state. In this state, the calling transport entity is waiting for a response. Receipt of a Disconnect (DISC) response shall result to a return to the CLOSE state. If a Connection Confirm (CC) response is received, then the transport connection is opened. At the remote transport entity, if the incoming CR is accepted, then a Connection Confirm (CC) response is returned to the calling entity. Otherwise, a DISCONNECT (DISC) is sent.

When a transport connection enters the OPEN state, it is in the data transfer phase. In this phase, a transport connection features a full duplex logical channel. But on each connection, the transmission of data is controlled separately for each direction and is based on the authorization from the receiver. For flow control, the receiver indicates to the sender the number of TPDUs it is ready to receive by means of a 'credit' mechanism. For error recovery, the sending transport entity keeps copies of the transmitted Data TPDUs until it receives a positive acknowledgement which allows copies to be released. A receiver will not send an ACK TPDU unless it has received

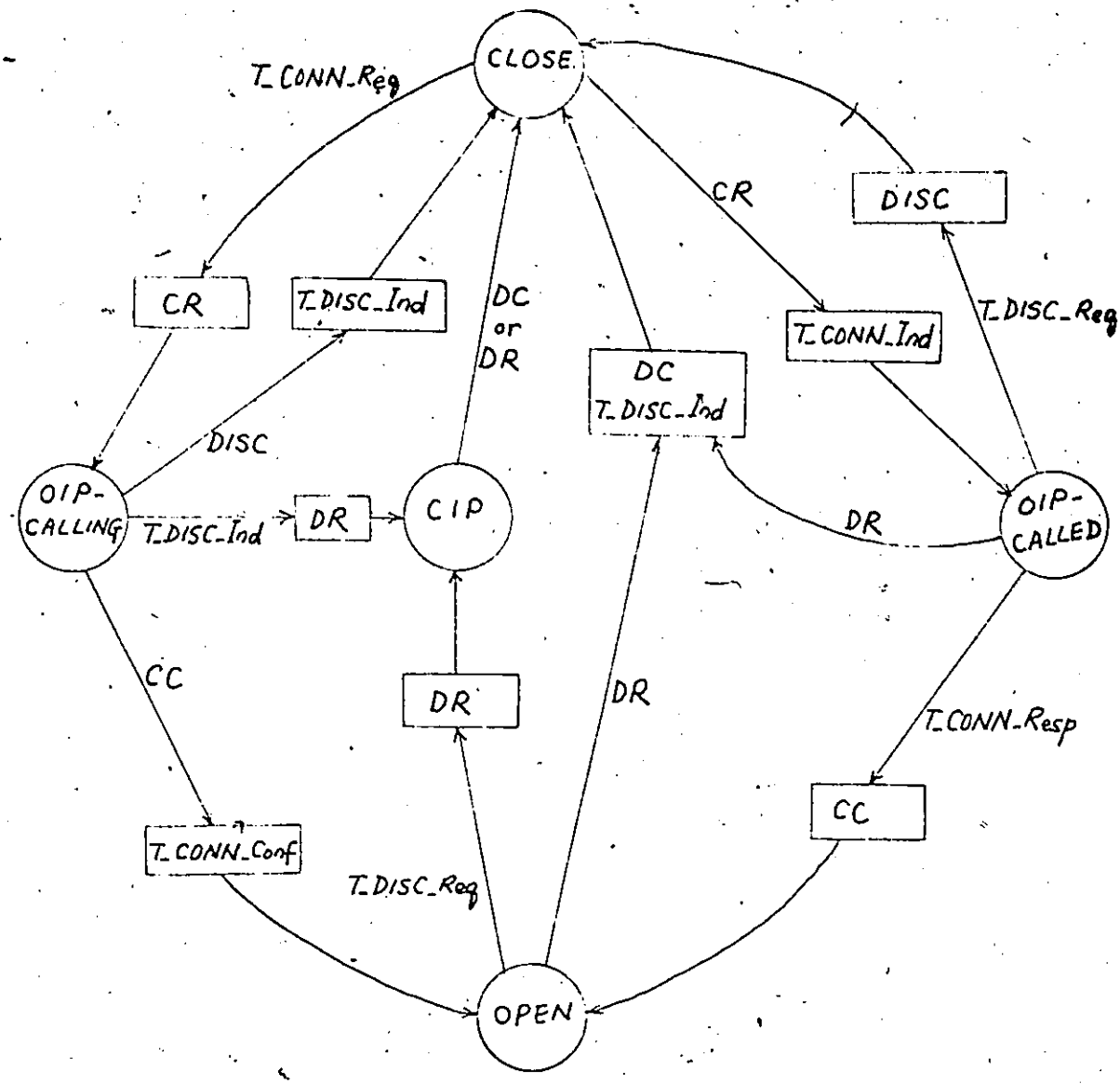


Figure 24: State Diagram of a Transport Connection

from the sender the request for acknowledgement or an out of order Data TPDU. The sender shall make a request for acknowledgement to the receiver only when it has the last credit, or when the end of a TSDU is encountered. Expedited data transfer and purge are allowed when the transport connection is in the data transfer phase. A detailed description of the transport protocol is given in section 5.5 (see Table 4).

TABLE 4

Grouping of Transitions

5.4 OVERALL STRUCTURE OF THE TRANSPORT ENTITY

Most protocols implemented in a given layer of a hierarchical system are so complex that a conceptual subdivision into serveral sublayers or functions is very useful. In this case, each sublayer or function corresponds to a module within each entity executing the protocol. The different modules of an entity are relatively independent of one another.

The overall structure of the transport entity is given in Figure 25 . A transport entity consists of one multiplexing module (to be defined in section 5.5.4), an arbitrary number of protocol modules (to be defined in section 5.5.5), and an arbitrary number of timer modules.

There is a protocol module for each transport-service-access-point (TSAP) identified by a particular transport address. Each protocol module manages an arbitrary number of transport-connection-endpoints (TCEPs). And each of these TCEPs is identified by a local identifier (ID). When this ID is concatenated with the local transport address, a transport-connection-endpoint-identifier (TCEP-ID) is formed. A protocol module identifies a transport connection by referring the ID at the local TCEP and the TCEP-ID at the other end of the connection. Protocol modules execute the transport protocol in managing the individual transport connection. The major states of a transport connection are

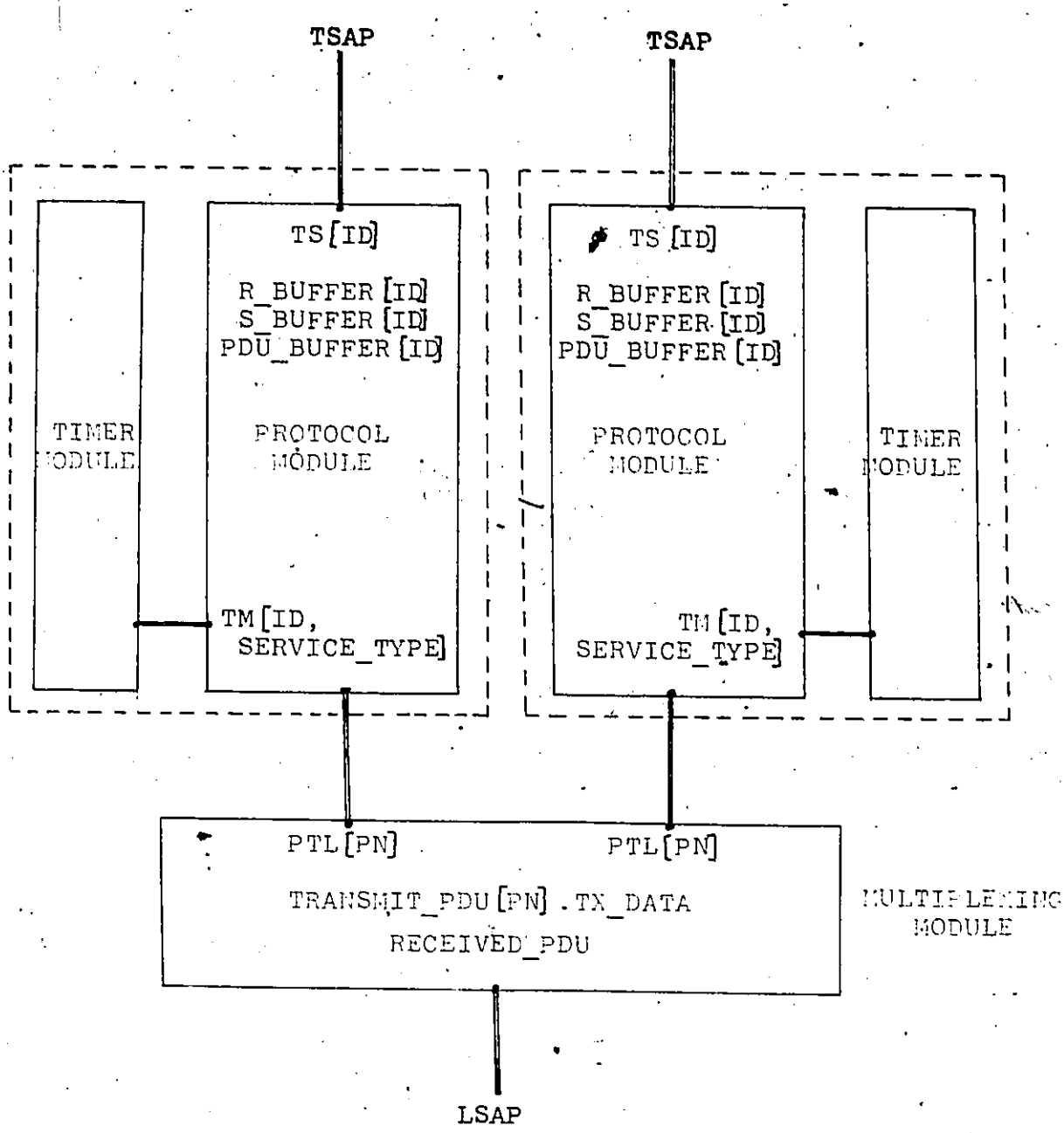


Figure 25: Structure of Transport Entity

specified by the variable 'state'. Within the OPEN state, a number of internal states are also defined. Each of these internal states is specified by a set of context variables. The multiplexing module is required in order to recognize the hierarchical structure of the transport address, so that data received from the link entity could be forwarded to the correct protocol module. The multiplexing module identifies each protocol module by a port number. The mapping of the destination transport address onto the destination host address is also performed by the multiplexing module. In this design, the timer module is assumed to provide two timers for each transport connection. One timer is used for normal data transfer and the other for expedited data transfer.

