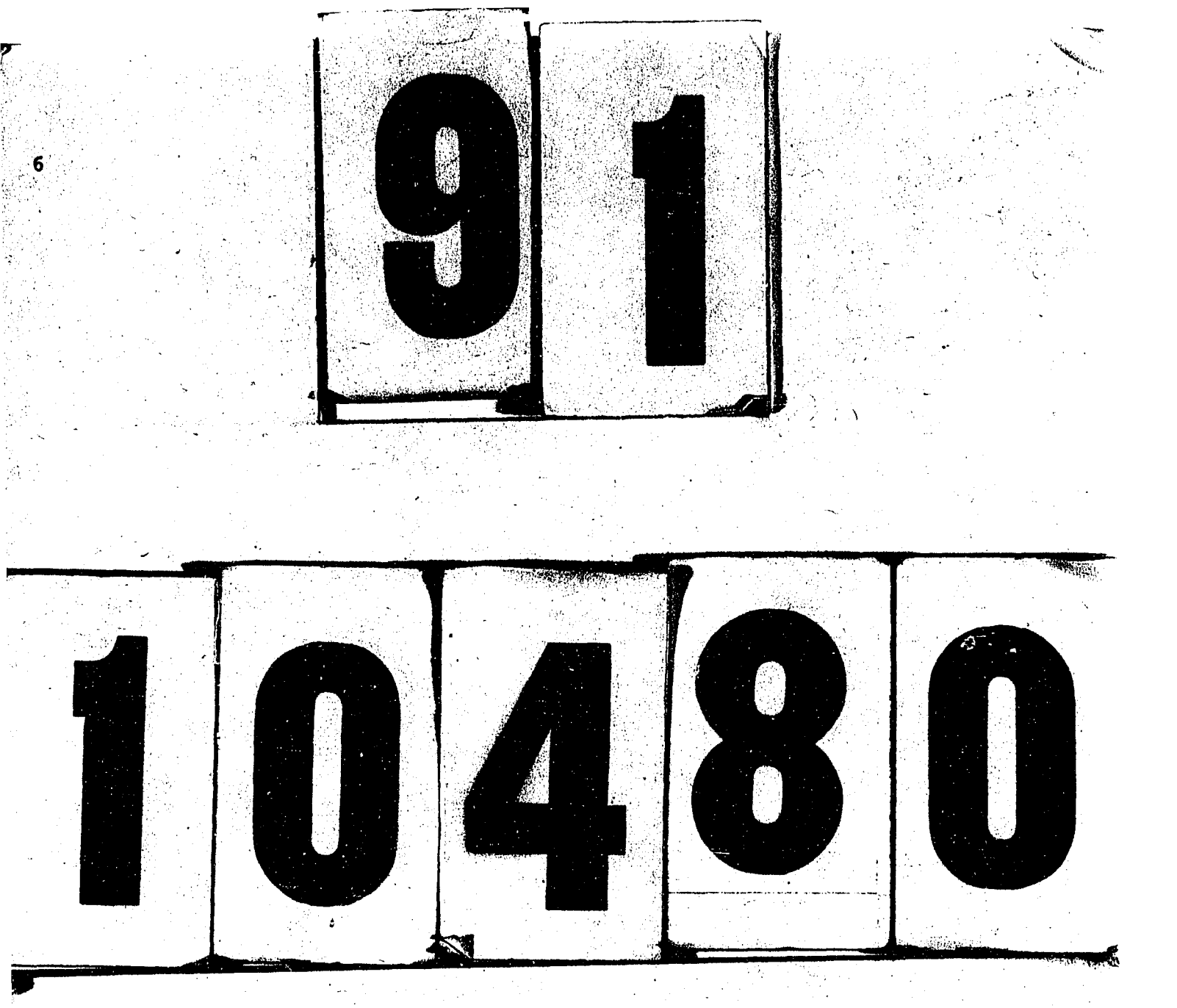
1990

Transport interoperability using a virtual transport layer

Ratinder Paul Singh Ahuja
Iowa State University



Order Number 9110480

Transport interoperability using a virtual transport layer

Ahuja, Ratinder Paul Singh, Ph.D.

Iowa State University, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Transport interoperability using a virtual transport layer

by

Ratinder Paul Singh Ahuja

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved :

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Members of the Committee:

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1990

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

The need for internetworking comes from the fact that there is no one network type that satisfies every computer communications requirement. The resources of interest to a user may be distributed on various different networks. Due to irreconcilable differences in the network technologies, it is impractical to consider merging them into a single network. What is needed is the ability to interconnect various networks so that any two stations on any of the constituent networks can communicate. An interconnected set of networks is referred to as an *internet.*

Standards and techniques exist to provide internetworking at various levels in the seven layer OSI [1] model. These methods require either a common transport layer in the end entities, with the internet providing services of the network layer, or protocol translators to map higher level services from dissimilar networks. The first approach requires the transport layers on the host to provide the end-to-end reliability, flow control and error control. The latter approach requires building protocol translators or services bridges between each pair of dissimilar network architectures that need to communicate. The focus of this research is to provide a method for internetworking in which the end systems do not need the same transport layer. The problem statement and the need to solve the problem is described in Sections 1.1

and 1.2, respectively. Section 1.3 describes the current approaches to transport layer interoperability and their limitations. The goals of the research are presented in Section 1.4.

1.1 Problem Statement

The problem to be solved is to provide internetworking at the transport layer for applications that use dissimilar transport protocols. The solution should preserve the end-to-end meaning of the transport service, and should simplify the task of communicating with any number of different transport protocols by translating TPDUs to those of an intermediate format. This intermediate meta transport is referred to as the *Virtual Transport Layer.*

The problem can then be further sub-divided into two phases. The first phase involves the design and specification of a virtual transport layer which is implemented using gateways. The second phase involves designing protocol convergence functions in the gateways to translate services and protocol data units from standard transport layers to those of the virtual transport layer. At this stage, OSI and DoD transport layers will be considered.

1.2 Why it Needs to be Solved

The transport layer in a communication architecture provides for an end-to-end data transfer in a reliable manner. A number of applications like NetBIOS [2] [3], RPC mechanisms [4], X-Windows [5] and distributed data management systems require transport services from the communication architecture. With programmatic interfaces like TLI [6] and XTI [7], and

service mapping protocols such as RFC1006 [8], it is possible to write these applications to work on different transport protocols with a minimum porting effort. It then becomes essential to provide a means for such applications to interoperate, even if they employ the services of different transport protocols.

1.3 Transport Interoperability Approaches

A survey of the research literature [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] shows two popular methods to provide interoperability at the transport layer and higher layers, namely *Protocol Conversion* and *Service Mapping*. These two methods as applied to DoD TCP and ISO TP4 interoperability, are discussed in this section along with their advantages and disadvantages.

1.3.1 Protocol Conversion Based Approach

Protocol translators interconnect heterogeneous networks at functionally similar, but incompatible higher level protocols. They map one high-level abstraction into another. The mapping can be implemented in a variety of ways such as software modules, dedicated hardware, temporary storage, etc. The main reasons for having protocol converters to interconnect heterogeneous networks are summarized in [12] and listed below:

- They increase connectivity.

- They minimize the number of protocols to be supported and used.
- They are strategic components for the migration to OSI.

Protocol conversion concerns itself with protocol interactions that occur between peers from different protocol suites that reside at the same logical layers in their respective protocol hierarchies. Figure 1.1 shows two protocol entities E1 and E2 performing similar tasks but in different protocol suites.

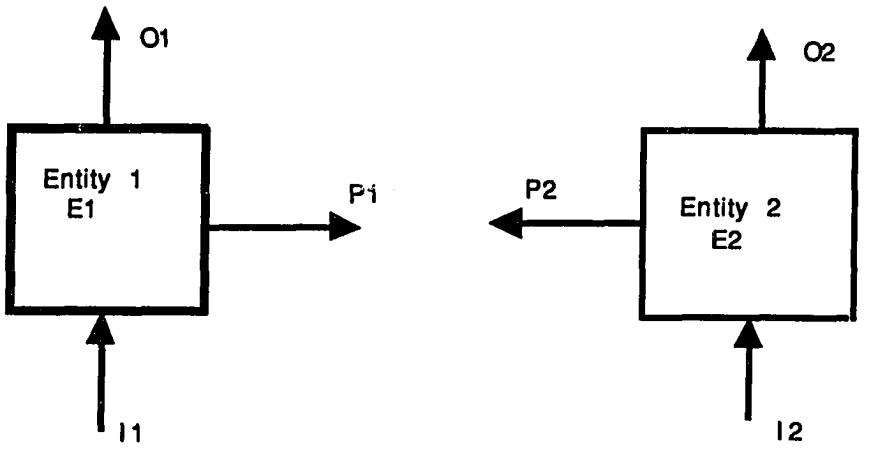


Figure 1.1 - Model of Inputs, Outputs and the Protocols of an Entity

Each entity En has its own set of definitions for the following:

1. The service **In** required from the entity below.
2. The service **On** offered to the entity above.
3. The protocol **Pn** used with its peers.

The task is to connect the two entities to achieve a cross-over between the protocol stacks using a mapping function, as depicted in Figure 1.2, which converts protocol interactions from one stack to another. Any two protocols will have some functional differences, so that an attempt to map between them will entail some loss of power and functionality in the end protocols. On the other hand, since protocol conversion is done in real-time with no store/forward characteristics, the *end-to-end significance* of service is maintained. A solution based on conversion between a common subset of the DoD TCP and ISO TP4 is presented in [15] and is discussed below.

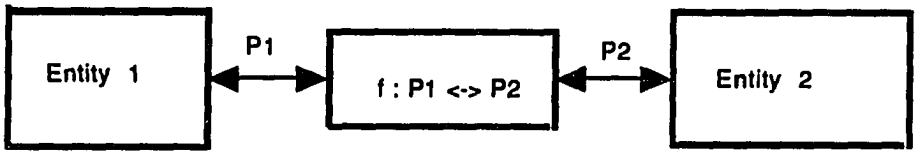


Figure 1.2 - Model of a Protocol Converter

1.3.1.1 Protocol Conversion Between TCP and ISO TP4 The study in [16] was motivated by NATOs intention to introduce ISO protocols in all new systems, while maintaining effective interoperation with existing TCP based systems. Thus, if the same applications can be ported onto the TCP stack and the TP4 stack, then the availability of a conversion facility between TCP and ISO TP4 would allow such interoperation.

The similarity of the two transport protocols makes it possible to achieve interoperability, with some restrictions to the set of services supported. An assessment of the three phases of transport operation is made as follows:

(a) **Connection Establishment Phase:** The ISO service is functionally more powerful than the TCP service, since user data are available in the connection request.

(b) **Data Transfer Phase:** The ISO data transfer service is considered superior because of the availability of expedited data. The TCP URGENT indication to the ULP does not specify any range of octets that are urgent. It merely indicates that some point in the upcoming data stream has been marked urgent by the sending ULP. The URG flag is turned off when the receive sequence number crosses the Urgent Data Pointer.

(c) **Connection Termination:** TCP offers graceful termination; all outstanding data are transmitted successfully before the connection is terminated. ISO defines orderly release in the Session Layer.

The complexity of the protocol converter is no worse than the complexity of either of the two protocols. The task is non-trivial as is noted by the fact that the specification of the protocol converter in [15] contains a description of 654 state/event combinations.

1.3.1.2 Proposed Complementing of TP4 To attain equivalence in the connection release phase, this approach proposes to redefine the level of interoperation on the TP4 side to include the session orderly release sublayer. Figure 1.3 illustrates the resulting level of interoperation.

Table 1.1 shows the compatible set of services, with some restrictions with respect to parameters that are conveyed.

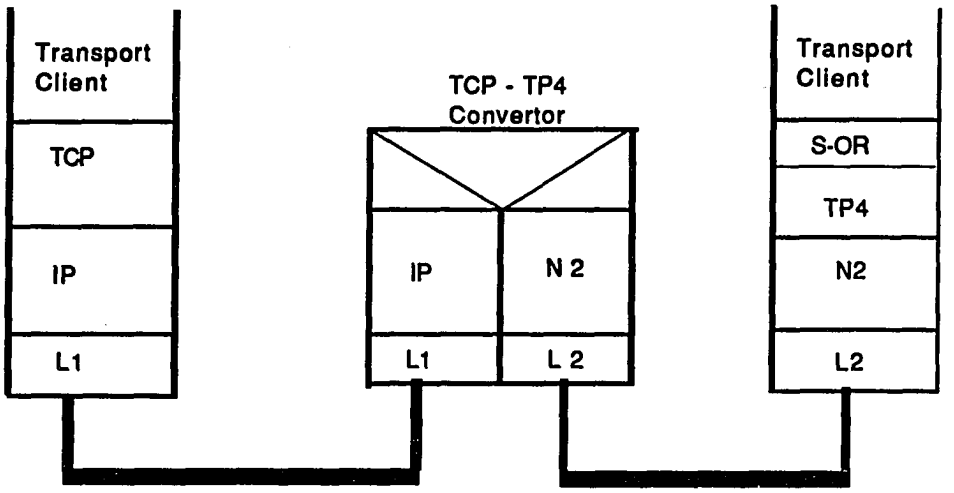


Figure 1.3 - TCP-TP4 Interoperability Model Using a Protocol Converter

Table 1.1 - Compatible Transport Service Primitives Between OSI TP4 and TCP

Phase	Service	Primitive	Parameter
Connection Establishment	TC Establishment	T-CONNECT request	Called Address, Calling Address[a] QOS[b]
		T_CONNECT indication	Called Address, Calling Address, QOS
		T_CONNECT response	QOS, Responding Address
		T_CONNECT confirm	QOS, Responding Address
Data Transfer	Normal Data Transfer	T_DATA request	TS User data[c]
		T_DATA indication	TS User data
Connection Release	Abrupt Release	T-DISCONNECT request [d]	
		T-DISCONNECT indication	Disconnect Reason

[a] Addresses are limited to fit within the TCP constraints on format and structure.

[b] The number of effective choices for the quality of service is limited to what is available for TCP.

[c] The TCP PUSH function may be used for SDU seperation.

[d] No reason parameter can be supported by TCP for the disconnect request primitive.

1.3.1.3 Service Restrictions Due to mismatches in the semantics of the service offered by the two transport mechanisms, the following major restrictions apply:

(a) No user data can be conveyed during TC establishment.

(b) No expedited data transfer.

(c) No URGENT signal in TCP.

(d) ISO session orderly release to be included by TP4.

(e) The use of the TCP PUSH function is restricted as it is now used to delimit TSDUs.

1.3.1.4 Advantages and Disadvantages Protocol conversion is effective if the mismatches between the two target protocols are small. In the case of Transport Layer protocol converters, TCP to TP4 in particular, the major advantages and disadvantages are listed below.

ADVANTAGES:

(a) End-to-end significance of the service is maintained. This being one of the major advantages (besides providing interoperability).

(b) Complexity of the protocol converter is no greater than any of the individual protocols.

(c) Only one more point of failure is introduced.

(d) With the increasing interest shown by the research community in formal methods of synthesis [17] [18], verification and validation of protocol converters [19], the

emergence of automatic tools may become available. With the availability of automatic tools, protocol conversion would be an attractive solution.

DISADVANTAGES:

(a) One such protocol converter is required for each architecture that interoperation is desired between. The complexity of the task, and lack of commercially available automatic tools, make this a non-trivial problem.

(b) The end systems invariably have to sacrifice the use of some services to define a common subset. As mentioned earlier, restrictions were placed on the use of certain services and service parameters to come up with a protocol convertor between TCP and TP4.

(c) A new point of failure is introduced in the connection path.

The next section discusses the approach of service mapping to solve the problem of interoperability and migration to OSI applications.

1.3.2 Service Mapping Based Approach

This approach provides the services from one protocol suite at the same logical layer in a different protocol suite. Figure 1.4 shows how one entity can be used to emulate the service (O2) of another entity.

In particular, providing ISO transport services on top of TCP is considered a favorable approach to introduce ISO applications to a wider set of users. This technique is popularly known as RFC1006. The use of TCP/IP as the lower layers for data transport is based on the following reasoning:

(a) The use of TCP/IP is wide spread, and technology in areas such as routing and network management are mature as compared to ISO.

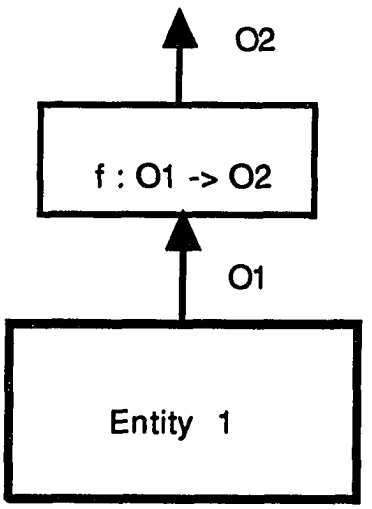


Figure 1.4 - Model of Service Mapping

(b) The migration to OSI applications is economically feasible because the service mapping approach is relatively simple to implement.

(c) Interoperability can be achieved by using TS-Service bridges [20] (described later in this section).

1.3.2.1 OSI Transport Service on TCP RFC1006 specifies how to provide ISO transport service on top of TCP. This is achieved by implementing TP0 on top of TCP/IP. This would allow ISO session, presentation and application entities to operate without knowledge of the fact that they are running on a TCP/IP internet. Figure 1.5 shows the model of operation. All aspects of the ISO transport service are supported except for the quality of service parameter (QOS). Here TCP primarily serves the role of CONS [21], with one fundamental difference: TCP manages a continuous stream of octets with no explicit boundaries. The protocol is described in RFC983 [22] (which precedes RFC1006).

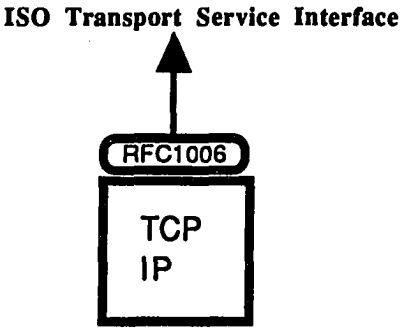


Figure 1.5 - ISO TP Service on TCP.

1.3.2.2 Transport Service Bridges Using the RFC1006 method to provide OSI TS service on top of TCP/IP, it is possible to interoperate with the applications running on ISO transport by using a transport service bridge (which is different from a protocol converter). The service bridge operates on top of the ISO transport stack and TCP/RFC1006 stack, as

shown in Figure 1.6. Since the transport services are (almost) the same regardless of the transport class, it is trivial to operate on top of a homogeneous service interface. The service bridge simply "copies" service primitives from one TS-stack to another. For example, upon receiving a connection indication from one TS-stack, the TS-bridge issues a connection request to the other TS-stack.

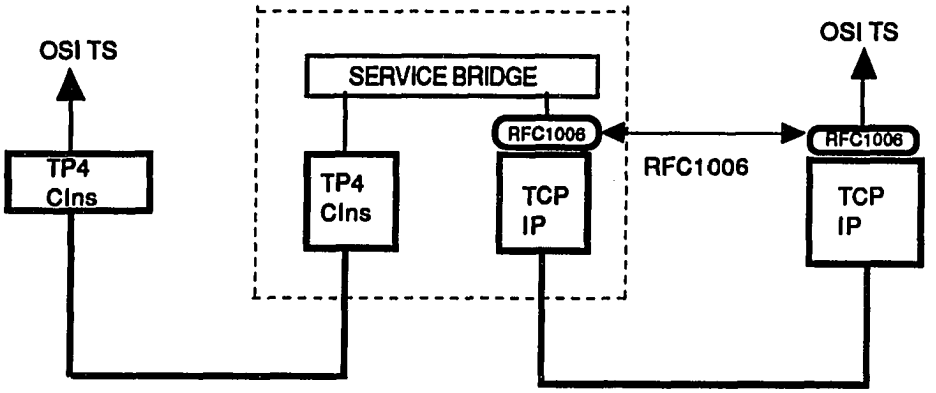


Figure 1.6 - TCP-ISO TP Interoperability using TS-Bridges and RFC1006 Protocol

1.3.2.3 Advantages and Disadvantages The advantages of service bridges are:

(a) The service bridge is a simple component as it needs to know nothing of the protocols themselves. It deals with the relatively simple service primitives.

(b) Interoperability can be achieved on a wide basis if vendors of various architectures provide OSI transport service on top of their transport service and then applications are written using this service interface. One such study for SNA is presented in [23].

The disadvantages of service bridges are:

(a) The service bridge does not provide a true end-to-end meaning to the service. For example, the data acknowledgements, as seen by the end systems would be those originating from the transport in the service bridge rather than from the peer end system. There are in fact two connections, one from the source to the service bridge, and one from the service bridge to the destination. As a result, critical information such as credit allocations and flow control information does not have an end-to-end significance. A similar problem exists with the addresses. The destination sees the transport address of the bridge, and not the transport address of the originator.

(b) There is some performance degradation as checksums are re-calculated at the service bridge. There is also an additional overhead of maintaining transport connections at the service bridge.

(c) The service bridge must implement both the stacks up to the transport layer.

1.3.4 Limitations of Current Approaches

As described above, both protocol convertors and service bridges impose some restrictions on the end users. The major limitations of both the approaches can be summarized as follows:

Protocol Converters: A unique protocol converter must be designed for each architecture that interoperability is required. The task is non-trivial as it requires detailed knowledge of both transport protocols.

Service Bridges: Service bridges lose the end-to-end significance of the transport service due to the introduction of another fully operational transport protocol in the path. Thus, acknowledgements and flow control information are not effective over the complete path of the connection. Service bridges can also become a performance bottleneck.

1.4 Goal of the Research

Keeping in view the above mentioned interconnection problems, the goal of this research is to provide an interconnection method at the transport level such that interoperability can be offered to a transport architecture by designing a single gateway to the Virtual Transport Layer (VTL). The conversion rules for the translators need to be specified so that the design of the translators depends only on the nature of the local transport architecture, making no assumption about the peer (same or different) transport is. The VTL architecture

should also preserve the end-to-end significance of the transport service. An attempt will thus be made to overcome the limitations of both protocol converters and service bridges.

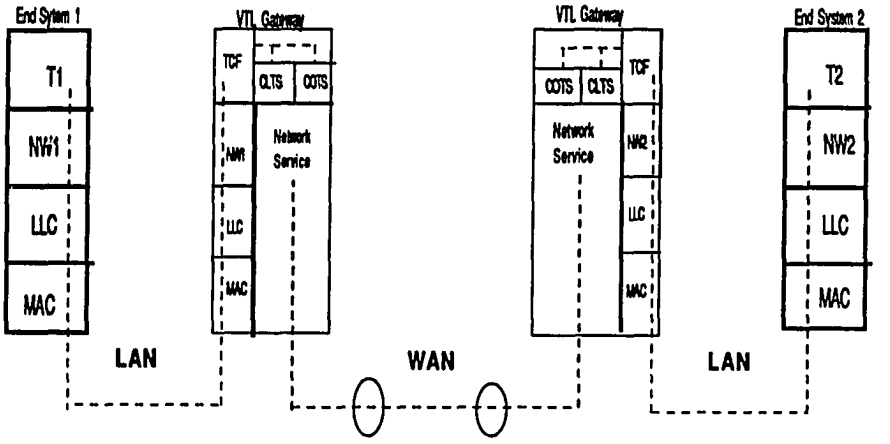


Figure 1.7 - Internet using the Virtual Transport Layer

1.4.1 The VTL Design Approach

The approach as described in this dissertation is based on translating the TPDUs into a common intermediate format, and transferring it between gateways using a CLTS. The intermediate format is identified as that belonging to a virtual transport layer, and the messages are accordingly called virtual transport data units (VTPDUs). Components in the gateways termed as transport convergence functions (TCFs) translate to and from VTPDUs to the local

TPDUs. An end system sees its peer due to a compounding of the VTL and the actual end transport entity.

The design effort formalizes the role of the VTL during the various phases of a transport connection [24]: connection establishment, data transfer and connection termination. TCFs for TCP and TP4 gateways have been formally specified using the ESTELL [25] formal description technique. Figure 1.7 shows how the VTL architecture provides for transport layer interoperability.

1.5 Organization of the Dissertation

Chapter 2 of this dissertation describes and compares the DOD TCP and the ISO transport protocol suites. Chapter 3 describes the VTL architecture. Chapters 4 through 6 describe the design of gateways for DoD TCP and ISO TP4 protocols. The Appendixes contain the ESTELL specification for the gateway components.

2 TRANSPORT SERVICES AND MECHANISMS

2.1 Introduction

The Transport Layer is the highest layer in a communication model which is directly involved with data communications. The network service provides routing and relaying across real subnetworks. In contrast the transport service is concerned only with communication between end systems and has no interest in the route actually taken by data. Its responsibility is to provide session entities with a reliable, cost effective means of transferring data while protecting them from the vagaries of the underlying network. An in-depth understanding of the mechanisms by which ISO transport and the DoD TCP protocol suites provide the data transport service is essential in order to understand interoperability issues. This chapter summarizes the services and functions offered by the ISO set of transport protocols as specified in ISO 8072 [26], ISO 8073 [27] [28], and ISO 8062 [29] and those offered by the DoD TCP [30] [31].

Section 2.2 describes the services and primitives of the ISO connection oriented and connectionless transports and the mechanisms by which their services are provided. The DoD TCP protocol suite is described in similar terms in Section 2.3, and the major differences between ISO TP4 and DoD TCP are listed in Section 2.4.

2.2 ISO Transport Services Overview

The transport service (TS) provides transparent transfer of data between TS users. It provides for the following:

(a) Transport Connection. Provides the means to establish a transport connection with another TS user for the purpose of exchanging TSDUs.

(b) Transferring TSDUs. Provides the means of transferring TSDUs, which consist of an integral number of octets in a transparent fashion.

(c) End-to-end Significance. The transport service provides for the transfer of data between two TS users in end systems.

(d) Quality of Service (QOS) Selection. The Transport Layer is required to optimize the use of available communication resources to provide the QOS required. QOS parameters representing characteristics such as throughput, transit delay, residual error and failure probability.

(e) Transparency of Transferred Information. The transport service hides from the users the differences in the QOS provided by the network service. This difference in the QOS arises from the use of a variety of communications media by the network layer to provide the network service.

(f) TS User Addressing. The transport service utilizes a system of addressing that is mapped into the addressing scheme of the supporting network service. Transport-Addresses can be used by TS users to refer unambiguously to transport service access points (TSAPs).

There are two models of transport service, namely connection-mode and connectionless-mode. The characteristics of each are described next.

2.2.1 Connection Mode Transport Layer Services

A connection oriented TS operation is characterized by three distinct phases: (1) T-Connection establishment, (2) T-Data transfer and (3) T-Connection release. Information is passed between a TS user and the TS provider by the service primitives, which may convey parameters. The primitives, as listed in Table 2.1, are abstract representations of TSAP interactions.

2.2.1.1 Quality of Transport Service The term *quality of service* (QOS) refers to certain characteristics of a transport connection (TC) as observed between TC end points. These parameters are attributes of the TS provider. Information about the QOS requirements of the TS users may be used by the TS provider for the purpose of protocol

Table 2.1 - Summary of COTS primitives and parameters

PHASE	SERVICE	PRIMITIVE	PARAMETERS
TC_establishment	TC_establishment	T_CONNECT request	(Called/calling address, Expedited data selection, QOS parameter set, TS_User_Data)
		T_CONNECT indication	(as in connect request)
		T_CONNECT response	(Responding address Expedited data selection, QOS parameter set, TS_User_Data)
		T_CONNECT confirmation	(as in connect response)
Data Transfer	Data_transfer	T_DATA_request	(TS_User_Data)
		T_DATA_indication	(as in data request)
	Expedited data transfer (User option)	T_EXPIDITED_ DATA_ request/indication	(TS_User_data)
TC release Request	TC_Release	T_Disconnect_	(TS user data)
	indication	T_Disconnect_ TS_User_Data)	(Reason,

selection. The QOS is normally negotiated between TS users and the TS provider on a per connection basis. The QOS requested by the calling TS user may be lowered either by the

called TS provider or by the called TS user. The negotiated QOS values then apply throughout the lifetime of the TC. QOS parameters as defined in ISO document 8072 are listed below.

(a) TC establishment delay

(b) TC establishment failure probability

(c) Throughput

(d) Transit delay

(e) Residual error rate

(f) Transfer failure probability

(g) TC resilience

(h) TC release delay

(i) TC release failure probability

(j) TC protection

(k) TC priority

2.2.2 ISO Connection Oriented TP Mechanisms

This section describes the procedure required to provide the connection oriented transport services as specified in ISO 8073. ISO 8073 specifies a set of rules expressed in terms of procedures to be carried out by peer entities at the time of communication. Addendum 2 to ISO 8073 [28] specifies the procedures to be used when operating over CLNS. The document formally specifies:

(a) Five classes of procedures when operating over Connection-Oriented Network Service (CONS):

CLASS 0: Simple class

CLASS 1: Basic error recovery class

CLASS 2: Multiplexing class

CLASS 3: Error recovery and multiplexing class

CLASS 4: Error detection and recovery class

(b) One class of procedures when operating over Connectionless Network (CLNS). Class 4 procedures are to be used for connection-oriented transfer between peer transport entities over the connectionless network service.

(c) The means of negotiating the class of procedures to be used by the transport entities.

(d) The structure and encoding of the TPDUs.

The procedures are defined in terms of the interactions between peer TP entities through the exchange of TPDUs, interactions between a TP entity and the TS user through TS primitives, and the interactions between a TP entity and the NS provider through NS primitives.

2.2.2.1 Classes and Options Over CONS The functions of the Transport Layer have been organized into classes and options. A *class* defines a set of functions and *options* define those functions within a class which may or may not be used. The use of classes and options is negotiated during connection establishment. The choice made by the transport entities depends upon the following:

(a) TS-user requirements expressed via T-CONNECT.

(b) Quality /type of network services.

(c) User required service versus cost ratio acceptable to the TS-user.

The network services are classified in terms of quality with respect to error behavior in relation to user requirements. The purpose is to provide a basis for the decision regarding which class of transport protocol should be used in conjunction with the given network connection.

TYPE A: Acceptable residual error rate and acceptable rate of signalled errors.

TYPE B: Acceptable residual error rate but unacceptable rate of signalled errors.

TYPE C: Unacceptable residual error rate and unacceptable rate of signalled errors.

Each TP entity needs to be aware of the quality of service provided by the NS provider. The characteristics of the various ISO TP classes are described next.

2.2.2.2 Class 0 Characteristics Class 0 is designed to have minimum functionality. It provides only the functions needed for connection establishment with negotiation, data transfer with segmenting and protocol error resetting. Class 0 provides flow control based network service provided flow control and disconnection based on network service disconnection. Class 0 has been designed to be used with type A network connections (NC).