The channels connecting the different modules (see Figure 25) define the type of interactions that may occur over them. The TSAPs are channels of type 'TS_primitives' (to be defined in section 5.5.1), and the link-service-access-points (LSAPs) are channels of type 'LS_primitives' (to be defined in section 5.5.2). The channels connecting the protocol and multiplexing modules are of type 'control_primitives' (to be defined in section 5.5.3), and lastly, the channels connecting the protocol and timer modules are of type 'timer_primitives' (to be defined in section 5.5.3).

The following steps are involved in the sending of a protocol data unit (PDU):

1. The creation of the PDU by a protocol module. A copy of the PDU is kept in the 'PDU_buffer[ID]' for retransmission in a later time if necessary. This PDU is then sent to the multiplexing module.
2. The multiplexing module receives the PDU and stores it in the appropriate 'transmit_PDU[PN]'. The destination host address will be extracted from the destination transport address by the module.
3. The PDU and the destination host address are then sent to the link entity.

The reception of a PDU proceeds similarly :

1. When a new PDU is received, the multiplexing will extract the local port number, which identifies the particular protocol module, from the destination transport address contained in the PDU. The PDU is then sent to the specified protocol module.
2. The processing of the PDU is done in the protocol module. This processing may lead to an interaction through the TSAP with the session entity.

It is assumed that all interactions involve a rendez-vous technique [28]. The flow control of normal data is described by using channel functions. For example, the function 'TS_user_ready' defined whether the user of the

transport service is ready to receive a certain length of data from the protocol module. The flow control of expedited data is explicitly described by 'T_EX_D_Response' and 'T_EX_D_Confirm' at the transport interface and also by the variables 'EX_D_sent' and 'EX_D_received' defined in the protocol module.

5.5 PROTOCOL SPECIFICATION

5.5.1 SPECIFICATION OF THE TRANSPORT SERVICE

The following specification defines the transport service primitives exchanged over a given transport connection endpoint. The end-to-end properties of the transport service are not defined here. The detailed description of this end-to-end properties can be found in [31][33].

```
type
   T_address_type = 0.. ...;
   local_ID_type = 0.. ...;
   string_of_octets = record
      length : positive_integer;
      contents : array [1.. ...] of 0..255;
      end;
   TS_user_control_data_type,
         (* property : max. length = 80 octets *)
   TS_user_reason_type,
         (* property : max. length = 80 octets *)
   TS_expedited_data_type
         (* property : max. length = 16 octets *)
      = string_of_octets;
   TS_disconnect_reason_type = (
         TS_USER_DISCONNECT, TS_FAIL,
         TS_USER_UNKNOWN, TS_CONGESTION);

(* the notation '...' is used to indicate that the specifier
   is leaving the interpretation to the implementor *)


channel
   TS_primitives (user, provider)
   by user :
```

```
      T_CONNECT_Request (
            called_T_address : T_address_type;
            TS_user_control_data : TS_user_control_data_type);

      T_CONNECT_Response (
            calling_T_address : T_address_type;
            TS_user_control_data : TS_user_control_data_type);

      T_DISCONNECT_Request (
            TS_user_control_data : TS_user_control_data_type);

      T_DATA_Request (TS_user_data : string_of_octets;
                      is_last_fragment_of_TSDU : boolean);

      T_EX_DATA_Request (
            TS_user_data : TS_expedited_data_type);

      T_EX_DATA_Response;

      T_PURGE_Request;

   by provider:
      T_CONNECT_Indication (
            calling_T_address : T_address_type;
            TS_user_control_data : TS_user_control_data_type);

      T_CONNECT_Confirm (
            called_T_address : T_address_type;
            TS_user_control_data : TS_user_control_data_type);

      T_DISCONNECT_Indication (
            TS_disconnect_reason : TS_disconnect_reason_type;
            TS_user_control_data : TS_user_control_data_type);

      T_DATA_Indication (TS_user_data : string_of_octets;
                         is_last_fragment_of_TSDU : boolean);

      T_EX_DATA_Indication (
            TS_user_data : TS_expedited_data_type);

      T_EX_DATA_Confirm;

      T_PURGE_Indication;

   by user : function TS_user_ready (
         data_length : positive_integer) : boolean;
   by provider : function TS_ready (
         data_length : positive_integer) : boolean;
(* The function 'TS_user_ready' determines whether the
   session entity is ready to receive data of length
   'data_length'.  And the function 'TS_ready' determines
   whether the PTL module can accept another data of length
   'data_length' from the user. *)
```

5.5.2 SPECIFICATION OF THE LINK SERVICE

The following specification defines the link service primitives exchanged over a link-service-access-point. The type of service provided by the Link Layer is in the form of connectionless. The Multiplexing module requests link service by sending 'frame_transmit' and the associate parameters to the local data link entity. It is assumed that the link entity knows the local host address. A link entity indicates a received frame to the user through the use of 'frame_receive'. The 'received_frame_status' indicates whether or not the received frame contains error. A similar specification of this type of service can be found in [19].

```
type
   host_address_type = ...;
         (* addresses used to identify transport entities *)
   link_user_data_type = record
      length : positive_integer;
      content : array [1.. ...] of 0..255;

channel
   LS_primitives (user, provider)
   by user :
      frame_transmit (dst_host_address : host_address_type;
                      LS_user_data : link_user_data_type);
   by provider :
      frame_receive (LS_user_data : link_user_data_type;
                     received_frame_status : boolean);
   by provider : function LS_ready : boolean;

(* This function determines whether the link entity can accept
   another data from its user *)
```

5.5.3 DECLARATION INTERNAL TO THE TRANSPORT LAYER

```
const
   max_PDU_size = ...;
         (* max. size of protocol data unit in octets *)
   max_count = ...; (* max. number of retransmissions *)

type (* the type and interaction of transport and link service
        specifications are used *)
   seq_number_type = 0..63;
   credit_type = 0..31;
         (* value indicates the max. number of TPDU of
            max_PDU_size that the receiver could accept *)
   port_number_type = ...;
   service_type = NRL, EXP ;
         (* normal and expedited data transfer
            respectively *)

   disconnect_reason_type = (
      128 (* disconnect initiated by session entity *),
      129 (* disconnect initiated by transport entity *),
      130 (* congestion at remote transport entity *),
      255 (* unknown reason *));

   TPDU_code_type = (CR, CC, DISC, DR, DC, DT, ACK, EDT,
                     EACK, PR, PC, CAR, CAA);

   TPDU_information = record
      CDT : credit_type; (* used for CR, CC, ACK, PR, PC, CAA *)
      dst_addr : T_address_type; (* used for all TPDU codes *)
      dst_ID : local_ID_type; (* used for all TPDU codes *)
      src_addr : T_address_type; (* used for CR, CC, DISC,
                                    DR, DC *)
      src_ID : local_ID_type; (* used for CR, CC, DISC,
                                 DR, DC *)
      user_data : strings_of_octets; (* used for CR, CC,
                                        DISC, DR, DT, EDT *)
      case kind : TPDU_code_type of
         CR, CC : ;
         DISC, DR : (reason : disconnect_reason_type);
         DC : ;
         DT : (send_sequence : seq_number_type;
               request_ACK : boolean;
               end_of_TSDU : boolean);
         ACK, EACK : (expected_seq_number : seq_number_type);
         EDT : (send_sequence : seq_number_type);
         PR, PC, CAA, CAR : ;
         end;


(* data buffer : abstract data type *)

type
```

```
   data_buffer = ...;
   buffer_type = array [local_ID_type] of data_buffer;

function enough_space (
      b : buffer_type, length : positive_interger) : boolean;
   (* Indicates whether 'b' has enough space for 'length'
      octets of data *)

function length_available (
      b : buffer_type) : positive_interger;
   (* Indicates the length of data unit in 'b' *)

function is_end_of_DU (b : buffer_type) : boolean;
   (* Indicates whether the data in 'b' includes a complete
      data unit *)

function get_next_fragment (
      b : buffer_type, length : positive_interger) :
      string_of_octets;
   (* Returns a data unit (or part there of) of maximum
      length 'length' which was stored in 'b' *)

function determine_CDT (b : buffer_type) : credit_type;
   (* Indicates the number of data units each of max_PDU_size
      could be accepted by 'b' *)

procedure append (b : buffer_size, f : string_of_octets,
                  end_of_DU : boolean);
   (* Appends a string of octets onto 'b', including an
      indication whether the string terminates a TSDU *)

procedure store_PDU (b : buffer_type, s : seq_number_type,
                     p : TPDU_information);
   (* Decodes 'p' into string of octets and identifies this
      string of octets by 's' and store it in 'b' *);

procedure retrieve_PDU (b : buffer_type, s : seq_number_type
                        p : TPDU_information);
   (* Retrieve a string of octets, which is identified by
      's', from 'b' and encodes this string of octets into
      'p' *)

procedure clear_PDU_buffer (
      b : buffer_type, s : seq_number_type);
   (* Deletes the strings of octets, which are identified by
      's' and all numbers smaller than 's', from 'b' *)

(* end of buffer declaration *)


channel
   control_primitives (PTL, MUX)
   by PTL, MUX :
      forward (PDU : TPDU_information);
```

```
   by MUX : function ready_for_sending : boolean;
   by MUX : function received_PDU_size : positive_interger;
   end (* control_primitives *);

(* The above channel declaration defines the interactions
   between the PTL and the Multiplexing modules. The
   function 'ready_for_sending' indicates whether the
   Multiplexing module is ready to accept another data unit
   from the PTL module. And the function
   'received_PDU_size' indicates the size of a received PDU
   to the PTL module. *)


channel
   timer_primitives (user, provider)
   by user :
      set_timer;
      stop_timer;
   by provider :
      time_out;
   end (* timer_primitives *);

(* The above channel declaration defines the interactions
   between the PTL and the timer modules. The timer delay
   is assumed to be constant. *)
```

5.5.4 SPECIFICATION OF THE MULTIPLEXING MODULE

```
module Multiplexing (
      PTL : array [port_number_type] of control
      _primitives (MUX); LINK : LS_primitives (user));

const local_host_addr : ...;
      (* used to identify local transport entity *)

var
   PN : port_number_type;
   received_PDU : TPDU_information;
         (* temporary variable for incoming PDU *)
   received_PDU_size : positive_integer;
   transmit_PDU : array [port_number_type] of record
      full : boolean;
      tx_data : link_user_data_type;
            (* temporary variable for outgoing PDU *)
      dst_host_addr : host_addr_type;
            (* used to identify a remote transport entity *)
      end;
```

```
(* Definition of Interface Function *)

any PN : port_number_type do
   PTL[PN].ready_for_sending := not transmit_PDU[PN].full;
   (* Defines whether the Multiplexing module can accept
another PDU from a Protocol module *)


any PN : port_number_type do
   PTL[PN].received_PDU_size := received_PDU_size;
   (* Makes the local variable 'received_PDU_size' of the
      Multiplexing module available to the PTL module *)



(* Definition of Local Functions and Procedures *)


function extract_host_addr (
      transport_address : T_address_type) :
      host_address_type;
   begin ... end;
   (* Extracts a host address from a given transport address *)

function extract_port_number (
      transport_address : T_address_type) :
      port_number_type;
   begin ... end;
   (* Extracts the port number from a given transport address *)

procedure decode_data (PDU : TPDU_information;
                       tx_data : link_user_data_type);
   (* This procedure decodes the 'PDU' into link_user_
      data_type and stores in 'tx_data' *)

procedure encode_data (LS_user_data : link_user_data_type;
                       received_PDU : TPDU_information);
   (* This    procedure    encodes    'LS_user_data'    into
      TPDU_information and assigns  the corresponding values
      to the 'received_PDU' *)



(* Initialization *)


initialize begin
   for all PN in port_number_type do
   transmit_PDU [PN].full := false;
   end;
```

```
(* Transistions *)

(* Handling Request From The Protocol Module *)
(* Receive a data unit from a PTL module *)

when PTL[PN].forward (* PDU *)
   with transmit_PDU[PN] do
   begin
      full := true;
      decode_data (PDU, tx_data);
      dst_host_addr := extract_host_addr (PDU.dst_addr);
      end; end when;


(* Sending a data unit to link entity *)

provided LS_ready = true and transmit_PDU[PN].full = true
   with transmit_PDU[PN] do
   begin
      full := false;
      out LINK.frame_transmit (dst_host_addr, tx_data);
      end; end provided;


(* Handling an Incoming Data From Link Layer *)

when LINK.frame_receive (* LS_user_data,
                           received_frame_status *)
   provided received_frame_status = true
                        (* i.e. frame contains no error *)
      begin
      encode_data (LS_user_data, received_PDU);
      if local_host_addr = extract_host_addr (
                           received_PDU.dst_addr)
         then begin
              received_PDU_size := LS_user_data.length;
              PN := extract_port_number (
                    received_PDU.dst_addr);
              if (/ there exists a protocol module 'PN' /)
                 then out PTL[PN].forward (received_PDU);
                 else (/ discards received_PDU /);
              end;
         else (/ discards received_PDU /);
      end; end provided;
      (* This transition describes the receiving of a correct
         frame, the encoding of the received frame and the
         determining of the protocol module where the data unit
         is to be sent *)
```

```
    provided received_frame_status = false (*
          i.e. frame contains error *)
       begin
       (/ discards LS_user_data /);
       end; end provided; end when;

end Multiplexing;
```