2.2.2.3 Class 1 Characteristics Class 1 provides the functionality of Class 0 plus the ability to recover after a failure signalled by the Network Service. Class 1 allows data transfer with flow control based on network service provided flow control, error recovery, expedited data transfer and the ability to support consecutive transport connections on a network connection. Class 1 is designed to be used with type B NCs.

2.2.2.4 Class 2 Characteristics Class 2 provides a way to multiplex several transport connections onto a single network connection. TCs can be used with or without explicit flow control. No error detection or recovery is provided. Class 2 is also designed primarily for use with type A NCs.

2.2.2.5 Class 3 Characteristics Class 3 provides the functionality of class 2 plus the ability to recover after a failure signalled by the Network Layer without involving the TS-User. It has been designed for use with type B NCs.

2.2.2.6 Class 4 Characteristics over CONS Class 4 allows operation both over CO and CL network service. While operating over CONS, class 4 provides the functionality of class 3, plus the ability to detect and recover from lost, duplicate, or out of sequence TPDUs without involving the TS-user. Class 4 detects signalled and unsignalled network failures and recovers from these failures by using time-out mechanisms. Damaged TPDUs are detected by using a checksum mechanism. The detection of errors is made by use of TPDU numbering, by time-out mechanisms and by additional procedures. Class 4 is designed to operate over type C network connections.

2.2.2.7 Classes and Options Over CLNS ISO specifies the use of only Class 4 for providing a connection oriented transport service over a CLNS. While operating over a CLNS, class 4 provides flow control between peer transport entities. The resilience inherent in class 4 allows operation over a low grade service available over a CLNS.

2.2.3 ISO TP Procedures

Table 2.2 lists the procedures by which ISO Transport protocols provide for data transfer and their inclusion in a particular class. The detailed description is given in ISO 8073.

2.2.4 Connectionless Mode Transport Services

A defining characteristic of transport connectionless mode transmission is the independent nature of each invocation of the Transport Service. TSDUs are transmitted from a source TSAP to a destination TSAP outside the context of a transport connection and without any requirement to maintain any logical relationship among multiple TSDUs. The purpose of connectionless transport is to allow the transfer of data between corresponding TS-Users on a connectionless basis. This service provides for data transfer without the overhead of transport connection. It is primarily intended to benefit those applications that require a one time, one way transfer of data.

Table 2.2 - Connection Oriented ISO-TP Sub-Functions

Sub-Function		CLASS 0	1	2	3	4	4(clns)
Assignment to NC		Y	Y	Y	Y	Y	N
Connection Establishment		Y	Y	Y	Y	Y	Y
Connection Refusal		Y	Y	Y	Y	Y	Y
Association of TPDUs with a TC		Y	Y	Y	Y	Y	Y
TPDU Transfer		Y	Y	Y	Y	Y	Y
TPDU Numbering	Normal	N	Y	Ym	Ym	Ym	Ym
	Extended	N	N	Yo	Yo	Yo	Yo
Expedited Data Transfer	NW normal	N	Ym	Y	Y	Y	Y
	NW expedited	N	Yo	N	N	N	N
Retention until ack of TPDU							
Segmentation and Reassembling		Y	Y	Y	Y	Y	Y
Concatenation and Separation		N	Y	Y	Y	Y	Y
Normal Release	Implicit	Y	N	N	N	N	N
	Explicit	N	Y	Y	Y	Y	Y
Error Release		Y	N	Y	N	N	N
Reassignment After Failure		N	Y	N	Y	Y	N
Resynchronization		N	Y	N	Y	Y	N
Multiplexing and Demultiplexing		N	N	Y	Y	Y	N
With Explicit Flow Control		N	N	Ym	Y	Y	Y
Without Explicit Flow Control		Y	Y	Yo	N	N	N

Table 2.2 - (Cont.)

Sub-Function	CLASS					
	0	1	2	3	4	4(clns)
Use of Checksum	N	N	N	N	Ym	Ym
Non-use of Checksum	Y	Y	Y	Y	Yo	Yo
Frozen References	N	Y	N	Y	Y	Y
Retransmission on Timeout	N	N	N	N	Y	Y
Resequencing	N	N	N	N	Y	Y
Inactivity control	N	N	N	N	Y	Y
Treatment of Protocol Errors	Y	Y	Y	Y	Y	Y
Splitting and Recombining	N	N	N	N	Y	N

N: Procedure not applicable

Y: Procedure always included in class

Ym: Negotiable Procedure whose implementation in equipment is mandatory

Yo: Negotiable Procedure whose implementation in equipment is optional.

Addendum 1 to ISO 8072 defines the connectionless mode transport service (CLTS). The CLTS primitives are summarized in Table 2.3.

2.2.4.1 QOS for CLTS For CLTS, no negotiation of the QOS takes place. No dynamic association is set up between the parties involved. Thus the TS user needs to have

explicit knowledge of the characteristics of the service it can expect to be provided with each invocation of the service. The QOS parameters identified for CLTS are listed below.

(a) Transit delay.

(b) Residual error rate.

(c) TC protection.

(d) TC priority.

Table 2.3 - Summary of CL-TS primitives and parameters

PRIMITVE	PARAMETERS
T_UNITDATA_Req	Source Address Destination Address QOS TS-User-Data
T-UNITDATA_ind	Source Address Destination Address QOS TS-User-Data

2.2.5 ISO Connectionless TP Mechanisms

This section describes the procedure required to provide the connectionless transport services (CLTS) as specified in ISO 8062. The functions in the transport layer bridge the gap between the service available from the Network layer and the services to be offered to the transport service users. The functions of CLTS are:

(a) Transmission of TPDUs.

(b) Network service selection.

(c) Address mapping: Determine the network address that will be used as the destination address in an N-UNITDATA request (over CLNS) or the called address in N-CONNECT request (over CONS).

(d) TPDU delimiting: Determine the beginning and end of a TSDU.

(e) Error Detection: Provide end-to-end error detection.

The procedures used to transfer data depend on the type of Network Service available, namely CLNS or CONS. Table 2.4 lists the procedures used by ISO CLTS.

2.2.5.1 Transfer over CLNS In the case of CLNS, no network connections have to be maintained. Each TPDU is transmitted over a pre-existing association between a pair of NSAPs. There is no indication given to transport entities about the ability of the network entity to fulfill the service requirements given in the N-UNITDATA primitive.

2.2.5.2 Transfer over CONS When operating over CONS, the transport entity has to go through explicit connection management procedures. The duration for which a NC is kept open is not defined in the ISO document and is left as implementation dependent.

2.3 TCP Data Transport Mechanisms

DoD Transmission Control Protocol is designed to provide reliable communication between pairs of processes in logically distinct hosts on a network. TCP provides connection oriented, reliable ordered, full duplexed and flow controlled data transfer. TCP implementations are prolific, and it was the forerunner of the ISO TP technology. Due to its installed base and popularity, TCP is chosen as one of the transport architectures for which a gateway to the Virtual Transport Layer will be designed. This section summarizes the services and functions offered by the DoD transport mechanism as specified in MIL-STD-1779 (namely the Transmission Control Protocol - TCP).

2.3.1 DoD TCP Services

TCP is designed to provide reliable communication between pairs of processes in logically distinct hosts on networks. TCP will operate successfully in an environment where loss, damage, misorder of data and network congestion can occur. It thus provides a connection oriented data transfer that is reliable, ordered, full duplex and flow controlled. The upper layer protocols (ULPs) can channel continuous streams of data through TCP. The services provided by TCP are organized as follows:

(a) Connection management service.

(b) Data transport service.

Table 2.4 - Procedure For ISO CLTS

PHASE	PROCEDURE
Data Transfer (Over CLNS)	Send Unit_Data (UD)
	Receive Unit_Data
Data Transfer (Over CONS)	Establish Network Connection
	Send UD_TPDU
	Receive UD_TPDU
	Release Network Connection
Other	Checksum
	Discard TPDUs

(c) Multiplexing service.

(d) Error reporting service.

A description of the service is presented below. The mechanisms used to provide these services are presented in the subsequent sections.

(a) Connection Management: A TCP connection provides a communication channel between a pair of ULPs. Connection management is subdivided into three phases: connection establishment, connection maintenance and connection termination.

(b) Data Transport: TCP provides data transport over established connections between ULP pairs. The data transport is full duplex, timely, ordered, labeled with security and precedence levels, flow controlled, and error checked.

(c) Multiplexing Service: TCP provides services to multiplex pairs of processes within upper layer protocols. A process within a ULP using TCP service shall be identified with a *Port Number*. A port when concatenated with an Internet Address forms a network-wide unique *Socket*.

(d) Error Reporting Service: TCP report service failure stemming from catastrophic conditions in the internetwork environment for which TCP cannot compensate.

2.3.2 TCP Service Interaction Primitives

TCP interaction primitives are grouped into *Service Request Primitives* and *Service Response Primitives*.

TCP service request primitives enable connection establishment, data transfer and connection termination. Table 2.5 lists the parameters associated with the service requests. The TCP specification describes the following request primitives:

(a) Unspecified Passive Open

(b) Fully Specified Passive Open

(c) Active Open

(d) Active Open With Data

(e) Send

(f) Allocate

(g) Close

(h) Abort

(i) Status

TCP service response primitives enable TCP to inform user of connection status, data delivery, connection termination and error conditions. The TCP specification describes the following response primitives.

(a) Open ID

(b) Open Failure

(c) Open Success

(d) Delivery

(e) Closing

(f) Terminate

(g) Status Response

(h) Error

Table 2.6 lists the parameters associated with the service responses. The primitives are categorized according to the phase or state of the connection.

Table 2.5 - Summary of TCP Service Primitives and Parameters

SERVICE PRIMITIVE	DESCRIPTION	PARAMETERS[a]
CONNECTION ESTABLISHMENT		
Unspecified Passive Open	Respond to connection attempts from an unnamed ULP	Source Port *ULP Timeout *ULP Timeout_Action *Precedence *Security_Range
Fully Specified Passive open	Respond to connection attempts from a fully named ULP	Source Port Destination Port Destination Address *ULP Timeout *ULP Timeout_Action *Precedence *Security_Range
Active Open	Initiate a connection attempt to a named ULP	Source Port Destination Port Destination Address *ULP Timeout *ULP Timeout_Action *Precedence *Security_Range
Active Open with data	Initiate a connection attempt to a named ULP accompanied by specified data.	Source Port Destination Port Destination Address *ULP Timeout *ULP Timeout_Action *Precedence *Security_Range Data Data Length PUSH flag URGENT flag

[a] All parameters marked * are optional.

Table 2.5 - (Cont.)

SERVICE PRIMITIVE	DESCRIPTION	PARAMETERS
DATA TRANSFER		
Send	Data Transfer across the named connection	Local_Connection_ name Data Data Length PUSH flag URGENT flag *ULP Timeout *ULP Timeout_Action
Allocate	Indicates the additional number of octets the ULP is willing to accept	Local_Connection_ Name Data Length
CONNECTION TERMINATION		
Close	Data transfer completed across named connection	Local_Connection_ Name
Abort	Named connection is to be terminated immediately	Local_Connection_ Name
STATUS		
Status	Query for current status of named connection	Local_Connection_ Name

Table 2.6 - Summary of TCP Service Responses and Parameters

SERVICE PRIMITIVE	DESCRIPTION	PARAMETERS
PHASE CONNECTION ESTABLISHMENT		
Open ID	Informs ULP of Local Connection Name assigned by TCP	Local_Connection_ Name Source Port Destination Port Destination Address
Open Failure	Informs ULP of failure of Active Open Request	Local_Connection_ Name
Open Success	Informs ULP of completion of one of the Open Service Requests	Local_Connection_ Name
PHASE DATA TRANSFER		
Deliver	Informs ULP of data arrival across the named connection	Local_Connection_ Name Data Data Length URGENT flag
PHASE CONNECTION TERMINATION		
Closing	Informs ULP of peer ULPs CLOSE service request	Local_Connection_ Name
Terminate	Named connection has been terminated as a result of remote connection reset or service failure	Local_Connection_ Name

Table 2.6 - (Cont.)

SERVICE PRIMITIVE	DESCRIPTION	PARAMETERS
STATUS		
Status Response	Return current status of named connection	1. Local_Connection_ Name 2. Source port 3. Source Address 4. Destination Port 5. Destination Address 6. Connection State 7. No. of octets that can be accepted by local ULP 8. No. of octets that can be sent to remote ULP 9. No. of octets awaiting ack. 10. No. of octets pending receipt by the local ULP 11. Urgent State 12. Precedence 13. Security 14. ULP timeout
Error	Informs ULP of illegal service requests, or of errors relating to the environment	Local_Connection_ Name Error Description

2.3.3 DoD TCP Procedures

TCP mechanisms are motivated by TCP services as described in the previous section. The mechanisms present in the TCP entity are listed below:

(a) Flow control windows

(b) Duplicate and out-of-order data detection

(c) Positive acknowledgements with retransmission

(d) Checksum

(e) Push

(f) Urgent

(g) ULP timeout

(h) ULP timeout action

(i) Security and precedence

(j) Security ranges.

(k) Multi-addressing

(l) Passive and active open requests

(m) Three way handshake for SYN exchange

(n) Open request matching

(o) Three way handshake for FIN exchange

(p) Resets

The selection of mechanisms, as formally specified in the DoD TCP document, to support a service is guided by design standards including simplicity, generality, flexibility and efficiency.

Table 2.7 - Differences between the OSI TP4 and TCP

Feature	OSI TP4	TCP
Number Of TPDU types	10	1
Connection Collision	2 Connections	1 Connection
Addressing Format	Not defined	32 bits
Quality of Service	Open Ended	Specific Options
User Data in CR	Permitted	Not Permitted
Data Stream	Messages (PDUs)	Octets
Emergency Data	Expedited	Urgent
Piggybacking AK/data	No	Yes
Explicit Flow Control	Negotiable	Not Negotiable
Subsequence Numbers	Permitted	Not Permitted
Connection Release	Abrupt	Graceful

2.4 TCP ISO TP4 Differences

TP4 and TCP have numerous similarities, but also some differences. Both protocols are designed for providing a reliable connection oriented, end-to-end service on top of an unreliable network that can potentially lose, corrupt, delay or duplicate packets. The two protocols are also alike in that both have a connection establishment phase, a data transfer phase and a connection release phase (although some details differ). However the two

protocols have some notable differences as listed in Table 2.7. Details are given in the following chapters when the various interoperability issues are described.

2.5 Conclusion

This chapter described the provision of data transport by the ISO transport and the DoD transmission control protocol suites. An understanding of the transport mechanisms and the usage model of the service is essential while designing a transport interoperability architecture. It becomes evident from the difference in the semantics of some of the services provided by TCP and TP4, that a simple translation scheme between the two protocols will not allow interoperability. A usage model of the services needs to be defined before a conversion on the TPDU level can provide interoperability. Further it should be evident that the design of a protocol convertor would require an intimate knowledge of both the protocol architectures.

The next chapter introduces the Virtual Transport Layer concept and an architecture to support it.

3 THE VTL ARCHITECTURE

3.1 Introduction

This chapter presents the Virtual Transport Layer (VTL) concept and describes how interoperability at that transport layer is achieved. An architecture for the VTL gateways is also proposed. The advantages and disadvantages of some of the current approaches that try to provide transport layer interoperability were discussed in Chapter 1. Section 3.2 describes the VTL approach and how our solution to the transport layer interoperability problem will attempt to solve some of the issues not resolved by other methods. The gateway architecture is described in Section 3.3. An addressing scheme is proposed in Section 3.4.

3.2 Interoperability using the Virtual Transport

The Virtual Transport Layer approach attempts to solve the transport interoperability problem by mapping the end transport protocols and messages into those of an intermediate meta transport. The mapping is done on both in syntax and semantics.

The mapping of syntax of end transport messages is done by generating an equivalent message, using a TPDU translator, which can be understood in the VT-Domain. These messages are referred to as Virtual Transport Data Units (VTPDUs). The TPDU translator component of the gateways is referred to as the transport convergence function (TCF).

TCFs in the gateways can participate in a peer protocol exchange. This may be needed if the set of VTPDUs chosen can not provide a certain functionality requested by the end transport, for error recovery, or management functions.

Figure 3.1 shows the proposed environment. The module identified as the transport convergence function in the gateway architecture changes when interoperability with a new transport is required. Thus it is possible to come up with design guidelines for the TCF to provide interoperability for other transport protocols. The complexity of the TCFs in each gateway is then limited to go between the end system transport and the virtual transport.

The VTPDUs are transferred to the destination gateway using a CLTS, thus leaving the responsibility of providing the reliability to the end transports. This seems to be the logical choice to go between TCP and TP4, and other such transport protocols that are designed to operate over an unreliable network. For the TCFs to participate in a peer protocol exchange, a reliable connection oriented path is provided by employing a connection oriented transport service.

3.3 Gateway Architecture Design Issues

The design of the VTL architecture is guided by the requirements imposed during the various phases of a transport connection, namely transport addressing, connection

establishment, data transfer and connection termination. The design issues that need to be resolved are enumerated below:

(1) Interoperability is provided by converting end system TPDUs to a common format at each end system interface. The syntax and the semantics of these common message objects (VTPDUs) needs to be defined.

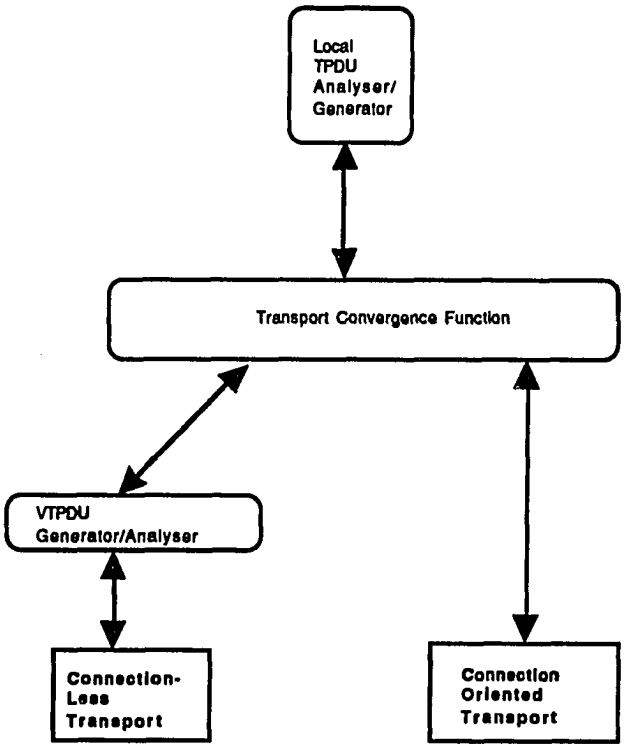


Figure 3.1 - VTL Gateway Architecture

(2) Address resolution, when the target system has a different representation of the transport address.

(3) The TCFs in the gateways need to provide the following functionality:

(a) Local TPDU syntax analyzer.

(b) Mapping functions and association control blocks to convert protocol specific information such as sequence numbers, acknowledgements, credit allocation and reference id's from the local format to the common format.

(c) VTPDU generator.

(d) The end-to-end reliability will be provided by the hosts themselves. Thus, packets lost in the VT domain will be compensated for by the end systems timeouts and subsequent retransmissions. To provide for reliability is a not a requirement.

(e) The TCFs can provide store and forward service if a simple syntactical conversion between the the local format and common format is not possible. In this case the TCF will also need to know the semantics of the TPDUs. Thus a minimal TPDU semantic analyzer may be required.

(f) The TCFs can employ a peer protocol if the need arises and exchange PDUs over an out of band transport connection.

The subsequent chapters of this dissertation describes the design of the VTL gateways. The design effort is broken down into three steps, wherein the design is guided by the requirements of the target transport architectures during the connection establishment phase,

data transfer phase and the connection termination phase. The next section describes a method by which addressing can be resolved in an heterogeneous network environment. Some techniques have been discussed in [32].

3.4 Addressing Mechanism

Before describing the cross network connection establishment phase between the two transport protocols described in the scope of this research, it is essential to discuss the fundamental issue of cross domain addressing. An overview of the addressing mechanism employed by DoD TCP/IP and ISO Network layer is presented.

3.4.1 TCP/IP Addressing

A transport end point is uniquely identified in the Internet by a "Socket". A socket is a concatenation of a Transport Port and an Internet Address. The Internet Address (popularly called IP Number) uniquely identifies a host machine in the Internet, and the Port is used to identify a process in the machine.

PORT: 2 Byte (positive integer)

IP Address: 4 Bytes

The semantics associated with the IP address is as Follows :

IP Address: NetworkID: HostID

The NetworkID portion of the IP address indicates a unique network (i.e., some organizational network which forms a part of the Internet), and the remaining portion of the IP address points to a unique machine in that network. Thus routers and gateways look at the NetworkID portion of the destination IP address and consult their routing tables to determine the outgoing link on which to send the packet. Thus in the Internet, the destination Network ID is of significance until it reaches a gateway which is a host in the destination network. Then the Host ID is used to determine the physical address of the target machine (using the ARP method).

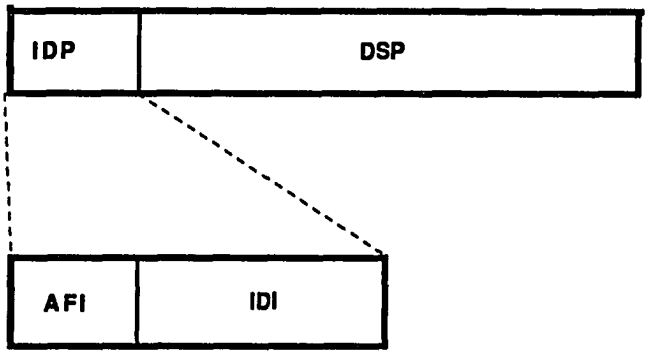
All IP addresses (if access is desired to the Internet) are provided by a central authority. The Network Information Center (NIC) located at SRI International assumes that role. NIC only assigns the network portion of the IP address and delegates responsibility for assigning host addresses to the requesting organizations.

3.4.2 ISO Network Layer Addressing

Just as in the TCP/IP architecture, an ISO transport end point is uniquely identified by a Transport Address. The Transport Address is a concatenation of a Transport Selector (equivalent to a port) and an NSAP Address. The differences being:

(a) The T_SELECTOR is not of fixed size, but has a Length-Value format. Implementation agreements limit it to a maximum of 32 Bytes.

(b) The NSAP address format is more elaborate as shown in Figure 3.2.



IDP: Initial Domain Part
AFI: Authority and Format Identifier
IDI: Initial Domain identefier
DSP: Domain Specific Part

Figure 3.2 - ISO NSAP Address Format

A description of the various components is as follows:

AFI: The AFI specifies the Network Authority responsible for allocating values of the IDI. It also determines the interpretation of the Initial Domain Identifier (IDI), both in syntax and semantics. The AFI also indicates whether the DSP is formatted using decimal or binary digits.

IDI: The Initial Domain Identifier specifies the network addressing domain from which values of the DSP are allocated and the Network Authority responsible for allocating values of the DSP from that domain.

DSP: The syntax and semantics of the Domain Specific Part is determined by the AFI.

ISO Network Layer addressing is highly flexible and is designed to allow for expansion. An example of an NSAP Address is:

Example 1

AFI: 36

IDI: 09115152927580

DSP: NULL

An AFI of 36 implies that the address is CCITT X.121 address, and the IDI consists of a sequence of upto 14 digits allocated according to CCITT recommendation X.121. The DSP is empty.

Example 2

AFI: 47

IDI: 04

DSP: (Subnet : SNPA : dlsap : nsel)

An AFI of 47 specifies the network authority as National Institute of Standards and Technology (NIST). NIST maintains an experimental OSI network (called OSINET). The IDI is an International Code Designator (ICD). The format of the DSP is as shown above. The SNAP is the hardware address of the target machine. The DLSAP is the LLC SAP being used and the NSEL is the Network Layer Protocol ID.

3.4.3 Addressing Across Heterogeneous Architectures

The aim of this research effort is to allow a transport end point in any OSI domain to be able to exchange service data units with a transport end point in the TCP/IP domain transparently. To maintain complete transparency, it should be made to appear to the end systems as if they are attempting to connect to another system in the same network architectural domain.

Thus a name resolution invoked by a TCP client, for a target on an ISO transport, should return a TCP /IP address. This implies that the end systems with which interoperability is desired must get an IP Address from an Internet Naming Authority (even though they are OSI nodes). This is being proposed due to the following reasons :

(1) In general it may not be possible to reuse an OSI TSAP address as a TCP Socket because of a difference in the upper limit of the size.

(2) IP addresses have a specific meaning as far as routing is concerned. The semantics of an IP address is fixed unlike that of an ISO NSAP address where the AFI specifies the semantics and syntax. So a non conforment IP address my be confusing.

Figure 3.3 shows a TCP client making a name resolve request for a destination on a ISO TP4. A proxy TCP address is returned as a response to the name request. The gateway connected to the TCP/IP subnets then append an AFI to the source and destination address which specifies that the NSAP address is an IP address with an NetworkID and a HostID. Upon reaching the

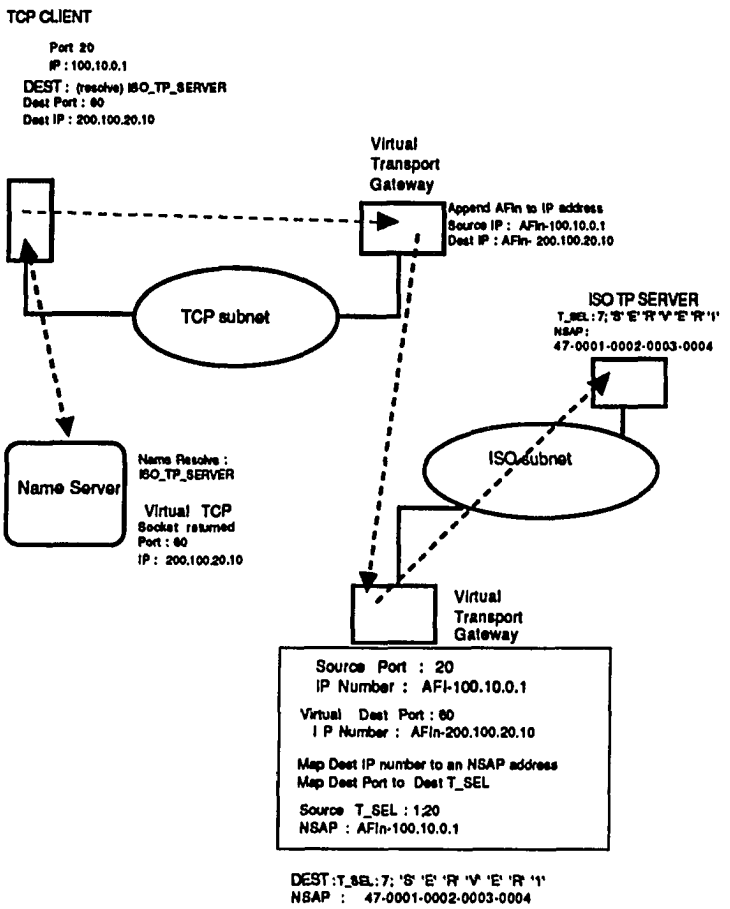


Figure 3.3 - Address Resolution (TCP Client ISO Server)

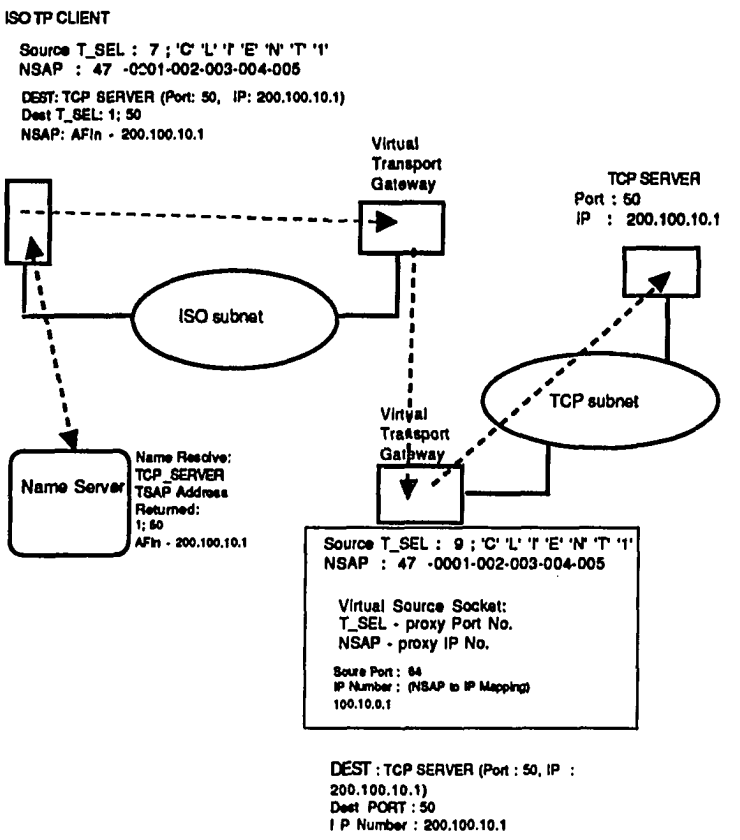


Figure 3.4 - Address Resolution (ISO Client TCP Server)

destination gateway, the destination port and the Host ID part can be used to map to the actual OSI TSAP address of the target transport end point, or can be used as such if the target happens to be on a TCP/IP subnet. Thus the gateways must maintain a data base between the proxy TCP addresses and the actual ISO transport address.

On the other hand a OSI Transport client can use the target socket addresses after prefixing the IP number with an AFI. This is acceptable because a TCP port can fit into a T-Selector, and an IP number prefixed with an AFI can be unambiguously interpreted. This implies that the name request, for a target on a TCP/IP network, returns a IP address with an AFI appended to it as the NSAP address. The destination gateway, as shown in Figure 3.4, on recognizing that the AFI of the calling NSAP address specifies a non-TCP/IP host, replaces the source TSAP address by a proxy socket address (as per the original assumption that all OSI hosts get IP addresses from an Internet Naming Authority) If the AFI of the calling NSAP address specifies a TCP/IP host, then no modification of the source TSAP address is done.

3.5 Conclusion

This chapter described the Virtual Transport Layer and a gateway architecture to support the concept. The design guidelines for the gateways were formulated. The approach maps the end host TPDUs to a common intermediate format. The gateways that connect the various subnets are only aware of the nature of the transport protocol being used by the hosts in the adjoining subnet. They do not attempt to determine the nature of the target transport protocol. The gateway architecture specified employs a Connectionless Transport to transfer

the intermediate format protocol data units between the gateways. The issue of transparent addressing has also been dealt with. An end system sees its peer due to the joint participation of the gateways and the actual target transport entity. The next chapters formally describes the VTL and the gateways from TCP and ISO TP4 architectures.