5.5.5 SPECIFICATION OF THE PROTOCOL MODULE

```
module Protocol (TS : array [local_ID_type] of
                      TS_primitives (provider);
                 MUX : control_primitives (PTL);
                 TM : array [local_ID_type, service_type]
                      of timer_primitives (user));

const local_T_addr = ...;

var
   module_state : entity_ready, entity_not_ready;
   S_buffer,
   R_buffer,
   PDU_buffer : data_buffer_type;
   IDU_length : positive_integer; (* size of interface_
                                     data_unit in octets *)
   TC : array [local_ID_type] of record
      state : (open, open_in_progress_calling, open_in
               _progress_called, close_in_progress, close);
      RS, (* receive sequence number : value indicates the
             next expected PDU sequence number *)
      SS, (* send sequence number : value indicates the
             latest transmitted PDU sequence number *)
      retransmit_SS, (* SS for the retransmitted PDU *)
      expedited_SS, (* send sequence number for expedited
                       data *)
      expedited_RS : seq_number_type; (* receive
                     sequence number for expedited data *)
      S_credit : credit_type; (* credit for sending TPDU *)
      retransmit_count : positive_integer; (* number of
                         retransmissions *)
      peer_addr : T_address_type; (* transport address of
                                     the remote entity *)
      peer_ID : local_ID_type; (* local ID at the peer addr *)
      ACK_TPDU_full, (* true if an ACK TPDU is waiting to
                        be sent *)
```

```
    wait_for_CAA, (* true if waiting for credit
                     allocation acknowledgement *)
    retransmit_PDU, (* true if presently retransmitting
                       TPDU *)
    wait_for_ACK, (* true if presently waiting for
                     acknowledgement *)
    purge_request, (* true if purge operation is not
                      finished *)
    EX_D_sent, (* true if expedited data is sent *)
    EX_D_received : boolean; (* true if expedited data
                                is received *)
    TPDU,
    EX_TPDU, (* used for expedited data only *)
    ACK_TPDU : TPDU_information; (*
                used for data acknowledgement only *)
    end;



(* Definition of Interface Function *)


any ID : local_ID_type do
   TS[ID].TS_ready (user_data.length) := enough_space (
                    S_buffer[ID], user_data.length);
(* Define whether Protocol module can accept another data
   unit from session entity *)



(* Definition of Local Function and Protocol *)


function ID_assign (PDU.src_addr : T_addr_type;
                    PDU.src_ID : local_ID_type) : boolean;
   begin
   for all ID : local_ID_type do
   if TC[ID].(peer_addr, peer_ID) = PDU.(src_addr, src_ID)
      then ID_assign := true;
      else ID_assign := false;
   end;
(* Determine whether a local identifier 'ID' of the
   receiving entity has been assigned to the received PDU's
   source address and source ID *)

function find_ID (PDU.src_addr : T_address_type;
                  PDU.src_ID : local_ID_type) : boolean;
   begin
   for all ID : local_ID_type do
   if TC[ID].(peer_addr, peer_ID) = PDU.(src_addr, src_ID)
      then find_ID := ID; end;
(* Get the local identifier that has been assigned to the
```

```
   PDU's source address and source identifier *)

function TCEP_match (PDU.dst_ID : local_ID_type) : boolean;
   begin
   if PDU.dst_ID <> 0 and
      TC[PDU.dst_ID].(peer_addr, peer_ID) = PDU.(src_addr,
                                                 src_ID)
      then TCEP_match := true;
      else TCEP_match := false; end;
(* This function determines whether the source address and ID
   of the received PDU match the peer address and ID at the
   receiving entity *)

function determine_reason (PDU.reason : disconnect_reason_type) :
                           TS_disconnect_reason_type;
   begin case PDU.reason of
      128 : determine_reason := TS_USER_DISCONNECT;
      129, 255 : determine_reason := TS_FAIL;
      130 : determine_reason := TS_CONGESTION;
   end; end;
(* This function makes the relation between the 'reason
   type' used in the TPDU's and the disconnect reason
   indicated to the user of the transport service *)

procedure clear_all_buffer (ID : local_ID_type);
   begin ... end;
(* This procedure clears the following buffers identified
   by 'ID' : S_buffer, R_buffer and PDU_buffer *)



(* Initialization *)


initialize begin
   for all ID in local_ID_type do TC[ID].state := close;
   for all ID in local_ID_type do clear_all_buffer(ID);



(* Transitions *)

(* Determines the module states *)

provided (/ TC[ID].state <> close, for all ID except ID = 0 /)
   from entity_ready to entity_not_ready
   begin end; end provided;

provided (/ TC[ID].state = close, for any ID except ID = 0 /)
   from entity_not_ready to entity_ready
   begin end; end provided;
```

```
(* Connection Establishment *)

(* Handling a Connection Request from Session Entity *)

when TS[ID].T_CONNECT_Request (* called_T_addr,
                                 user_control_data *)
   provided ID <> 0 and MUX.ready_for_sending
   with TC[ID] do
   from close to open_in_progress_calling
   begin
      RS := 1;
      SS := 0;
      expedited_RS := 0;
      expedited_SS := 0;
      retransmit_count := 0;
      peer_addr := called_T_addr;
      peer_ID := 0;
      EX_D_received := false;
      EX_D_sent := false;
      retransmit_PDU := false;
      wait_for_ACK := false;
      ACK_TPDU_full := false;
      purge_request := false;
      wait_for_CAA := false;
      with TPDU do begin
         kind := CR;
         CDT := determine_CDT (R_buffer[ID]);
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         user_data := user_control_data;
         end;
      store_PDU (PDU_buffer[ID], SS, TPDU); (* store for
                                               retransmission *)
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end provided;

   provided ID <> 0
   with TC[ID] do
   from (/ any state except close /) to same
   begin
      out TS[ID].T_DISCONNECT_Indication (TS_CONGESTION,
                                          ... (* dummy *));
      end; end provided; end when;

(* Handling a Connection Request from Peer Entity *)

when MUX.forward (* PDU *)
```

```
provided PDU.kind = CR and PDU.dst_ID = 0
         and ID_assign (PDU.src_addr, PDU.src_ID) = false
         and (/ for all ID' do
                if TC[ID'].state = close
                   then ID = ID',
                        i.e. transport entity ready /)
   with TC[ID] do
   from close to open_in_progress_called
   begin
      RS := 1;
      SS := 0;
      expedited_RS := 0;
      expedited_SS := 0;
      retransmit_count := 0;
      S_credit := PDU.CDT;
      peer_addr := PDU.src_addr;
      peer_ID := PDU.src_ID;
      EX_D_received := false;
      EX_D_sent := false;
      retransmit_PDU := false;
      wait_for_ACK := false;
      ACK_TPDU_full := false;
      purge_request := false;
      wait_for_CAA := false;
      out TS[ID].T_CONNECT_Indication (peer_addr,
                                       PDU.user_data);
      end; end provided;
   (* The above transition is executed when the connection
      has not been opened and the transport entity is ready
      to accept the request *)

provided PDU_kind = CR and PDU.dst_ID = 0
                       and MUX.ready_for_sending
   from entity_not_ready to same
   with TC[0].TPDU do begin
      kind := DISC;
      dst_addr := PDU.src_addr;
      dst_ID := PDU.src_ID;
      src_addr := local_T_addr;
      src_ID := 0;
      reason := 130; (* indicates remote congestion *)
      user_data := ...; (* dummy *)
      out MUX.forward (TPDU);
      end; end provided;
   (* The above transition is executed when transport
      entity is not ready to accept the request *)

provided PDU.kind = CR and PDU.dst_ID = 0
         and ID_assign (PDU.src_addr, PDU.src_ID) = true
   with TC[find_ID(PDU.src_addr, PDU.src_ID)] do
   from open_in_progress_called, close_in_progress to same
   begin  (/ ignores PDU /) end; end provided;
(* The above transition describes the receiving of CR in the
   process of opening the connection, or in the state of
```

```
      aborting the connection after it has been opened *)

   provided PDU.kind = CR and PDU.dst_ID = 0
            and ID_assign (PDU.src_addr, PDU.src_ID) = true
            and MUX.ready_for_sending
      with TC[find_ID (PDU.src_addr, PDU.src_ID)] do
      from open to same
      begin
      ID := find_ID (PDU.src_addr, PDU.src_ID);
      retrieve_PDU (PDU_buffer[ID], 0, TPDU);
      if SS <> 0 then begin (* check if data has been sent *)
         out TM[ID, NRL].stop_timer;
         wait_for_ACK := false;
         purge_request := false;
         EX_D_sent := false;
         ACK_TPDU_full := false;
         retransmit_PDU := true;
         retransmit_SS := 0;
         S_credit := PDU.CDT;
         end;
      out MUX.forward (TPDU);
      end; end provided; end when;
   (* The above transition describes the reception of CR
      after the connection has been opened *)


(* Handling a Connection Response from Session Entity *)

when TS[ID].T_CONNECT_Response (* calling_T_address,
                                  user_control_data *)
   provided MUX.ready_for_sending
   with TC[ID] do
   from open_in_progress_called to open
   begin
   with TPDU do begin
      kind := CC;
      CDT := determine_CDT (R_buffer[ID]);
      dst_addr := peer_addr;
      dst_ID := peer_ID;
      src_addr := local_T_addr;
      src_ID := ID;
      user_data := user_control_data;
   store_PDU (PDU_buffer[ID], SS, TPDU);
   out MUX.forward (TPDU);
   end; end when;

(* Handling a Connection Confirm from Peer Entity *)

when MUX.forward (* PDU *)
provided PDU.kind = CC
         and TC[PDU.dst_ID].peer_addr = PDU.src_addr
         and PDU.(dst_ID, src_ID) <> 0
with TC[PDU.dst_ID] do
   from open_in_progress_calling to open
```

```
      begin
      out TM[PDU.dst_ID, NRL].stop_timer;
      S_credit := PDU.CDT;
      peer_ID := PDU.src_ID;
      out TS[PDU.dst_ID].T_CONNECT_Confirm (
          PDU.src_addr, PDU.user_data);
      end; end from;

   from open, close_in_progress to same
      begin (/ ignores PDU /); end; end from; end when;



(* Connection Termination *)


(* Handling a Disconnect Request from Session Entity *)

when TS[ID].T_DISCONNECT_Request (* user_control_data *)
provided MUX.ready_for_sending
with TC[ID] do
   from open_in_progress_called to close
      begin
      with TPDU do begin
         kind := DISC;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         reason := 128; (*disconnect initiated by user *)
         user_data := user_control_data;
         end;
      out MUX.forward (TPDU);
      end; end from;
      (* The above transition  is executed when the  user of
         the called transport entity  rejects the connection
         request *)

   from open_in_progress_calling to close_in_progress
      begin
      out TM[ID, NRL].stop_timer;
      with TPDU do begin
         kind := DR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         reason := 128; (* disconnect initiated by user *)
         user_data := user_control_data;
         end;
      store_PDU (PDU_buffer[ID], SS, TPDU);
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
```

```
      end; end from;
      (* The above transition is executed when the user of the
         calling transport entity aborts the connection
         establishment before the connection is opened *)

   from open to close_in_progress
      begin
      out TM[ID, NRL].stop_timer;
      with TPDU do begin
         kind := DR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         reason := 128; (* disconnect initiated by user *)
         user_data := user_control_data;
         end;
      store_PDU (PDU_buffer[ID], SS, TPDU);
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end from; end when;
      (* The above transition describes the request from a
         session entity to close a connection. It is
         assumed that a Session Layer should provide the
         means for achieving ordering termination of a
         session before initiating termination of the
         corresponding transport connection *)


(* Disconnect Initiated by Transport Entity *)

any ID : local_ID_type do
provided MUX.ready_for_sending and
         (/ transport entity not able to continue providing
            service /)
with TC[ID] do
   from open, open_in_progress_calling to close_in_progress
      begin
      out TM[ID, NRL].stop_timer;
      out TS[ID].T_DISCONNECT_Indication (
            TS_FAIL,...,(* dummy *));
      with TPDU do begin
         kind := DR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         reason := 129; (* disconnect initiated by
                           transport entity *)
         user_data := ...; (* dummy *)
         end;
      store_PDU (PDU_buffer[ID], SS, TPDU);
      retransmit_count := 0;
      out TM[ID, NRL].set_timer;
```

```
      out MUX.forward (TPDU);
      end; end from;

   from open_in_progress_called to close
      begin
      out TS[ID].T_DISCONNECT_Indication (TS_FAIL,...
                                          (* dummy *));
      with TPDU do begin
         kind := DISC;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := ID;
         reason := 129;
         user_data := ...; (* dummy *)
         end;
      out MUX.forward (TPDU);
      end; end from;

   from close_in_progress, close to same
      begin end; end from; end any;


(* Handling a Disconnect from the Peer Entity *)

when MUX.forward (* PDU *)
with TC[PDU.dst_ID] do
   provided PDU.kind = DISC
            and TCEP_match (PDU.dst_ID) = true
            and retransmit_count = 0
      from open_in_progress_calling to close
      begin
         out TM[PDU.dst_ID, NRL].stop_timer;
         peer_addr := 0;
         out TS[PDU.dst_ID].T_DISCONNECT_Indication (
             determine_reason (PDU.reason), PDU.user_data);
         end; end provided;
      (* when receiving a DISC before the retransmission of
         CR, the above transition is executed *)

   provided PDU.kind = DISC and retransmit_count <> 0
            and TCEP_match (PDU.dst_ID) = true
      from open_in_progress_calling to close_in_progress
      begin
      out TM[PDU.dst_ID, NRL].stop_timer;
      out TS[PDU.dst_ID].T_DISCONNECT_Indication (
          determine_reason (PDU.reason), PDU.user_data);
      with TPDU do begin
         kind := DR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := PDU.dst_ID;
         reason := 255; (* unknown reason *)
```

```
            user_data := ...; (* dummy *)
            end;
         store_PDU (PDU_buffer[PDU.dst_ID], SS, TPDU];
         retransmit_count := 0;
         out TM[PDU.dst_ID, NRL].set_timer;
         out MUX.forward (TPDU);
         end; end provided;
         (* When receiving a DISC after the retransmission of CR,
            this transition is executed *)

      provided PDU.kind = DISC
         from close_in_progress to same
         begin (/ ignores the PDU /); end; end provided;
                                          end when;
         (* The receiving of DISC in these cases has no effect
            on the entity *)


   (* Handling a Disconnect Request from the Peer Entity *)

   when MUX.forward (* PDU *)
      provided PDU.kind = DR and PDU.dst_ID = 0
               and ID_assign (PDU.src_addr, PDU.src_ID) = true
               and MUX.ready_for_sending
      with TC[find_ID(PDU.src_addr, PDU.src_ID)] do
         from open, open_in_progress_called to close
         begin
         ID := find_ID (PDU.src_addr, PDU.src_ID);
         out TM[ID, NRL].stop_timer;
         with TPDU do begin
            kind := DC;
            dst_addr := peer_addr;
            dst_ID := peer_ID;
            src_addr := local_T_addr;
            src_ID := 0;
            end;
         peer_addr := 0;
         peer_ID := 0;
         out TS[ID].T_DISCONNECT_Indication (
            determine_reason (PDU.reason), PDU.user_data);
         out MUX.forward (TPDU);
         end; end from;

         from close_in_progress to close
         begin
         ID := find_ID (PDU.src_addr, PDU.src_ID);
         out TM[ID, NRL].stop_timer;
         with TPDU do begin
            kind := DC;
            dst_addr := peer_addr;
            dst_ID := peer_ID;
            src_addr := local_T_addr;
            src_ID := 0;
            end;
```

```
   peer_addr := 0;
   peer_ID := 0;
   out MUX.forward (TPDU);
   end; end from; end provided;

provided PDU.kind = DR and TCEP_match (PDU.dst_ID) = true
         and MUX.ready_for_sending
with TC[PDU.dst_ID] do
   from open to close
      begin
      with TPDU do begin
         kind := DC;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         src_addr := local_T_addr;
         src_ID := PDU.dst_ID;
         end;
      peer_addr := 0;
      peer_ID := 0;
      out TS[PDU.dst_ID].T_DISCONNECTION_Indication (
         .determine_reason (PDU.reason), PDU.user_data);
      out MUX.forward (TPDU);
      end; end from;
      (* The above transition is executed when one of the
         entities initiates a termination on a connection
         which has been opened *)

   from close_in_progress to close
      begin
      out TM[PDU.dst_ID, NRL].stop_timer;
      peer_addr := 0;
      peer_ID := 0;
      end; end from; end provided;
      (* The above transition is executed when both entities
         initiate connection termination at the same time *)

provided PDU.kind = DR and
         ((PDU.dst_ID = 0 and
          ID_assign (PDU.src_addr, PDU.src_ID) = false) or
         (TCEP_match (PDU.dst_ID) = false)) and
         MUX.ready_for_sending
   from entity_ready, entity_not_ready to same
   with TC[0].TPDU do
   begin
      kind := DC;
      dst_addr := PDU.src_addr;
      dst_ID := PDU.src_ID;
      src_addr := PDU.dst_addr;
      src_ID := PDU.dst_ID;
      out MUX.forward (TPDU);
      end; end provided; end when;
      (* The above transition is executed when receiving
         a DR on a connection which has been closed *)
```

```
(* Handling a Disconnect Confirm from Peer Entity *)

when MUX.forward (* PDU *)
   provided PDU.kind = DC and TCEP_match (PDU.dst_ID) = true
   with TC[PDU.dst_ID] do
   from close_to_progress to close
      begin
      out TM[PDU.dst_ID, NRL].stop_timer;
      peer_addr := 0;
      peer_ID := 0;
      end; end provided;

   provided PDU_kind = DC
   with TC[PDU.dst_ID] do
   from (/ any state except close_in_progress /) to same
   begin (/ ignore PDU /) end; end provided; end when;



(* Normal Data Transfer *)


(* Receiving an interface_data_unit from Session Entity *)

when TS[ID].T_DATA_Request (* user_data,
                              is_last_fragment_of_TSDU *)
   with TC[ID] do
   from open to same
   provided TS[ID].TS_ready (user_data.length) = true
            and purge_request = false
            and retransmit_PDU = false
            and wait_for_ACK = false
   begin
   append (S_buffer[ID], user_data,
           is_last_fragment_of_TSDU);
   end; end when;