4 CONNECTION ESTABLISHMENT PHASE

4.1 Introduction

This chapter addresses the issues of connection management in an environment where transport interoperability is provided by mapping host transport protocol data units to a common format. The mapping is done with the objective of retaining the end-to-end significance of primitives without the knowledge of the nature of the destination transport protocol. The Virtual Transport concept was introduced in the previous chapter. For this research effort, the target transports have been identified as the ISO transport class 4 and the DoD TCP.

Section 4.2 specifies the services needed from the Virtual Transport to satisfy the Connection Establishment Phase. A usage model of the transport services in order to facilitate interconnection is presented in Section 4.3. An informal description of the role of the VTL during connection establishment is described in Section 4.4. Section 4.5 presents a formal specification of the gateway components for DoD TCP and ISO TP4. The specification is done using the ESTELL FDT and is contained in Appendix A.

4.2 Connection Establishment Phase

The connection establishment phase of both TCP and ISO TP4 is based on the *three way handshake* principal [33], but there are some notable differences in the services provided. These are explained below.

4.2.1 Sequence of Events

The TCP architecture is based on the concept of an active initiator and a passive listener. The active initiator issues a Synchronize (SYN) Request segment. The distinguishing features of the SYN segment are:

(a) Source Port

(b) Destination Port

(c) Initial Sequence Number (ISN)

(d) Advertised Receive Window

(e) Maximum Receive Segment Size

(f) User Data

(g) And possibly Urgent Data

Detection of duplicate/invalid SYN segment is done as follows. The initiator generates a SYN segment with an *Initial Sequence Number*. The listener records the sequence number of

the incoming SYN segment and responds with another SYN carrying its ISN, along with an ACK which acknowledges the SYN that it just received. At this point, the listener is not sure about the validity of the received SYN segment. It could be a duplicate or delayed packet. When the initiator gets a SYN segment with an ACK number supposingly acknowledging the SYN it had sent, it checks to see if the ACK number falls inside the send window. If it does, then the SYN and ACK are accepted as valid and an ACK is transmitted completing the three way handshake. If the ACK is invalid, then the initiator responds with a RESET segment. If the listener receives a valid ACK to the SYN it had sent out, then it is assured of the validity of the SYN it had received and the connection proceeds. Figure 4.1 shows the sequence.

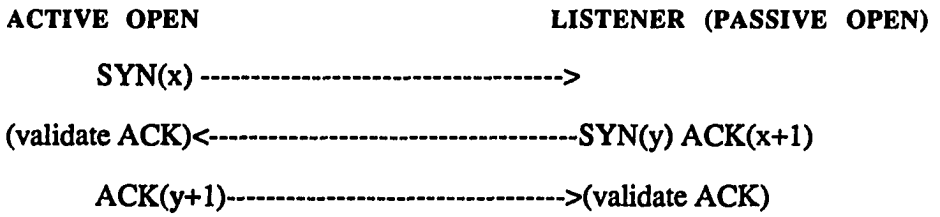


Figure 4.1 - TCP three way handshake

In summary, a connection is established after exchanging SYN segments with ACKs to validate them.

ISO TP4 uses a Connection Request (CR) TPDU to initiate a connection. A CR TPDU carries the following important information:

(a) Credit Allocation

(b) Source Reference

(c) Destination Reference set to zero

(d) Class and options

(e) User Data

The ISO transport provides a service interface to the client. Invocation of the connection request service primitive causes TP4 to generate a CR TPDU. A unique Source Reference is generated. *This reference ID along with the source and destination NSAP addresses completely specify the initiating transport end point.* In fact this property is used by the remote TP entity to determine the validity of the received CR TPDU.

On the responder side, unlike TCP, there is no passive open. Instead a connection indication is generated and given to the client. The client responds with a connection response, upon which the TP entity generates a Connection Confirm (CC) TPDU with the following information:

(a) Credit

(b) Destination Reference ID

(c) Source Reference ID

(d) Class and Options

(e) User data.

The CC TPDU is than acknowledged by the initiator, as shown in Figure 4.2, completing the three way handshake.

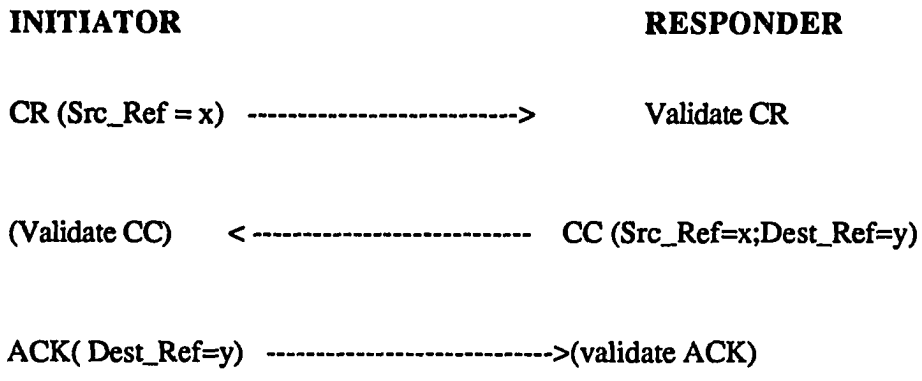


Figure 4.2 - ISO TP Connection Establishment

4.2.2 Connection Establishment Procedures for TP4 and TCP

Although both TCP and TP4 use three way handshake as a basis for reliable connection establishment, there are some notable differences. This section highlights the differences in philosophies behind the two protocols, so as to aid in the formal specification of the role and design of the Transport Convergent Function (TCF).

4.2.2.1 Role of Responder In the case of TCP, the client who recognizes its role as that of a responder does a Passive Open. This causes the TCP entity to enter a

Listening State, waiting for a SYN segment to arrive. When the SYN does arrive, the state transitions progress without the clients intervention until the connection is established.

ISO transports do not provide the facility to post a Listen. An incoming CR is conveyed to the client as a connection indication event. It is at the clients discretion to accept or reject the connection.

4.2.2.1 Data in Connection Request Both protocol specifications allow data in connection request, but the manner in which connection data are handled by the receiving transport entity is different. In TCP, the listening TCP buffers the data until its SYN is acknowledged (completion of the three way handshake). The reason for this is that the TCP specification does not allow for informing the client of an incoming connection request. The advantage of this approach is that it shields the client from receiving an invalid connection indication, i.e., one originated by a peer client which for some reason is no longer active. The successful completion of the three-way handshake filters out cases of duplicate and delayed/stray SYN segments. There is no limit on the amount of data accompanying the SYN segment, as long as it is smaller than the maximum segment size.

ISO transports, on the other hand, filter out only duplicate CR TPDUs before generating a connection indication for the client, along with any data that accompanies the connection request. This approach allows the client to reject connections if it so desires. The data in connection indication can be used as a basis for accepting or rejecting the connection. Data in a connection request/response are limited to 32 Bytes. This limit is imposed by the fact that the CR TPDU is limited to 128 bytes. The 128 byte CR TPDU limit stems from the fact

that it is forbidden to fragment a CR TPDU. Fragmentation may occur if the underlying layers have smaller PDU sizes.

4.2.2.3 Simultaneous Connection Requests In TCP, the end systems Transport Ports involved in the connection establishment are both identified in the SYN segment. As a result a, SYN that is received when the TCP entity is in the SYN_SENT state does not cause any ambiguity. The state machine is designed such that both entities cycle to the ESTABLISHED state. A single connection is the net result.

In the case of ISO transports, the distinguishing feature of the CR TPDU is a unique Source Reference ID (the Destination Reference ID is zero). Thus the TP entity cannot associate a connection control block with the connection request that it receives and treats it as a request for a new connection. It is left up to the client to recognize that it is receiving a connection indication (by examining the called and calling T_Selectors) from a peer to which it has sent a connection request. There is a potential of two different connections to be established.

4.2.2.4 Other Differences The two protocols differ in unit of data. TCP is based on octet boundaries, while ISO transports uses packets or frames. Thus sequence numbers, acknowledgements and credit allocations reflect a byte count in the case of TCP and a packet count in the case of ISO transports.

4.3 Usage Model of ISO Transport Service

ISO 8073, the connection oriented transport protocol specification leaves the interpretation of a number of options and parameters as user defined. This can pose a dilemma for implementors and more so for the interoperability issue that is being addressed here. To provide a consistent understanding of the specifications, NBS holds workshops that provide a stable implementation agreements of OSI protocols [34]. A set of protocol agreements for ISO transport have also been proposed. Some of the agreements which aid in defining the usage model are listed below

(a) All implementations propose using the extended format (4 octets) for sequence numbers in the connection request, but they must be able to accept a request to use normal format (1 octet).

(b) ISO 8073 leaves the interpretation of the security parameter to be user defined. The NBS agreement specifies that implementations should not send the security parameter in the CR TPDU. If received in a CR, it should be ignored.

(c) All implementations must be able to operate with checksum if requested.

(d) Throughput, priority, and transit delay are optional in the ISO specification. NBS agreements specifies not to use them in the CR TPDU and ignore them in the CC TPDU.

(e) User data in CR and CC TPDU are optional. NBS agreements specify not to send data in the CR, but implementation should be able to accept data in the CC TPDU.

(f) Any unknown parameter in the CR TPDU is ignored.

The NBS agreement discourage the use of data in CR. This forbids the TP Clients to import any significance to data in CR. This research effort will restrict itself to NBS compliant ISO networks.

4.4 VTL Design Issues

This section presents the responsibilities that the VTL assumes in order to provide interoperability between ISO TP4 and TCP during the connection establishment phase. The aim will be to provide interoperability in a transparent and non-restrictive manner. Design issues pertaining to connection establishment are listed below:

(a) Format of the Connection Request VTPDU.

(b) Transfer functions to map between credit allocations, sequence numbers, acknowledgements.

(c) Conveying operational parameters such credit allocation, acknowledgments for the remote TCF to interpret and attempt to map to those available in the local transport.

d) Usage model of services by end systems.

The sequence of events with the Virtual Transport in place can be informally summarized as follows. A SYN segment (with or without data) is received by the VTL

Gateway. The local PDU syntax analyzer component of the gateway recognizes the SYN segment and commences translation of parameters to the common format as shown below:

TCP Parameters	Common Format VTPDU Parameters
SYN	--> Connection Request Code
Initial Sequence Number	--> Local Reference
Acknowledgement Number	--> Remote Reference (set to zero)
Advertised Window (Octets)	--> Credit Allocation (Number Of Packets) (where a packet size is previously defined for the gateway)
Source Port	--> Source Identifier
Destination Port	--> Destination Identifier
User Data	--> Data
OPTIONS	
Maximum Segment Size	--> Maximum Packet Size
CheckSum	--> Use_CheckSum Option
Precedence Level	--> Default Precedence
Security Level	--> Unclassified --> Allow_Expdt_Data = TRUE --> Sequence Space = 4 Bytes

The SYN segment carries an ISN which will uniquely identify the connection through its lifetime. ISO TPs have a similar identifier known as the Source Reference Id.

Recomputation of the checksums at the TCFs for the VTL domain can be a configurable parameter. If the checksum is used in the original PDU, then a flag is set which implies that a checksum is being used by the end system.

Besides the parameters carried in the SYN segment, there are two IP parameters which are of significance during the TCP connection establishment phase: namely Precedence and Security. A three bit field, in a single octet (Type of Service) is used specify precedence, with values ranging from 0 (normal precedence) to 7 (network control). Most hosts and gateways ignore the type of service. For the purpose of this research effort, operation at *Normal Precedence* is assumed.

IP Security information is carried as part of the IP Options field. RFC 1038 [35] describes the DoD Basic and Extended IP security options. The use of this option requires that a host be aware of the classification level or levels at which it is permitted to operate, and the protection authorities responsible for its certification. The achievement of this is implementation dependent. For the purpose of this research effort, a default or *Unclassified* mode of operation will be assumed.

TCP has the concept of urgent data, and the specification (for the sake of completion) allows for urgent data in the SYN segment[1] . The usage and resolution of the urgent mechanism will be described during the data transfer phase. To convey the fact that such emergency data are to be allowed, an options flag ALLOW_EXPDT_DATA is set in the VTPDU.

---

[1] In practice, most TCP implementations do not send data or Urgent data in SYN Request. This is so because the sender may be transmitting into a closed window.

ISO transports can use a 1 byte (normal format) or a 4 byte (extended format) sequence number, where as it is always 4 bytes for TCP. This information is conveyed for remote gateway to interpret.

The translation functions and the format of the VTPDU should be designed such that there is no loss of any critical information when the destination gateways translate the VTPDU back to the local format, regardless of the nature of the end transports.

Thus the sequence:

TCP_SYN_REQ -> VTL_CON_REQ -> ISO_CON_REQ

OR

TCP_SYN_REQ -> VTL_CON_REQ -> TCP_SYN_REQ

should produce valid results. If some information cannot be conveyed by a simple translation of messages, then the TCFs may have to engage in a peer protocol message exchange. The guidelines for use of the TCFs and messages thus exchanged will be specified in the sections that follow. The VTPDUs are transmitted using a CLTS between the gateways as was described previously .

When the VTPDU reaches the destination gateway, the following transformations take place to generate an equivalent TPDU. In this case we assume an ISO subnet as the adjoining network (the gateway is aware of the nature of its adjoining network).

Common Format VTPDU Parameters		ISO-TP4 Parameters
Connection Request Code	--->	CR code
Local Reference	--->	Source Reference

Remote Reference	--->	Destination Reference
Credit Allocation	--->	Credit
Source Identifier	--->	Calling TSAP ID
Destination Identifier	--->	Called TSAP ID
Data	--->	If > 32 bytes Reject Else User Data
OPTIONS		
Maximum Packet Size	--->	Closest Valid TPDU Size
Precedence	--->	If Normal then ignore Else Map to Priority
Security	--->	If Normal then ignore Else Map to Protection Parameter
Use_CheckSum	--->	If True, set use CheckSum Option
Allow_Expdt_Data	--->	If True, use Transport Expdt Data
Sequence Space	--->	If 1 Bytes - Normal Else Extended
		Class: 4

The Source Reference carried by an ISO CR TPDU is meant to be a unique identifier selected by the transport entity initiating the connection to identify the requested transport connection. The Local Reference field of the VTPDU is mapped to the Source Reference field of the TPDU.

ISO TP4 allows only 32 bytes of data in connection request. If the Connection Request VTPDU carries more than that amount, the data are rejected. This model does not pose a problem, as the sending TCP will retransmit the data if it is not suitably acknowledged. Also TCP clients do not attach any special significance to data in the SYN_REQ (if at all they use it).

After building a CR TPDU, a checksum is computed and CR is handed down to the IP component for eventual delivery to the transport end point (the addressing issues were discussed in the previous chapter).

The ISO TP client responds with a connection response which causes the TP entity to generate a Connection Confirm (CC) TPDU. The CC TPDU is similar to the CR TPDU in most respects. In the CC TPDU, the Destination Reference field now carries the reference ID that was received in the CR. The Source Reference is selected by the transport entity initiating the CC TPDU.

These two fields are mapped to the VTPDU as follows :

Destination Reference	-->	Remote Reference
Source Reference	-->	Local Reference

The Remote Reference in now the ISN that was generated by the TCP entity that did the active open.

The rest of the parameters are analyzed and mapped into the connection confirm VTPDU in a fashion similar to the mapping of the CR VTPDU to ISO CR TPDU. When the CC VTPDU is received by the gateway, it is to be mapped back to a SYN_ACK. The Acknowledgement Number is mapped from the value that is carried in the Remote Reference field of the connection confirm VTPDU.