(* Sending a Data TPDU to Peer Entity *)

any ID : local_ID_type do
   with TC[ID] do
   from open to same
   provided S_credit <> 0
            and wait_for_ACK = false;
            and enough_space (PDU_buffer[ID], max_PDU_size)
                = true
            and ((length_available (S_buffer[ID]) >=
                max_PDU_size - (/ DT header length /)) or
               is_end_of_DU (S_buffer[ID]))
            and MUX.ready_for_sending
   begin
```

```
      SS := SS + 1;
      S_credit := S_credit - 1;
      with TPDU do begin
         kind := DT;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         send_sequence := SS;
         end_of_TSDU := is_end_of_DU (S_buffer[ID]) and
                        (length_available (S_buffer[ID])
                        <= max_PDU_size - (/ DT header
                        length /));
         request_ACK := (S_credit = 0) or end_of_TSDU
                        or (determine_CDT (PDU_buffer[ID])
                        < 2);
         user_data := get_next_fragment (S_buffer[ID],
                      max_PDU_size - (/ DT header size /));
         end;
      store_PDU (PDU_buffer[ID], SS, TPDU);
      if TPDU.request_ACK = true then begin (* enter the
                            state of waiting for Ack *)
         retransmit_count := 0;
         wait_for_ACK := true;
         out TM[ID, NRL].set_timer;
         end;
      out MUX.forward (TPDU);
      end; end any;


(* Retransmitting a Data TPDU to Peer Entity *)

any ID : local_ID_type do
   with TC[ID] do
   from open to same
   provided S_credit <> 0
            and retransmit_PDU = true
            and wait_for_ACK = false
            and MUX.ready_for_sending
   begin
      retransmit_SS := retransmit_SS + 1;
      S_credit := S_credit - 1;
      retrieve_PDU (PDU_buffer[ID], retransmit_SS, TPDU);
      TPDU.request_ACK := (S_credit = 0) or TPDU.end_of_TSDU;
      if TPDU.request_ACK = true
         then begin (* enters the state of waiting for ACK *);
              retransmit_count := 0;
              out TM[ID, NRL].set_timer;
              wait_for_ACK := true;
              end;
         else if SS = retransmit_SS
                 then retransmit_PDU := false; (* returns
                 to the state of normal data transfer *)
      out MUX.forward (TPDU);
      end; end any;
```

```
(* Receiving a data TPDU from Peer Entity *)

when MUX.forward (* PDU *)
with TC [PDU.dst_ID] do
from open to same
   provided PDU.kind = DT
            and purge_request = false
            and enough_space (R_buffer[PDU.dst_ID], MUX.received_
            PDU_size - (/ DT header length /)) = true
   begin
      if RS = PDU.send_sequence
         then begin
              append (R_buffer[PDU.dst_ID], PDU.user_data,
                      PDU.end_of_TSDU);
              RS := RS + 1; end;
      if (RS < PDU.send_sequence) or (PDU.request_ACK = true)
         then with ACK_TPDU do begin
              kind := ACK;
              CDT := determine_CDT (R_buffer[PDU.dst_ID]);
              dst_addr := peer_addr;
              dst_ID := peer_ID;
              expected_seq_number := RS;
              ACK_PDU_full := true;
              end;
         else (/ discards duplicated PDU /);
      end; end provided;
   (* The above transition describes the appending of a data
      unit into the 'R_buffer', and the forming of an ACK
      TPDU if necessary.  The sending of this ACK TPDU, if
      any, is executed by another transition which will be
      described later *)

   provided PDU.kind = DT and purge_request = true
   begin (/ ignores PDU /) end; end provided; end when;


(* Sending an interface_data_unit (IDU) to Session Entity *)

any ID : local_ID_type do
   with TC[ID] do
   from open to same
   provided TS[ID].TS_user_ready (IDU_length) = true and
            ((length_available (R_buffer[ID]) >= IDU_length)
            or (is_end_of_DU (R_buffer[ID]) = true))
   begin
      out TS[ID].T_DATA_Indication (
         get_next_fragment(R_buffer ID]), IDU_length,
         is_end_of_DU (R_buffer ID]) and
         (length_available (R_buffer ID]) = IDU_length));
      end; end any;
(* The above transition is executed when the 'R_buffer'
   contains user data of at least 'IDU_length' octets.
   This parameter may be chosen by an implementation in
```

```
   arbitrary manner, except in the case of the end of a
   TSDU where it must correspond to the lenght up to the
   end of the TSDU *)


(* Data Acknowledgement *)


(* Sending Acknowledgement *)

any ID : local_ID_type do
   with TC[ID] do
   from open to same
   provided ACK_PDU_full = true
            and MUX.ready_for_sending
   begin
      ACK_PDU_full := false;
      out MUX.forward (ACK_TPDU);
      end; end any;


(* Receiving Acknowledgement *)

when MUX.forward (* PDU *)
with TC[PDU.dst_ID] do
from open to same
   provided PDU.kind = ACK and purge_request = false
                           and wait_for_ACK = false
                           and retransmit_PDU = false
                           and wait_for_CAA = false
      begin
      if SS <= PDU.expected_seq_number - 1
         then (/ discards the incorrect PDU /);
         else begin (* enters the state of retransmission *)
              S_credit := PDU.CDT;
              retransmit_PDU := true;
              retransmit_SS := PDU.expected_seq_number - 1
              clear_PDU_buffer (PDU_buffer[PDU.dst_ID],
                                retransmit_SS);
              end;
         end; end provided;
      (* The above transition is executed when ACK is
         received during the normal data transfer *)

   provided PDU.kind = ACK and wait_for_CAA = false
                           and wait_for_ACK = false
                           and retransmit_PDU = true
      begin
         if SS <= PDU.expected_seq_number - 1
            then (/ discards the incorrect PDU /);
            else begin
                 S_credit := PDU.CDT;
                 retransmit_SS := PDU.expected_seq_
                                  number - 1;
```

```
                clear_PDU_buffer (PDU_buffer[PDU.dst_ID],
                                  retransmit_SS);
                end;
            end; end provided;
        (* The above transition describes the receiving of ACK
           during the retransmission of TPDUs *)

    provided PDU.kind = ACK and purge_request = false
                        and wait_for_ACK = true
       begin
          out TM[PDU.dst_ID, NRL].stop_timer;
          S_credit := PDU.CDT;
          if SS = PDU.expected_seq_number - 1
             then begin (* returns to the state of normal
                           data transfer *)
                wait_for_ACK = false;
                retransmit_PDU := false;
                clear_PDU_buffer (PDU_buffer[PDU.dst_ID],
                                  SS);
                end;
             else if SS > PDU.expect_seq_number - 1
                then begin (* enters the state of
                              retransmission *)
                     wait_for_ACK := false;
                     retransmit_PDU := true;
                     retransmit_SS := PDU.expected_
                                      seq_number - 1;
                     clear_PDU_buffer (PDU_buffer[PDU.
                        dst_ID], retransmit_SS);
                     end;
                else begin (* remains in the present
                              state *)
                     (/ discards the incorrect PDU /);
                     out TM[PDU.dst_ID, NRL].set_timer;
                     end;
                end; end provided;
       (* The above transition describes the receiving of ACK
          during the waiting of ACK *)

    provided PDU.kind = ACK and (wait_for_CAA = true or
                                 purge_request = true)
       begin (/ ignores PDU /); end; end provided; end when;
       (* The above transition describes the receiving of ACK
          during the waiting of Credit Allocation Ack or purge
          operation *)


(* Expedited Data Transfer *)


(* Receiving the Expedited Data from Session Entity *)
```

```
when TS[ID].T_EX_DATA_Request (* TS_user_data *)
   with TC[ID] do
   from open to same
   provided EX_D_Sent = false and MUX.ready_for_sending
                              and purge_request = false
   begin
      EX_D_sent := true;
      expedited_SS := expedited_SS + 1;
      with EX_TPDU do begin
         kind := EDT;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         send_sequence := expedited_SS;
         user_data := TS_user_data;
         end;
      out TM[ID, EXP].set_timer;
      out MUX.forward (EX_TPDU);
      end; end when;


(* Receiving the Expedited Data from Peer Entity *)

when MUX.forward (* PDU *)
with TC[PDU.dst_ID] do
from open to same
   provided PDU.kind = EDT and EX_D_received = false
                           and purge_request = false
                           and MUX.ready_for_sending
   begin
      if expedited_RS = PDU.send_sequence - 1
      then begin
           expedited_RS := expedited_RS + 1;
           EX_D_received := true;
           out TS[PDU.dst_ID].T_EX_DATA_Indication (
              PDU.user_data);
           end;
      else if expedited_RS = PDU.send_sequence
           then with TPDU do
              begin
                 kind := EACK;
                 dst_addr := peer_addr;
                 dst_ID := peer_ID;
                 expected_seq_number := expedited_RS;
                 end;
              out MUX.forward (TPDU);
           else (/ ignores PDU /);
           end; end provided;

   provided PDU.kind = EDT and (EX_D_received = true
                            or purge_request = true)
   begin (/ ignores PDU /) end; end provided; end when;


(* Receiving the Expedited Data Response from Session
```

```
Entity *)

when TS[ID].T_EX_DATA_Response
   with TC[ID] do
   from open to same
   provided EX_D_received = true and MUX.ready_for_sending
   begin
      EX_D_received := false;
      with TPDU do begin
         kind := EACK;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         expected_seq_number := expedited_RS;
         end;
      out MUX.forward (TPDU);
      end; end when;


(* Receiving the Expedited Data Acknowledgement from
   Peer Entity *)

when MUX.forward (* PDU *)
with TC [PDU.dst_ID] do
from open to same
   provided PDU.kind = EACK and EX_D_sent = true
   begin
      if expedited_RS = PDU.expected_seq_number
         then begin
            out TM[ID, EXP].stop_timer;
            EX_D_sent := false;
            out TS[PDU.dst_ID].T_EX_DATA_Confirm;
            end;
         else (/ ignores PDU /);
         end; end provided;

   provided PDU_kind = EACK and EX_D_sent = false
   begin (/ ignores PDU /) end; end provided; end when;


(* Credit Allocation Request *)


(* Sending the Allocation Request to Peer Entity *)

any ID : local_ID_type do
   with TC[ID] do
   from open to same
   provided S_credit = 0
            and wait_for_ACK = false
            and MUX.ready_for_sending
            and (retransmit_PDU = true
                 or length_available (S_buffer[ID]) <> 0)
            and (/ after a certain delay /)
   begin
```

```
      wait_for_CAA := true;
      with TPDU do begin
         kind := CAR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         end;
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end any;


(* Receiving the Credit Allocation Request from Peer Entity *)

when MUX.forward (* PDU *)
with TC[PDU.dst_ID] do
from open to same
   provided PDU.kind = CAR
            and purge_request = false
            and MUX.ready_for_sending
   begin
      with TPDU do begin
         kind := CAA; (* Credit Allocation Acknowledgement *)
         CDT := determine_CDT (R_buffer[PDU.dst_ID]);
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         end;
      out MUX.forward (TPDU);
      end; end provided;

   provided PDU.kind = CAR and purge_request = true
   begin (/ ignores PDU /) end; end provided; end when;


(* Receiving the Credit Allocation Acknowledgement from
   Peer Entity *)

when MUX.forward (* PDU *)
with TC[PDU.dst_ID] do
from open to same
   provided PDU.kind = CAA and wait_for_CAA = true
      begin
         out TM[PDU.dst_ID, NRL].stop_timer;
         wait_for_CAA := false;
         S_credit := PDU.CDT;
         end; end provided;

   provided PDU.kind = CAA and wait_for_CAA = false
      begin (/ ignores PDU /); end; end provided; end when;
```

```
(* Purge Operation *)

(* Handling the Purge Request from the Session Entity *)

when TS[ID].T_PURGE_Request
   provided MUX.ready_for_sending
   with TC[ID] do
   from open to same
   begin
      out TM[ID, NRL].stop_timer;
      out TM[ID, EXP].stop_timer;
      RS := 1;
      SS := 0;
      retransmit_SS := 0;
      expedited_SS := 0;
      expedited_RS := 0;
      retransmit_count := 0;
      retransmit_PDU := false;
      wait_for_ACK := false;
      wait_for_CAA := false;
      ACK_TPDU_full := false;
      EX_D_sent := false;
      EX_D_received := false;
      purge_request := true;
      clear_all_buffer (ID);
      with TPDU do begin
         kind := PR;
         CDT := determine_CDT (R_buffer[ID]);
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         end;
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end when;


(* Handling the Purge Request from Peer Entity *)

when MUX.forward (* PDU *)
   provided PDU.kind = PR.and MUX.ready_for_sending
   with TC[PDU.dst_ID] do
   from open to same
   begin
      out TM[ID, NRL].stop_timer;
      out TM[ID, EXP].stop_timer;
      S_credit := PDU.CDT;
      RS := 1;
      SS := 0;
      retransmit_SS := 0;
      expedited_SS := 0;
      expedited_RS := 0;
      retransmit_count := 0;
      retransmit_PDU := false;
```

```
      wait_for_ACK := false;
      wait_for_CAA := false;
      ACK_TPDU_full := false;
      EX_D_sent := false;
      EX_D_received := false;
      purge_request := false;
      clear_all_buffer (ID);
      with TPDU do begin
         kind := PC;
         CDT := determin_CDT (R_buffer[ID]);
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         end;
      out TS[PDU.dst_ID].T_PURGE_Indication;
      out MUX.forward (TPDU);
      end; end when;


(* Handling the Purge Confirm from Peer Entity *)

when MUX.forward (* PDU *)
with TC [PDU.dst_ID] do
from open to same
   provided PDU.kind = PC and purge_request = true
   begin
      out TM[ID, NRL].stop_timer;
      purge_request := false;
      S_credit := PDU.CDT;
      end; end provided;

   provided PDU.kind = PC and purge_request = false
   begin (/ ignores PDU /) end; end provided; end when;



(* Time Out *)


(* Handling the Timeout before the max. number of
   Retransmission is Reached *)

when TM[ID, NRL].time_out;
with TC[ID] do
   provided MUX.ready_for_sending
            and retransmit_count <= max_count

      (* Handling a Timeout After Sending Connection Request *)
      from open_in_progress_calling to same
         begin
            retransmit_count := retransmit_count + 1;
            retrieve_PDU (PDU_buffer[ID], 0, TPDU);
            out TM[ID, NRL].set_timer;
            out MUX.forward (TPDU);
```

```
      end; end from;

   (* Handling a Timeout After Sending a Disconnect Request *)
   from close_in_progress to same
      begin
         retransmit_count := retransmit_count + 1;
         retrieve_PDU (PDU_buffer[ID], SS, TPDU);
         out TM[ID, NRL].set_timer;
         out MUX.forward (TPDU);
         end; end from; end provided;

(* Handling a Timeout After Retransmitting Data and
   requesting ACK *)
provided MUX.ready_for_sending
         and wait_for_ACK := true
         and retransmit_PDU = true
         and retransmit_count <= max_count
   from open to same
   begin
      retransmit_count := retransmit_count + 1;
      retrieve_PDU (PDU_buffer[ID], retransmit_SS,
                    TPDU);
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end provided;

(* Handling a Timeout After Sending Data and Requesting
   ACK *)
provided MUX.ready_for_sending
         and wait_for_ACK = true
         and retransmit_PDU = false
         and retransmit_count <= max_count
   from open to same
   begin
      retransmit_count := retransmit_count + 1;
      retrieve_PDU (PDU_buffer[ID], SS, TPDU);
      out TM[ID, NRL].set_timer;
      out MUX.forward (TPDU);
      end; end provided;

(* Handling a Timeout After Sending Credit Allocation
   Request *)
provided MUX.ready_for_sending
         and wait_for_CAA = true
         and retransmit_count <= max_count
   from open to same
   begin
      retransmit_count := retransmit_count + 1;
      with TPDU do begin
         kind := CAR;
         dst_addr := peer_addr;
         dst_ID := peer_ID;
         end;
      out TM[ID, NRL].set_timer;
```

```
        out MUX.forward (TPDU);
        end; end provided;

   (* Handling a Timeout After Sending Purge Request *);
   provided MUX.ready_for_sending
            and purge_request = true
            and retransmit_count <= max_count
      from open to same
      begin
         retransmit_count := retransmit_count + 1;
         with TPDU do begin
            kind := PR;
            dst_addr := peer_addr;
            dst_ID := peer_ID;
            end;
         out TM[ID, NRL].set_timer;
         out MUX.forward (TPDU);
         end; end provided; end when;


(* Handling a Timeout After Sending Expedited Data *)

when TM[ID, EXP].time_out
with TC[ID] do
from open to same
   provided MUX.ready_for_sending and EX_D_sent = true
      begin
         out TM[ID, EXP].set_timer;
         out MUX.forward (EX_TPDU); (* retransmit expedited
                                       data *)
         end; end when;


(* Handling a Timeout After Max. Number of Retransmission
   has Reached *)

when TM[ID, NRL].time_out
with TC[ID] do
   provided retransmit_count > max_count
      from open_in_progress_calling to close
         begin
            out TS[ID].T_DISCONNECT_Indication (
                TS_USER_UNKNOWN,...(* dummy *));
            peer_addr := 0;
            peer_ID := 0;
            clear_all_buffer (ID);
            end; end from;

      from close_to_progress to close
        begin
            (/ informs user that the remote entity
               has failed /);
            peer_addr := 0;
            peer_ID := 0;
```

```
            clear_all_buffer (ID);
            end; end from; end provided;

      provided retransmit_count > max_count
            and MUX.ready_for_sending
         from open to close_in_progress
            begin
            with TPDU do begin
               kind := DR;
               dst_addr := peer_addr;
               dst_ID := peer_ID;
               src_addr := local_T_addr;
               src_ID := ID;
               reason := 129; (* transport entity initiates
                                 disconnect *)
               user_data := ...; (* dummy *)
               end;
            store_PDU (PDU_buffer[ID], SS, TPDU);
            out TS[ID].T_DISCONNECT_Indication (TS_FAIL,...
                                                (* dummy *));
            out TM[ID, NRL].set_timer;
            out MUX.forward (TPDU);
            end; end provided; end when;


end Protocol;
```