The TCP entity's reply to a valid SYN_ACK in the SYN_SENT state is to generate an ACK to complete the three way handshake. The ACK carries the next expected sequence number. The gateway generates a ACK VTPDU, which also conveys the next expected sequence number. In the virtual transport domain, the sequence number (referred to as Virtual Sequence Space) carries both an octet binding and a packet binding. The Virtual Sequence

Space concept is described in greater detail in the Chapter 5 when the data transfer phase is discussed.

The specification of the VTPDUS involved in the connection establishment and the mapping rules for the gateways involved are presented in the following sections.

4.4.1 Transport Convergence Function Design

The preceding discussion informally described the process by which a connection is established between transport end points using the Virtual Transport architecture. The protocol specific TPDUs are mapped to a common format VTPDUs by the TCFs. Design guidelines for the TCFs in the gateways can be stated as follows:

(a) The TCF is aware of the nature of the adjoining subnet of its gateway only and makes no assumption about the remote end transport protocol.

(b) The VTPDU generated by the TCFs are consistent. This implies that a connection request VTPDU generated from an ISO CR should have the same syntax and semantics as that generated from a TCP SYN segment.

(c) The TCF attempts to maintain a minimal (if any) state information. This propagates from the decision to make the end systems provide reliability.

4.4.1.1 Mapping Initial Sequence Numbers Both TCP and ISO transports generate a unique sequence number when birthing a new connection. This is known as the Initial Sequence Number (ISN) for TCP and a Source/Destination Reference ID for ISO TPs. In the ISO transport model, these reference IDs remain fixed (and unique) during the lifetime of the connection. A separate sequence space is used during data transfer. TCP on the other hand uses the ISN as a starting point for a sequence space during the lifetime of the connection.

The Virtual Transport uses fields termed *Local Reference* and *Remote Reference* to carry the information that is conveyed by the ISN, ACK number, SEQ number and the Reference IDs during connection establishment. As was mentioned in the previous section, the sequence number in the VTL domain will carry an octet and packet count binding. Shown below is a scenario where a TCP sends a SYN segment with an ISN=100. It expects a SYN request from its peer, accompanied with an acknowledgement number of 101, positively acknowledging the initiator's SYN. To complete the three way handshake, the initiator sends an ACK segment with an acknowledgement number 2021 and a sequence number 101. TCP ACK segments carry the value of the next sequence number expected, and the sequence number that they carry indicate the number of the first octet in the segment. The corresponding next expected packet number (Ns) field of the ISO ACK packet is 0.

SYN (100)	---> (CR VTPDU) --->	CR (SrcRef=100)
SYN (2020), ACK(101)	<--- (CC VTPDU) <---	CC (SrcRef=2020, DstRef=100)
ACK (2021) SEQ(101)	---> (ACK VTPDU) --->	ACK(DstRef=2020, Ns=0)

ISO Transports expect a unique source reference in CR TPDU. The responder then uses that as the destination reference in the CC TPDU, while generating a unique source

reference for itself. Thus the responders end of the connection is identified by the reference ID 2020 and the initiators end point is identified by the reference ID 100 throughout the lifetime of the connection.

A one to one mapping of :

ISN in TCP SYN -> Local Reference in CR VTPDU -> Src_ref in ISO CR

is acceptable, and so is

Src_Ref in ISO CC -> Local Reference in CC VTPDU -> ISN in TCP SYN_ACK

but Dst_Ref in ISO CC -> Remote Reference in CC VTPDU -> ACK Num in TCP SYN_ACK

is *not valid* because TCP accepts an ACK as acceptable only if the ACK Number is ISN+1 (for completion of three way handshake).

It would be a simple matter to generate the ACK Number by incrementing the value of the remote reference if the TCF knew that this VTPDU was generated due to an ISO CC. This approach is unacceptable, as it forces the TCFs to be aware of the nature of the remote transport. Traditional protocol converters employ such a technique. Instead the TCFs make decisions based on the knowledge of the transport protocol being employed in the attached subnet.

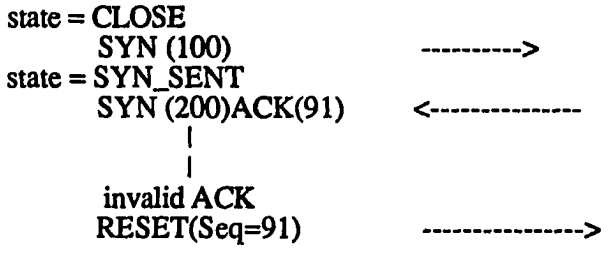
To provide for a consistent interpretation for the local reference and remote reference fields of the VTPDU, they will be kept constant during the connection establishment phase.

4.4.1.2 Handling of Error Conditions The TCFs play a passive role while doing the translation to a VTPDU. They do not attempt to ascertain the validity of the TPDU received from the local transport. This is done at the destination end systems. Thus

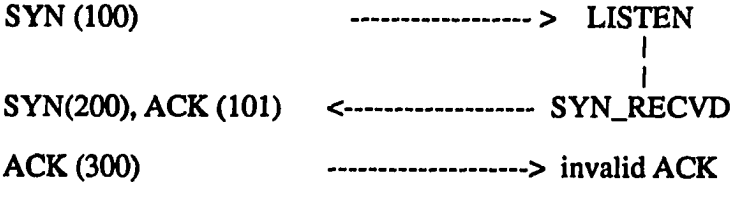
delayed/duplicated packets are not detected by the TCFs, but are mapped to the equivalent VTPDUs. The mapping functions should be such that the end transport entity can recognize the delayed/duplicated packets and take the appropriate actions. Any Local TPDUs that the TCFs cannot recognize or those that fail the checksum are discarded. The following discussion describes how error conditions detected by TCP and TP4 are conveyed transparently by the VTL.

(a) **Reset Generation and Processing by TCP:** During the connection establishment phase, TCP detects and generates RESETS as follows:

1. In SYN_SENT state: The TCP entity is now expecting an ACK for the SYN it transmitted. If it gets an invalid ACK, a RESET is generated as shown below.



2. In SYN_RECVD state: The TCP entity is now expecting an ACK to complete the three way handshake. If an invalid ACK is received, a RESET is generated as follows:



```
                                                    |
                                                    |
RESET (Seq = 300)        <---------------------- Send Reset
```

3. In ESTABLISHED state: Detection of illegal sequence of events during the data transfer phase also result in RESET generation. This case will be discussed when the connection termination phase of the VTL is specified.

(b) **Disconnects from ISO TP4:** ISO TP4 generates disconnect requests when an invalid condition is detected. The rules associated with are as follows :

1. State CLOSED: If an unacceptable CR TPDU is received, a DR is sent with source reference set to zero.

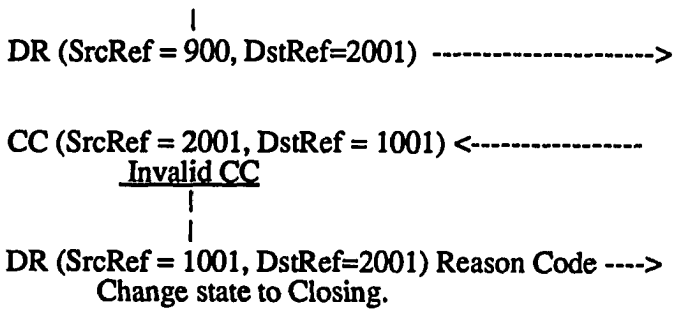
```
CR (SrcRef = 1001, DstRef = 0)  <-----------------------
        If invalid CR
                |
                |
DR (SrcRef = 0, DstRef=1001) Reason Code
```

2. State WFCC: While in waiting for connection confirm state, if a CC is received which cannot be associated with any existing transport end point[2], a DR is transmitted. A DR is also transmitted if an association is made but the CC is invalid for some other reason. In this case, a Disconnect Indication is given to the Client.

```
CR (SrcRef = 1001, DstRef = 0)  ----------------------->

CC (SrcRef = 2001, DstRef = 900) <-----------------
    CC cannot be associated
                |
```

[2] This association is performed by attempting to match the three tuple <DST_REF in the received TPDU, Source NSAP, Destination NSAP > with those of existing transport connections.



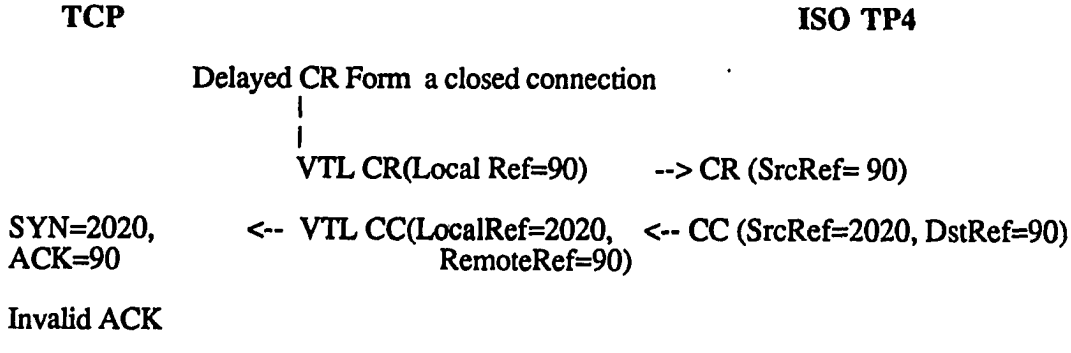
3. State AKWAIT and OPEN: A DR is sent when the retransmission count exceeds a maximum value.

For both TCP and ISO TP4, the RESET or the DR should have valid sequence number/reference IDs so that the receiving entities can recognize and act on them. In both the transports, a connection is rejected during connection establishment if:

1. An unsolicited SYN_ACK/CC is received.
2. An Invalid ACK in TCP; or
3. An Invalid CR in ISO TP4 is received.

Some of the scenarios that can be envisioned are as follows:

1. Delayed CR received by TP4



RST (Seq=90) -- > VTL DR (LocalRef=90, RemoteRef=2020) --> DR (SrcRef=90, DstRef=2020)

2. Delayed SYN received by TCP

TCP		ISO TP4
	Delayed CR form a closed connection	
SYN 90	<-- VTL CR(Local Ref=90)	
SYN=2020, ACK=90	--> VTL CC(LocalRef=2020, RemoteRef=90)	--> CC (SrcRef=2020, DstRef=90)
		Unsolicited CC send DR
RST (Seq=90)	<-- VTL DR (LocalRef=90, RemoteRef=2020)	<-- DR (SrcRef=90, DstRef=2020)

3. Detection of Duplicate CR: Duplicate detection is handled by the end systems. The VTL merely converts the sequence numbers to the local format.

4.4.1.3 Usage of Checksum by the VTL VTPDUs generated by the TCFs are covered by a checksum even if one of the end transports requires it and if the configuration allows so. The usage of checksum often results in performance degradation, so it should be possible to turn off checksum computations at the discretion of the system administration. The connection request and the connection confirm VTPDUs always carry a checksum. A VTPDU with a invalid checksum is rejected by the TCFs.

4.5 Rules for the TCFs During Connection Establishment

The previous section described the role played by the VTL and the TCFs during the connection establishment phase. Although the TCFs are designed specifically for a particular local transport, the mappings they perform are done with the aim of generating a Virtual Transport PDU whose syntax is not uniquely influenced by the nature of the local transport.

The set of rules that the TCFs must follows to generate VTPDUs during the connection establishment phase are listed below.

(a) The VTL uses Local and Remote Reference IDs which are mapped from the local transports representation of the ISN.

(b) The value of the Local and Remote references remains fixed during the connection establishment phase and are kept as state variables by the TCF instances.

(c) Credit allocations are normalized with respect to the VTL packet size.

(d) The maximum PDU size information that the VTPDUs may carry is the minimum of the PDU size supported by the local transport and the PDU size that the gateway can handle.

(e) A unique initial interaction, i.e., a TP4 CR or a VTL connection request causes the TCF state to change from CLOSE to OPEN. The uniqueness of the initial interaction is determined by comparing the calling and the called transport addresses with those maintained by the active TCF instances. A unique end point identifier (end point ID) is

associated with the TCF. This end point ID is carried by all VTPDUs to facilitate associating the incoming VTPDU with the correct TCF instance.

(f) The Virtual Sequence Space has an octet binding as well as a packet count binding. This concept is detailed in the next chapter.

(g) Each TCF module instance is uniquely identified by the calling and called transport end point address. This property is used by the IP module and the CLTS module to pass interactions to the correct TCF module. Figure 4.3 shows the association between IP module, the TCFs and CLTS modules. An end point identifier is generated from the transport end point addresses and this value is exposed to the peer TCF. After the connection establishment phase, all VTPDUs will carry this end point identifier instead of transport addresses. This end point ID is used to index the correct instance of the TCF. The actual transport addresses are then accessible.

(h) A TCF association is established when a unique initial interaction, i.e. a connection request TPDU, is received either form the VTL domain or the local subnetwork.

(i) The connection request and the connection confirm VTPDUs are protected by a checksum. Subsequent VTPDUs carry a checksum if one or both the end transport entities requested it. When the TCFs generate a local TPDU, a checksum is computed only if the local target transport entity had indicated its use.

(j) A connection termination TPDU from TCP or TP4 is mapped to a disconnect request VTPDU. The details of the connection termination mechanism are given Chapter 6.

The formal specifications of the TCFs for TCP and ISO TP4 gateways and the formats of the VTPDUs used during connection establishment phase are described in the subsequent section. Figure 4.4 shows the scenario where ISO and TCP clients interoperate transparently with peer clients of either transport using the VTL during connection establishment.