CONCLUSIONS

In this thesis, a proposed design of the Transport Layer for local area network networks is discussed. It is believed that the Transport Layer built on the Data Link Layer (rather than on the Network Layer as in the OSI architecture) is sufficient for providing a reliable data delivery service between processes in a local network environment. The service provided by the Data Link Layer is assumed to be an unacknowledged connectionless service. As such, the Transport Layer is responsible for the provision of all functions which bridge the gap between the services provided by the Data Link Layer and the services needed by the Session Layer.

The Transport Layer provides a connection-oriented service to the Session Layer. This service requires that a transport connection is established between the two communicating session entities before data transfer can begin. A transport connection features a three-party agreement : this allows the two communicating session entities and the transport service provider to reserve a set of resources for their exclusive use throughout the lifetime of their associations. By doing so, orderly communications and status information are maintained. The

connection-oriented service provided to the Session Layer is a reliable, in-sequence data delivery service in which messages could be transmitted at two different priority levels. They are normal and expedited data transfer. The Transport Layer also provides the means for the session entities to uniquely identify their counter-parts.

The transport protocol specifies the communication behaviour of transport entities in the establishment, maintainance and termination of transport connections. The functions performed by the transport protocol are transmission error recovery, fragmentation, flow control, sequencing, duplicate detection, purge and expedited data transfer. During the design, the protocol was analyzed by using a technique of perturbation based on a reliability analysis. Design errors such as state deadlocks, message ping-ponging, unspecified receptions and unexecutable interactions were identified and corrected. Finally, the proposed transport protocol was specified by the Formal Description Technique, and a list of commands and their formats was given.

The proposed protocol can be integrated into commercially available LANs, such as Ethernet, simple because of the service that provided to the Transport Layer is an unacknowledged connectionless type. Some of the limitations of this design are :

1. not efficient in supporting the requirements of user applications that are naturally connectionless-oriented,
2. not efficient for users that could accept reliable service, and
3. must rely totally on the Data Link Layer for error detection.

Although the Transport Layer provides only connection-oriented service, it allows transaction service to be easily integrated into the connection-oriented service. On the other hand, the Transport Layer was structured into separate different modules which can be easily modified. Since FDT was used to describe the transport protocol, it provides a specification which is closed to an implementation. Some of the requirements for implementing this design are :

1. to define the exact format and mechanisms for conveying the transport service primitives,
2. to determine the local conditions which transport entity may terminate transport connections, and
3. to define the timer module.

In this design, only the problem of protocol reliability was addressed. It will be interesting to study the efficiency performance of the proposed protocol. For other further studies, the following areas can be considered :

1. Providing a connectionless service to the Session Layer.
2. Providing different classes of transport protocol for different applications.
3. Allowing the Transport Layer to employ connection-oriented service.

The contributions of this thesis are the design of the transport protocol for local area networks and the representation of the protocol design in FDT.

APPENDIX A - OVERVIEW OF THE SEVEN LAYERS OF THE OSI ARCHITECTURE

This appendix gives an overview of the seven layers defined in the OSI architecture. These seven layers are Application, Presentation, Session, Transport, Network, Data Link and Physical.

A.1 The Application Layer : Being the highest layer defined by the OSI architecture, this layer supports the application-processes which exchange meaningful information with each other. Protocols of this layer directly serve the end user by providing the distributed information service appropriate to an application.

A.2 The Presentation Layer : This Layer provides the set of services which may be selected by the Application Layer to enable it to interpret the meaning of the exchange. These services are for the management of the entry, exchange, display and control of structured data. In other words, the Presentation Layer is responsible for resolving differences in data format.

A.3 The Session Layer : This layer provides the means necessary for co-operating Presentation entities to organize and synchronize their dialogue and to manage their data exchange in a proper order.

A.4 The Transport Layer : The Transport Layer provides, in association with the layers below it, an universal transport service which is independent of the physical medium in use. This layer provides transparent transfer of data between session entities and relieves them from any concern with the detailed way in which reliable and efficient transfer of data is achieved. All protocols defined in this layer have end-to-end significance, i.e. transport protocols operate only between end systems.

A.5 The Network Layer : This layer provides functional and procedural means for exchanging network-service-data-units between two transport entities. The Network Layer relieves the transport entities of routing and switching considerations associated with the transfer of data units.

A.6 The Data Link Layer : The Data Link Layer provides functional and procedural means for establishing, maintaining and releasing data links between network entities of the adjacent systems.

A.7 The Physical Layer : This layer provides mechanical, electrical, functional and procedural characteristics for establishing, maintaining and releasing physical connection between data link entities. A physical connection is a communication path in the physical media between two physical entities.

APPENDIX B - SERVICES AND FUNCTIONS DEFINED IN THE OSI ARCHITECTURE

In this appendix, a list of the services and functions for each of the layers of the OSI architecture is presented. This list is compiled directly from the Draft International Standard ISO/DIS 7498 : Information Systems - Open System Interconnection - Basic Reference Model (April 1982).

B.1 Physical Layer

Services provided by the layer

1. Physical-connections
2. Physical-service-data-unit
3. Physical-connection-endpoints
4. Data circuit identification
5. Fault condition notification

Functions in the layer

1. Physical-connection activation and deactivation
2. Physical-service-data-unit transmission
3. Physical layer management

B.2 Data Link Layer

Services provided by the layer

1. Data-link connections

2. Data-link-service-data-units
3. Data-link-connection-endpoint-indentifiers
4. Sequencing
5. Error notification
6. Flow control
7. Quality of service parameters

Functions in the layer

1. Data-link-connection establishment and release
2. Data-link-service-data-unit-mapping
3. Delimiting and synchronization
4. Error detection and recovery
5. Flow control
6. Data link layer management

B.3 Network Layer

Services provided by the layer

1. Network address
2. Network-connections
3. Network-connection-endpoints-identifiers
4. Normal and expedited transfer of network-service-data-units
5. Error notification and reset
6. Sequencing
7. Flow control
8. Quality of service parameters

Functions in the layer

1. Routing and relaying
2. Network-connection multiplexing
3. Segmenting and blocking
4. Error detection and recovery
5. Sequenceing
6. Flow control
7. Expedited data transfer
8. Service selection
9. Network layer management

B.4 Transport Layer

Services provided by the layer

1. Identification
   - Transport-addresses
   - Transport-connections
   - Transport-connection-endpoint-identifiers
2. Establishment services
   - Transport-connection establishment
   - Class of service selection
3. Data transfer services
   - Transport-service-data-unit
   - Expedited transport-service-data-unit
4. Transport-connection release

Functions in the layer

1. Addressing
2. Connection multiplexing and splitting
3. Connection establishment
4. Data transfer
   - Sequencing
   - Segmenting and blocking
   - Concatenation
   - Multiplexing and splitting
   - Flow control
   - Error detection and recovery
   - Expedited data transfer
   - Transport-connection identification
5. Connection release
6. Transport layer management

B.5 Session Layer

Services provided by the layer

1. Session-connection establishment and release
2. Normal and expedited data exchange
3. Interaction Management
   - Two-way simultaneous
   - Two-way alternate
   - One-way
4. Session-connection synchronization

Functions in the layer

1. Mapping of session-connection onto transport-connection
2. Session-connection flow control
3. Session-connection recovery
4. Session-connection release
5. Expedited data transfer
6. Session layer management

B.6 Presentation Layer

Services provided by the layer

1. Data formatting
2. Data transformation
3. Syntax selection
4. Presentation connections

Functions in the layer

1. General
   - Session establishment request
   - Presentation image negotiation and renegotiation
   - Data transformation and formatting
   - Session termination request
2. Addressing and multiplexing
3. Presentation layer management

APPENDIX C - PROTOCOL VALIDATION

This appendix describes the validation of the transport protocol by using a technique of perturbation. Figure 26 shows a two-process interaction which is derived from the state diagram of a transport connection (see Figure 24). The perturbation technique is used to validate the protocol for normal operation before adding recovery actions.

The perturbation technique creates the reachability tree for a n-process interaction by simply defining the system states as n*n arrays. Since Figure 26 represents a two-process interaction, the system state SS in Figure 27 are 2*2 arrays. A system state is defined where the elements on the main diagonal represent the individual process state (element 1,1 is state of P_1 and so on) and each off-diagonal element i, k represents the message content of the communication medium from process P_i to process P_k.

One begin by defining SS0 which is the initial system state. It consists of both processes in CLOSE (state 0) and both channels empty (represented by E). SS0 is then 'perturbed' into all possible successor state reachable by executing a single transition in one of the individual processes P_1 ,P_2 (in Figure 26). The procedure continues

until all reachable system state have been determined, as shown in Figure 27

Deadlocks are identified in a reachability tree by system states with all channels empty and no departing transitions. Unspecified receptions are identified by system states with no departing transition to absorb the next output from one of the channels. Nonexecutable interactions are identified as state transitions present in the design that are absent in the reachability tree.

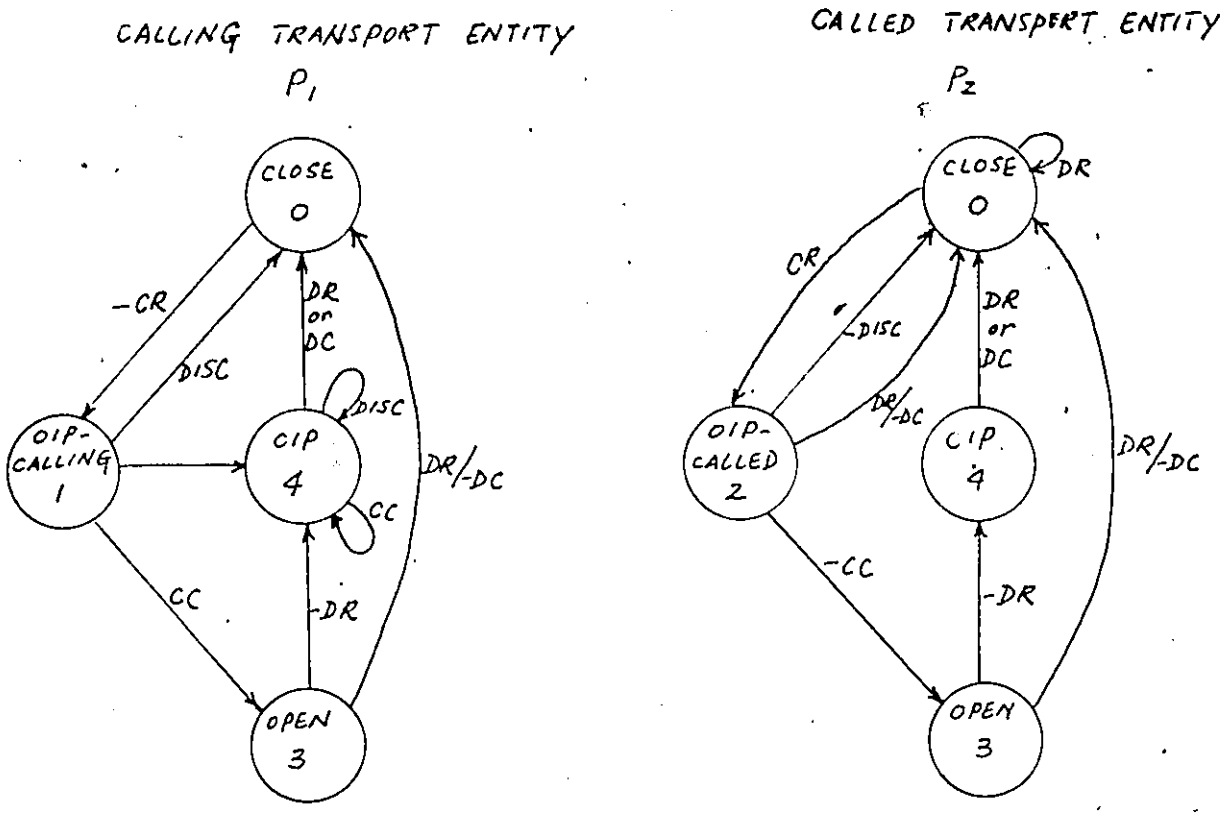


Figure 26: Two-Process Interaction

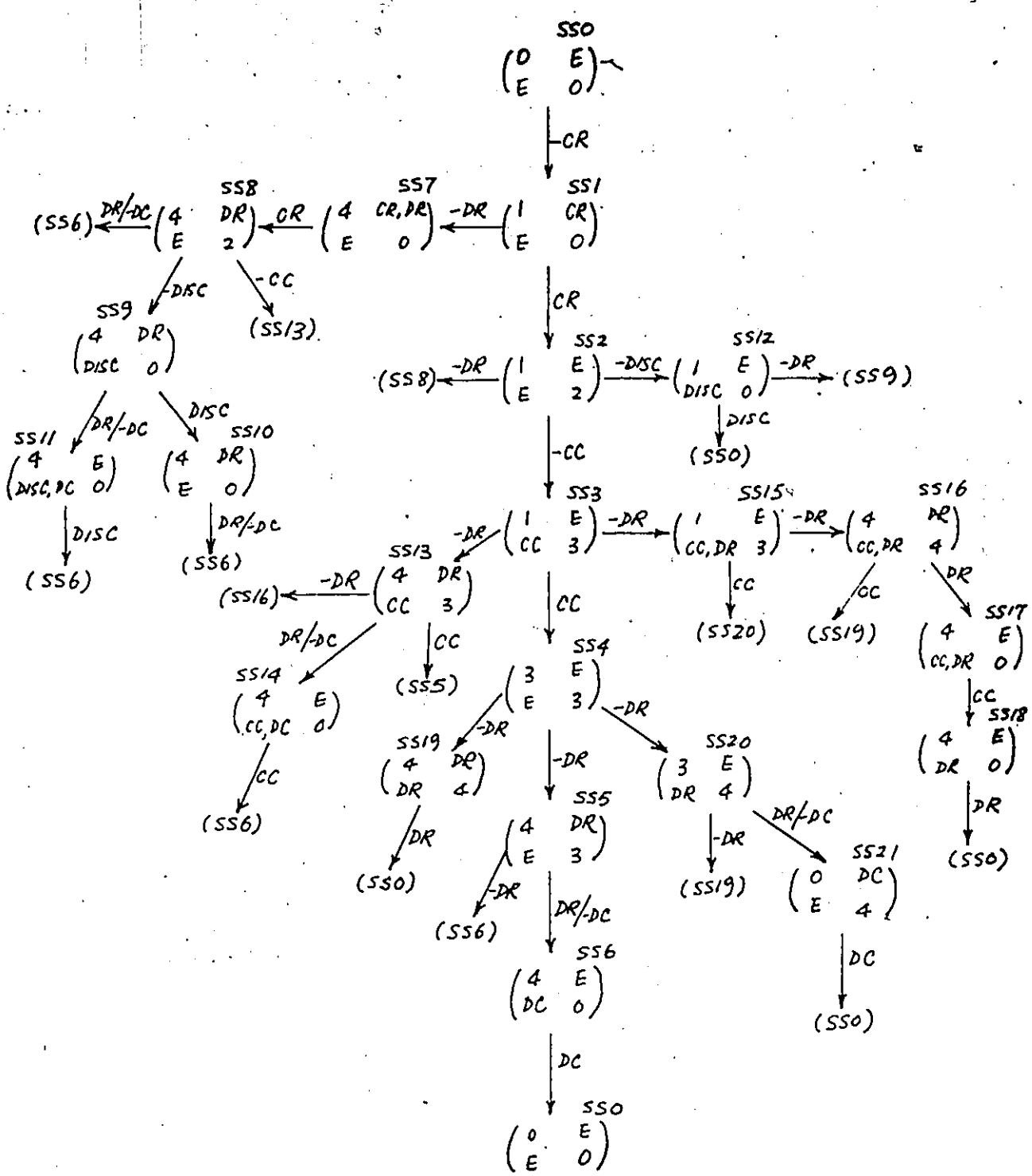


Figure 27: Reachability Tree for Two-Process Interaction

REFERENCES

1. K.J.Thurber, H.A.Freeman, Tutorial : Local Computer Networks, 2nd Edition, IEEE Computer Society, 1981.
2. K.C.E.Gee, Local Area Networks, NCC Publications, 1982.
3. A.J.Mayne, Linked Local Area Networks, October press, 1982.
4. D.C.Walden, 'Host-to-Host Protocols', Tutorial : A Practical View of Computer Communications Protocols (edited by J.M.Mcquillan and V.G.Cerf), IEEE Computer Society, 1978.
5. V.G.Cerf, R.E.Khan, 'A Protocol for Packet Network Interconnection', IEEE Trans. Comm., vol-22, pp.637-648, May 1974.
6. V.G.Cerf et al, 'Proposal for an International End to End Protocol', Tutorial : A Practical View of Computer Communication Protocol (edited by J.M.Mcquillan and V.G.Cerf), IEEE Computer Society, 1978.
7. A.Danthine, F.Magnee, 'Transport Layer - Long-haul Vs Local Network', Local Networks for Computer Communications (edited by A.West and P.Janson), North-Holland Publishing Company, 1981.

8. ECMA, 'Standard ECMA-72 : Transport Protocol', ECMA, 1981.

9. E.A.Yakubaitis, Network Architecture for Distributed Computing, Allerton Press, 1983.

10. A.S.Tanenbaum, Computer Networks, Prentice-Hall, 1981.

11. ISO, 'Information Processing System - Open System Interconnection - Basic Reference Model', ISO/DIS 7489, 1982.

12. IEEE, 'IEEE Standard 802.3: CSMA/CD Access Method and Physical Layer Specification', IEEE Project 802 LAN Standards, IEEE Computer Society, July 1983.

13. C.K.Miller, D.M.Thompson, 'Making a Case for Token Passing in Local Networks', The Local Network Handbook (edited by G.R.Davis), McGraw Hill, 1982.

14. J.Buruss, 'Draft Report - Features of the Transport and Session Protocols', ISCT, National Bureau of Standards, March 1980.

15. A.L.Chapin, 'Connection and Connectionless Data Transmission', Proc. IEEE, vol-71, pp.1365-1371, Dec. 1983.

16. J.M.Mcquillan, V.G.Cerf, Tutorial : A Practical View of Computer Communication Protocols, IEEE Computer Society, 1978.

17. H.C.Folts, 'X.25 Transaction-Oriented Features-Datagram and Fast Select', IEEE Trans. Comm. vol-28, pp.496-500, April 1980.

18. C.A.Sunshine, 'Transport Protocol for Computer Network', Protocols and Techniques for Data Communication Networks', (Edited by F.F.Kuo), Prentice-Hall, pp.35-77, 1981.

19. DIX, 'The Ethernet, A Local Area Network : Data Link Layer and Physical Layer Specification', Version 1.0, Digital Equipment, Intel, Xerox, Sept. 1980.

20. D.D.Clark, K.T.Pogran, D.P.Reed, 'An Introduction to Local Area Networks', Proc. IEEE, vol-66, pp.1497-1517, Nov. 1978.

21. W.S.Lai, 'Protocol Traps in Computer Networks - A Catalog,', IEEE Trans. Comm., vol-30, pp.1434-1449, June 1982.

22. P.Zafiropulo et al, 'Towards Analyzing and Synthesizing Protocols', IEEE Trans. Comm., vol-28, pp.651-660, April 1980.

23. J.Hajek, 'Automatically Verified Data Transfer Protocols', Proc. 4th Int. Conf. Comput. Comm., Kyoto, Japan, pp.749-756, Sept.26-29, 1978.

24. G.V.Bochmann, C.Sunshine, 'Formal Methods in Communication Protocol Design', IEEE Trans. Comm., vol-28, pp.624-632, April 1980.

25. P.F.Linington, 'Fundamentals of the Layer Service Definitions and Protocol Specification', Proc. IEEE, vol-71, pp.1341-1345, Dec. 1983.

26. P.E.Green, 'An Introduction to Network Architectures and Protocols', IEEE Trans. Comm., Vol-28, pp.413-424, Apr. 1980.

27. ISO, 'Tutorial on Formal Description Techniques', ISO TC97/SC16/WG1, ad hoc group on FDT, Jan. 1981.

28. ISO, 'Concepts for Describing the OSI architecture', ISO TC97/SC16/WG1, Subgroup A of ad hoc group on FDT, Working Document N1346, Nov. 1982.

29. ISO, 'An FDT Based on an Extended State Transition Model', ISO TC97/SC16/WG1, Subgroup B of ad hoc group on FDT, Working document N1347, May 1983.

30. C.A.Vissers, R.L.Tenney, G.V.Bochmann, 'Formal Description Techniques', Proc.IEEE, vol-71, pp.1356-1364, Dec. 1983.

31. G.V.Bochmann, 'Examples of Transport Service Specifications', Working Document No.145, Dept. of Computer Science and Operational Research, Univ. of Montreal, Nov. 1983.

32. G.V.Bochmann, 'Example of a Transport Protocol Specification', Working Document No.146, Dept. of Computer Science and Operational Research, Univ. of Montreal, Nov. 1983.

33. G.V.Bochmann, M.Raynal, 'Structured Specification of Communicating Systems', IEEE Trans. Computers, vol-32, pp.120-133, Feb. 1983.

34. IEEE, 'IEEE Draft Standard 802.2 : Logical Link Control', IEEE Project 802 LAN Standards, IEEE Computer Society, Sept. 1983.

35. IEEE, 'IEEE Draft Standard 802.4 : Token Passing Bus Access Method and Physical Layer Specifications', IEEE Project 802 LAN Standards, IEEE Computer Society, July 1983.

36. A.A.S. Danthine, 'Network Interconnection', Local Computer Network, North-Holland Publishing Company, IFIP, 1982.

37. V.B. Hunt, 'Olivetti Local Network System Protocol Architecture', Local Network for Computer Communications, North-Holland Publishing Company, IFIP, 1981.

38. R.Ryan et al, 'Intel Local Network Architecture', IEEE Micro, Nov. 1981.

39. W. Myers, 'Toward a Local Network Standard', IEEE Micro, Aug. 1982.

40. T. Kalin, G.L. Moli, 'Standardization Issues and Protocol Architecture for LANs in View of OSI Reference Model', Proceedings of the Sixth ICCC, pp.111-114, London, Sept.7-10, 1982.