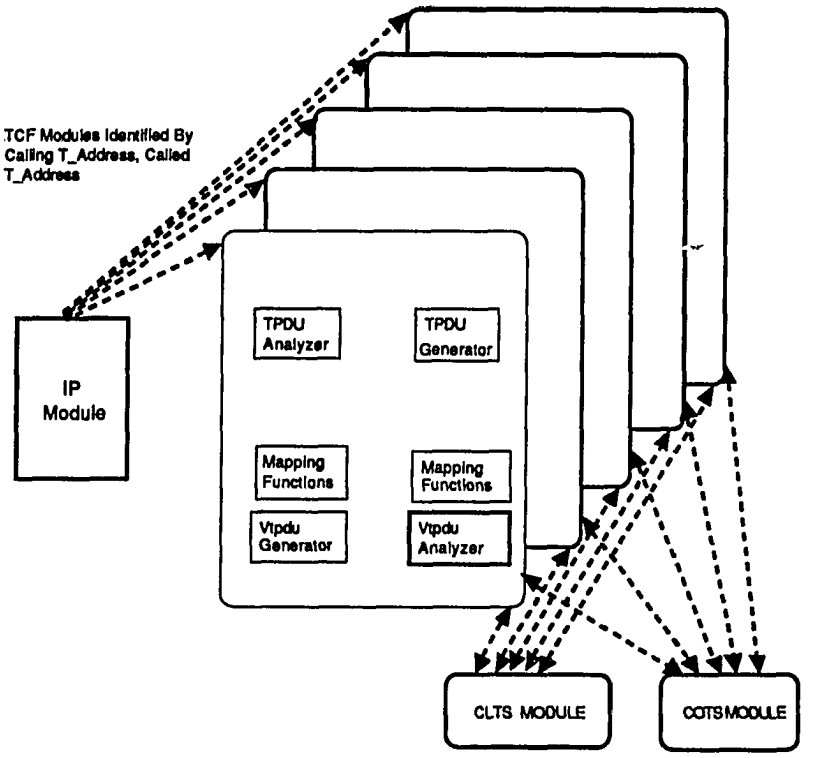


Figure 4.3 - Association of Modules

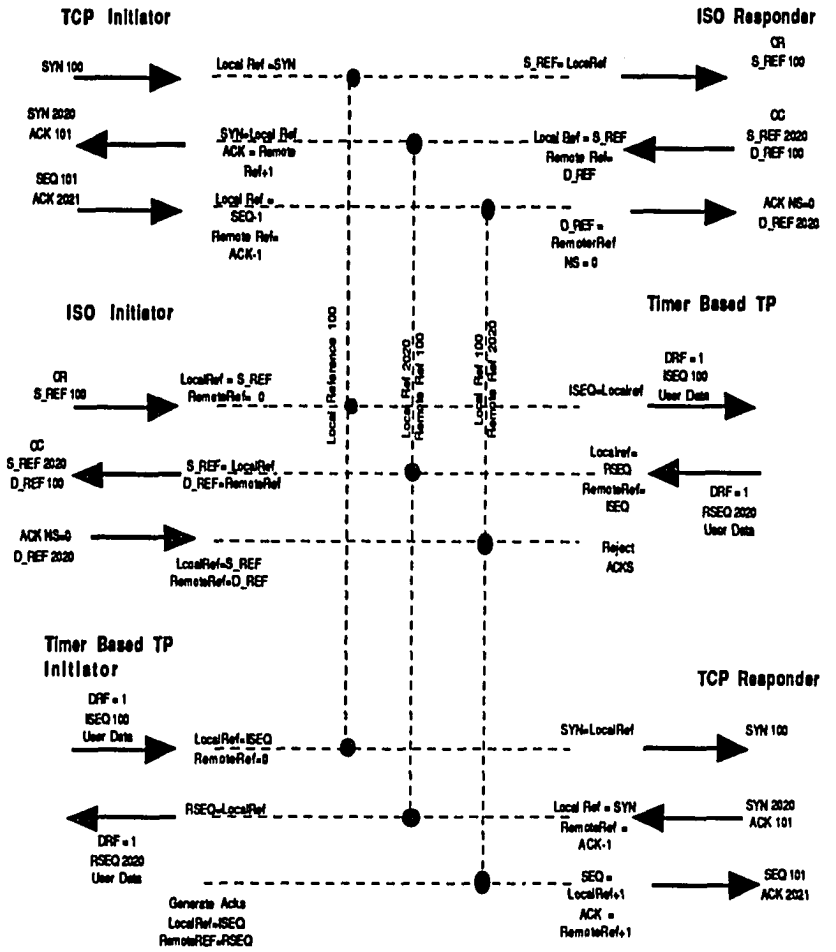


Figure 4.4 - Connection Establishment using the VTL

4.5.1 TCF Specification for TCP Gateways

The role of the TCF, for a TCP gateway, during connection establishment is formally specified in this section. Figure 4.5 depicts pictorially the modules that play a role in the specification.

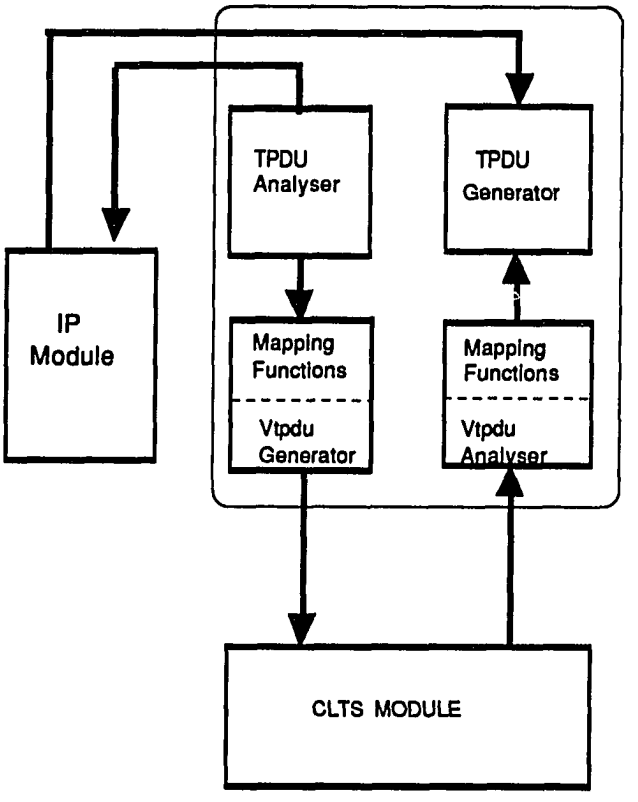


Figure 4.5 - Formal Specification of TCP_TCF

The TCP_TCF specification is done using the ESTELL FDT and is presented in Appendix A. The rules that are formally specified are as follows:

(A) **Connection Request:**

1. A connection request VTPDU is generated from a TCP SYN segment. The Local Reference field of the CR VTPDU assumes the values of the ISN as carried by the SYN segment. The sequence space size of 4 is used for the VTPDU.

2. The packet size as carried by the VTPDUS is given by:

   packet size = MIN (Max VTL Packet size, Max TCP Segment Size)

3. The credit allocation is computed from the TCP SYN request as:

$$\text{VtlCredit} = \frac{\text{TcpWindow in octets}}{\text{VTL Packet Size}}$$

4. A received VTL CR is mapped to a TCP SYN segment. The sequence number for the TCP segment is: ISN = Local_Ref field of the VTL CR.

5. From a received VTL CR the window size and the maximum segment size is computed as:

   TcpWindow in octets = VtlCredit * VTL packet size

   Max Segment Size = Max VTL packet size.

(B) Connection Confirm:

1. A TCP SYN_ACK is mapped to a VTL connection confirm (CC) VTPDU. The ISN value of the TCP segment is mapped to the Local Reference field of the VTPDU, and the Remote Reference field of the VTPDU take the value: ACK-1.

2. The credit allocation, and the maximum packet size is computed as in the case of connection request.

3. A VTL CC is mapped to a TCP SYN_ACK. The sequence number for the SYN_ACK equals the Local_Ref field of the VTPDU, and the acknowledgement number equals the value: Remote_Ref+1.

(C) Acknowledgements:

1. The VTL ACK carries both the next expected octet number and the next expected packet number. The next expected sequence number is the same as the TCP ACK number and the next expected packet number is 0. Acknowledgement generation is dealt with in greater detail in the next chapter.

(D) General:

1. A RESET segment is mapped to/from a disconnect request VTPDU. The detail of the connection termination mechanism is described in Chapter 6.

2. Reject any unknown segment from TCP or VTPDUs from the VTL.

Each instance of the TCF is uniquely identified by the calling transport address and the called transport address. Local variable that are maintained by a TCF instance are:

1. Local Reference derived from the ISN.

2. Remote Reference derived from the Local_Ref field of the VTL Connection Confirm VTPDU.

3. Calling and Called Transport Addresses for association purposes. This is referred to as the END_POINT_TYPE or the End Point Identifier in the specification.

4. The TCF changes state from CLOSE to OPEN when a unique SYN segment or a unique VTL CR VTPDU is received. The uniqueness is determined by comparing the called and calling addresses of the interaction with those maintained by the active TCFs.

5. The initial values of the next expected packet number is zero.

These rules for TCP TCFs are formally specified using the ESTELL FDT and are contained in Appendix A.

4.5.2 TCF Specification for ISO TP4 Gateways

The role of the TCF, for a TP4 gateway, during connection establishment is formally specified in this section. Figure 4.6 below depicts pictorially the modules that play a role in the specification.

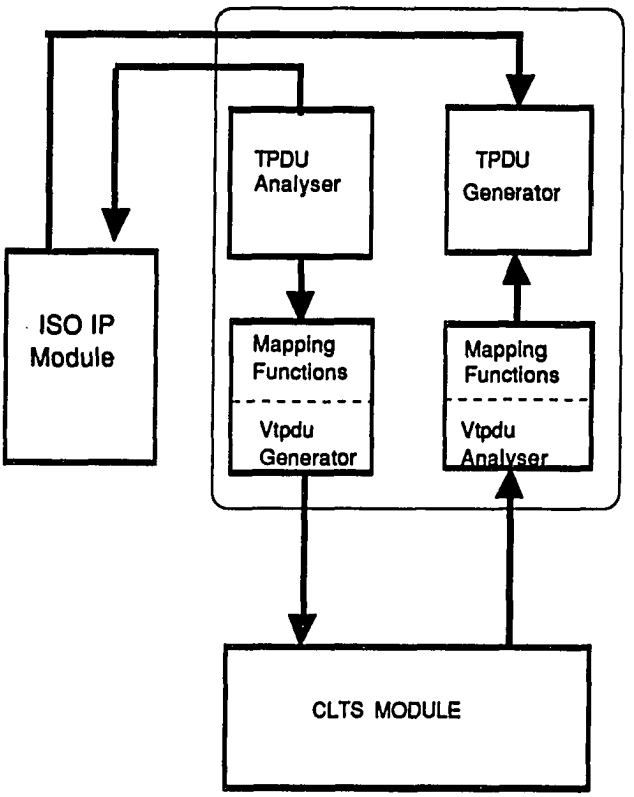


Figure 4.6 - Formal Specification of TP4 TCF

The TP4 TCF specification is done using the ESTELL FDT. The rules that are formally specified are:

(A) **Connection Request:**

1. A TP4 CR TPDU is mapped to a connection request VTPDU. The local reference field of the VTPDU takes the value of the source reference. The TP4 credit allocation is normalized to the VTL credit. VTL packet size is computed as MIN (Maximum VTL packet size, Maximum TPDU size).

2. A VTL Connection Request is mapped to a CR TPDU, with the source reference equal to the local reference filed. The TP4 credit is derived from the normalized VTL credit. The maximum TP4 TPDU length is the nearest valid packet size less than or equal to maximum packet size as carried by the VTPDU.

3. A TP4 CR or a VTL connection request causes a TCF state change from CLOSE to OPEN. An end point identifier is also associated with the TCF at time.

(B) **Connection Confirm**

1. A TP4 CC TPDU is mapped to a VTL Connection Confirm VTPDU. The source reference and the destination reference fields of the CC TPDU are mapped to the local reference and the remote reference field of the VTPDU respectively. The credit and maximum packet size are normalized as in the case of a CR TPDU.

2. A VTL connection confirm is mapped to the TP4 CC TPDU. The local reference and the remote reference fields of the connection confirm VTPDU are mapped to the source reference and the destination reference fields of the CC TPDU.

3. If the data in VTL connection establishment primitives are greater than 32 bytes, the data are discarded.

(C) **Acknowledgments:**

1. TP4 AKs are mapped to VTL acknowledgements. Since the VTL acknowledgements carry an octet binding as well as a packet count, the octet binding carries the values of Remote Reference + 1. The packet binding carries the value of the NS field in the AK TPDU. The next expected packet number of a received VTL acknowledgement is mapped to the NS filed of a TP4 AK TPDU.

(D) **General:**

1. Local variables stored are: LocalRef, RemoteRef, CallingTpAddress, CalledTpAddress, local end point identifier and remote end point identifier.

2. A DR TPDU is mapped to/from a disconnect request VTPDU. The details of the connection termination are described in Chapter 6.

The above mentioned rules for TP4 TCFs are formally specified using the ESTELL FDT and is presented in Appendix A.

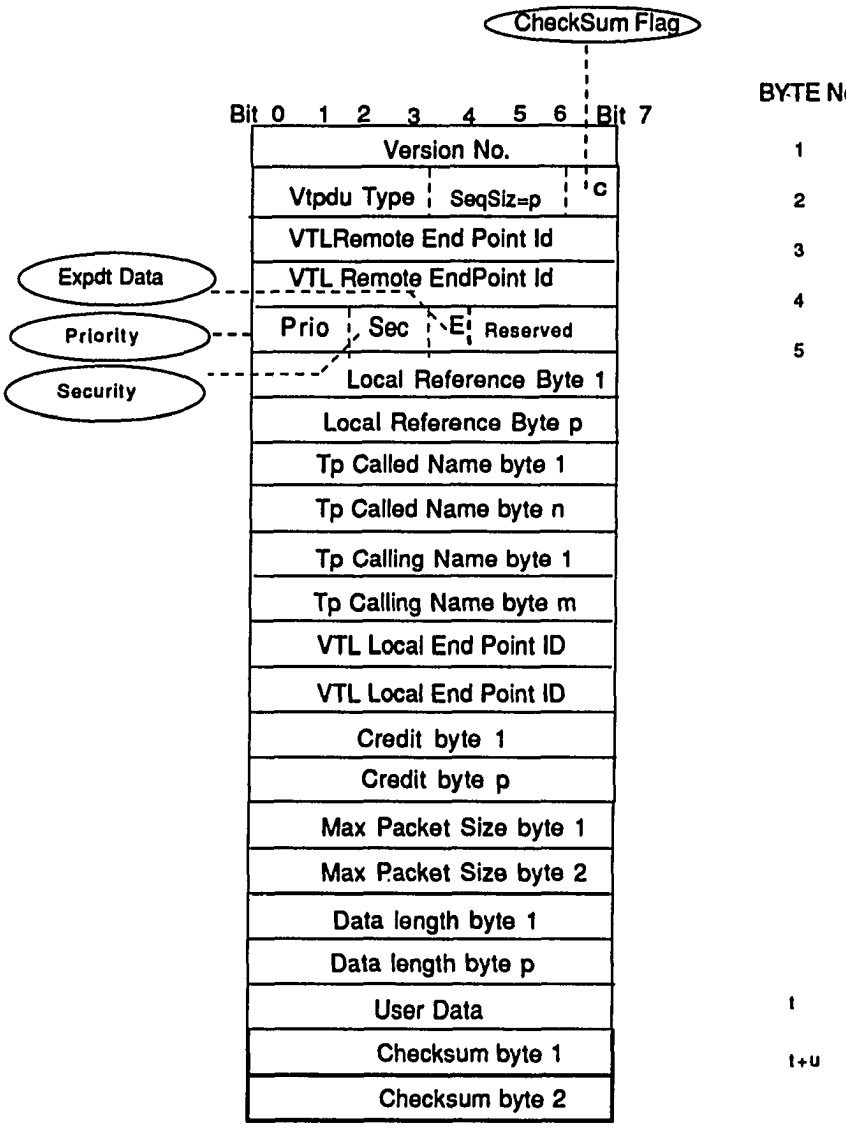


Figure 4.7 - Connection Request VTPDU

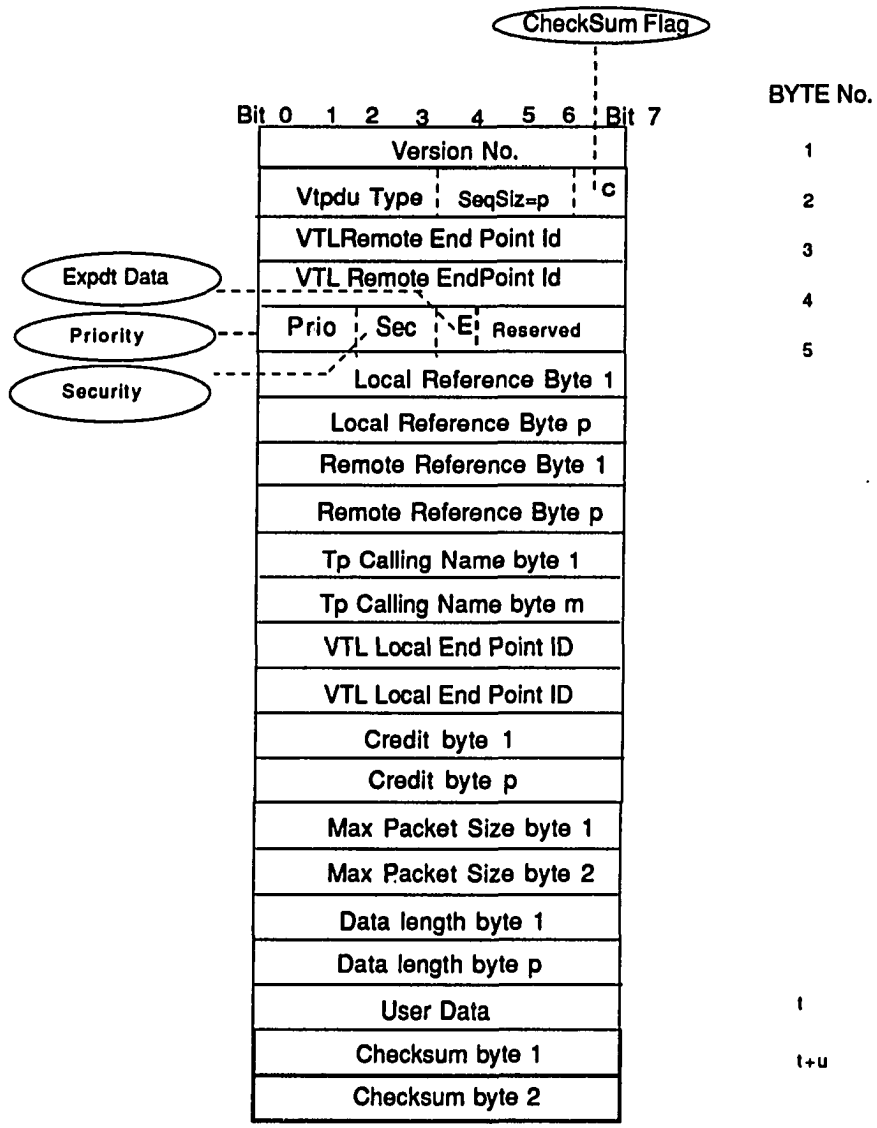


Figure 4.8 - Connection Confirm VTPDU

4.5.3 VTPDU Format

At this stage the following VTPDUs have been identified:

(a) Connection Request VTPDU

(b) Connection Confirm VTPDU

(c) Acknowledgement VTPDU

(d) Disconnect Request VTPDU

The acknowledgement and the disconnect request VTPDUs are described in Chapters 5 and 6 respectively, when the data transfer phase and the connection termination phase are formalized. Figures 4.7 and 4.8 show the connection request and the connection confirm VTPDU, respectively. The data structures are formally described in the ESTELLL specification presented in Appendix A.

Since the header overhead can affect performance, the need to optimize the headers is apparent and will be kept in perspective during the later phases of the design.

4.6 Conclusion

This chapter described the design of the VTL to support transport connection establishment. The role of the gateway components for DoD TCP and ISO TP4 were formalized. The next chapter describes the provision of data transfer by the VTL.

5 DATA TRANSFER PHASE

5.1 Introduction

This chapter describes the data transfer mechanism employed by the VTL to provide transparent interoperability between ISO TP4 and DoD TCP. It may be noted that the VTL technique is general and open ended, implying that interoperability is possible with other transport protocols as well. ISO TP4 and DoD TCP have been chosen to prove the concept primarily because of their popularity and installed base. Interoperability is achieved by translating the local TPDUs into a common format message object. This virtual TPDU is transmitted to the destination gateway using a CLTS. The conversion is done by a sub-module of the gateway, identified as the Transport Convergence Function (TCF), following a set of rules. The gateway architecture and the conversion rules for the connection established phase were formally specified in the previous chapters.

Section 5.2 and 5.3 discuss the methods employed by TCP and ISO TP4 to provide the data transfer service. Section 5.4 describes the design issues and the rule set for the TCFs to translate to and from the data VTPDU. Section 5.5 formally specifies the data transfer phase for TCFs of ISO TP4 and TCP gateways, respectively.

5.2 TCP Mechanisms to Provide Data Transport

A brief description of the some of the key features of the TCP data transfer mechanism that have implications for the interoperability solution are presented below. The distinguishing feature of the TCP data stream is that it is octet based.

5.2.1 Sequence and Acknowledgement Numbers

An outgoing TCP data segment carries a sequence number which binds the first data octet of the segment to the data stream. The sequence numbers are used by the receiving entity to generate ACK segments, update receive variables and windows, and to detect out of order and duplicate segments. The acknowledgement number carried in the segment conveys the sequence number of the next expected data octet. It confirms the reception of all data up to (but not including) the ACK number. Acknowledgements can be piggy-backed with data and are cumulative. Variations in the accept policy of the TCP entity can allow segments that straddle the receive window, i.e., a portion of the segment may be outside the window, to be accepted upto the octet which is in the receive window.

5.2.2 Sequencing of Acknowledgements

An ACK segment is accepted if the segment carrying it has a sequence number greater than the previous sequence numbers, or the acknowledgement number carried by the segment is higher than any previous acknowledgement number.

5.2.3 Push and Urgent Data

Data on a TCP connection are conceptually a stream of octets. The TCP entities can therefore buffer the data on both sending and receiving sides and transmit it at its convenience. Thus the sending ULP has no way of knowing if the data have been sent or is retained by the local or remote TCP entity while waiting for a more suitable segment or delivery size. The *push* mechanism is provided to the ULPs such that a TCP entity segments and sends all internally stored data within flow control limits and the receiving TCP must promptly deliver the pushed data to the receiving ULP.

TCP provides a means to communicate to a receiving ULP that some point in the data stream has been marked urgent. The urgent field in the segment is added to the sequence number to compute the last octet of the urgent data. The urgent pointer field is interpreted if the URG Flag is set. Note though that there is nothing which indicated the start of the urgent information.

5.3 TP4 Mechanisms to Provide Data Transport

Unlike TCP, ISO TP4 uses "data packets" as the boundary in the data stream. Thus, the sequence numbers track the number of data packets handled instead of number of bytes. There are separate TPDUs to carry data and acknowledgements, so piggy backing is not possible.

5.3.1 Sequence and Acknowledgement Numbers

A Transport entity allocates a sequence number of 0 to the first data TPDU that is transmitted. For subsequent data TPDUs, the sequence number is incremented by one. For normal format sequence numbers, modulo 2**7 arithmetic is used, and for extended format sequence numbers modulo 2**31 arithmetic is used. These sequence numbers are used by the receiving entity to generate ACKs, reorder TPDUs and detect duplicate packets.

An in-sequence data TPDU is acknowledged, within a certain time, by an AK TPDU which carries a sequence number of the *next expected* data TPDU. This is termed as *Next Receive* (and abbreviated as NR). Acknowledgements can be cumulative.

5.3.2 Sequencing of Acknowledgements

Since ACKs carry a credit value, interpreting the ACKs in a wrong order could be misleading to a transport entity. To prevent this, ACK TPDUs carry a subsequence number field. The usage of this field gives an order to the ACK TPDUs. Sequence control for transmission of ACK TPDUs is achieved as follows:

```
If (NR > NR used in previous ACK)
                    Sub_Seq No. = 0;

If (Credit >= Credit in previous ACK)
                    Sub_Seq No.(if used) = Previous Sub_Seq No.;

If ( NR = NR in previous ACK and Credit < Credit in previous ACK)
                    Sub_Seq No. = Sub_Seq No. in previous ACK + 1;
```

Correspondingly, a received ACK is defined to be in order as follows:

If (NR > NR in previous ACK)

OR

If ((NR = NR in previous ACK) AND
(Sub_Seq No. > Sub_Seq No. in any previous ACK))

OR

If ((NR = NR in previous ACK) AND
(Sub_Seq No. = Sub_Seq No. in any previous ACK) AND
(Credit > Credit in previous ACKs))

Thus, we see that a reduction is credit is signalled by sequencing the ACK that carried the information. The usefulness of this feature can be shown in the following example.

If a transport entity advertises a credit of 5 and subsequently reduces it to 3 for the *same* NR, then according to the rules specified above the later ACK carries a higher subsequence number. Now suppose that the ACKs get misordered and the ACK carrying credit 3 reaches the destination first, and then the ACK with credit 5. If sequencing of ACKs was not done, then the transport entity will use 5 as it send window. To avoid this, a check is made to determine if the ACK is in sequence. According to the rules above, this later ACK will be invalid as it carries a higher credit, but has a lower subsequence number.

Thus the use of subsequence numbers eliminates any confusion when a credit reduction is advertised by a transport entity.

5.3.3 Expedited Data

Expedited data (ED) use a separate sequence space than that used by the normal data. The receiving transport entity transmits an expedited acknowledgement (EA) TPDU with a sequence number equal to the sequence number of the received ED TPDU. The sender of the ED does not transmit any more data until it receives the corresponding EA. Also the semantics of the ED service dictates that an ED TPDU will reach the destination no later than any Data TPDU transmitted after the ED TPDU. Expedited data are limited to 16 bytes in size.

5.4 VTL Design Issues for Data Transfer Phase

The data transfer phase presents a set of issues pertaining to sequencing of data packets, piggybacking acknowledgments, sequencing of acknowledgments and handling of expedited data. Each of these is explained below.

5.4.1 The Virtual Sequence Space

As was seen in the case of ISO TP4 and DoD TCP, the sequence space was bound to the packet count and an octet count respectively. The representation of the virtual sequence number (VSN) should be such that it is possible for the TCFs to consistently generate the corresponding sequence number for the local transport entity. Thus the VSN should somehow convey both the octet count and the packet number of the data VTPDU. The virtual sequence space can follow any of the following bindings:

(a) Octet count based

(b) Packet number based

(c) Carry both packet number and octet count information

The choice of the nature of the virtual sequence space has ramifications on the design of the TCFs. A discussion of the TCF design issues depending on the nature of the virtual sequence space follows.

5.4.1.1 Octet Based VSN This would be the best suited for TCP TCFs, as it matches with the native sequence number representation. No extra mapping information would be needed by the TCFs. On the other hand, TP4 TCFs would need a cumulative byte count along with a history of recent packet number to octet count association. The mapping history needs to be maintained to handle transmissions.

5.4.1.2 Packet Count Based VSN This approach favors the TP4 TCFs as it is the native sequence number representation. For TCP TCFS however, to associate a packet number with a TCP segment, a mapping table has to be maintained between the sequence numbers carried by the segments and the corresponding packet number. The example below shows the need for maintaining a mapping table, and why generating a packet number dynamically will not suffice.

TCP Seq No.	VTL VSN.
100	0
200	1
300	2
200 --	

Due to a retransmission, VSN of 1 cannot be associated with this segment without maintaining a history of previous sequence numbers and corresponding VSNs.

Similarly while receiving data VTPDUs, the VSN has to converted to an octet based seq. number. Assume packets of 100 octets, RemoteRef = 99;

VTL VSN.	TCP Seq No.
0	RemoteRef +1 = 100 ; RecvCount = Data_Len = 100;
1	RemoteRef+1+RecvCount = 200; RecvCount=RecvCount + Data =200;
2	RemoteRef+1+RecvCount = 300; RecvCount = 300;
1 (retransmission)	Seq = 200; RecvCount = 300;
3	RemoteRef + 1 + RecvCount = 400; RecvCount = 400;

Thus we see that a table of recently used sequence numbers and packet numbers has to be maintained.

5.4.1.3 Packet and Octet Count Based VSN In this case, the VSN conveys both the octet count and the packet number. While doing so, the task of the receiving TCFs (for both TCP and TP4) is simplified as the required sequence number is carried by the

VTPDU. However, the transmitting TCF still has to maintain a mapping table to associate a sequence number with a TPDU.

An ideal VSN representation would be one which can be mapped to and from the target sequence space without a computational overhead, or the overhead of maintaining a mapping table. Such a representation, although highly desirable, does not seem possible. At this stage, an engineering decision is being made to use a packet and octet count based VSN. The overhead is then restricted to the transmitting TCFs. The receiving TCFs merely use what ever matches the local format.

Even with this approach there are problems. Since there is no guaranty that the gateways will receive packets in order. Thus, the TCFs may not be able to associate the correct VSN with a TPDU. This is explained later in the section on error recovery.

5.4.2 Acknowledgement Strategy

The issues of prime concern are piggybacked acknowledgements, cumulative acknowledgements and sequencing of acknowledgements.

5.4.2.1 Piggybacked vs Explicit Acknowledgements TCP uses piggybacked acknowledgements, implying that the data segments also carry an acknowledgement number. TP4 on the other hand uses explicit acknowledgements. If the VTL uses piggybacked acknowledgements, then the TP4 gateways have to dismantle the ACK VTPDU into a data TPDU (if there is any) and an acknowledgement TPDU. Lost TPDUs

would be compensated for by eventual retransmissions. On the other hand, if distinct ACK and data VTPDUs are employed, TCP gateways would have to break the TCP segments into ACK and data VTPDUs.

5.4.2.2 Cumulative Acknowledgements Both TCP and TP4 can accept cumulative acknowledgements. If a certain transport architecture requires explicit acknowledgements for every data TPDU, then a strategy has to be adopted by the VTL and the TCFs by which such acknowledgements are made available. This can be accomplished by breaking up cumulative acknowledgements at either the transmitting or the receiving TCFs. The implications of either decision are being left as area of further study, if and when interoperability such a transport architecture is required. For the scope of this research, cumulative acknowledgements do not raise an issue.

5.4.2.3 Sequencing of Acknowledgements As was explained in a previous section, TP4 uses a subsequence number with acknowledgements as a safeguard against interpreting out of sequence credit allocations. TCP does not have such a mechanism. If the ACK VTPDUs are to carry a sequence number, then there are two options on how to generate it. Either the TP4 sequence number can be used as such for TP4 gateways, while this parameter is ignored at the TCP gateways. The other option is to always have a sequence number for the ACK VTPDUs. In this case the sequence number is generated autonomously by the gateways. This approach may be flawed as there is no guarantee that the gateways themselves will receive acknowledgements in order. In view of this, the ACK VTPDUs will

mirror the TP4 ACK subsequence number value. A virtual sequence space can be chosen for the ACK VTPDUs, but at this stage TP4 subsequence number will be used as such.

5.4.2.4 Flow Control Confirmation Flow control confirmation (FCC) is a mechanism of the TP4 protocol whereby acknowledgement messages containing critical flow control information are confirmed using some optional fields of the TP4 AK TPDU. Critical acknowledgement messages are those that open a closed flow control window, or reduce credit. If this critical information is lost, then the resynchronization of the flow control relies on the expiry of the window timer which is generally of relatively long duration. In order to reduce delay in synchronizing the flow control, the receiving entity can repeatedly send, within short intervals, AK TPDUs carrying a request for confirmation of flow control state. This procedure is known as "Fast Retransmission of Acknowledgements". If the sender responds with an AK TPDU carrying an FCC parameter, fast retransmission is halted. If no AK TPDU carrying an FCC parameter is received, the fast retransmission halts after having reached a maximum number of retransmissions and the window timer resumes control of AK TPDUs. FCC is an optional mechanism and the data sender is not required to respond to a request for confirmation of flow control state. ACK VTPDUs will carry the FCC information in the TP4 format. The TCP gateways will ignore this parameter.

5.4.3 Expedited Data and Acknowledgements

TCP uses the urgent mechanism to signal the ULP that some information in the data stream has to handled in an expedited manner. An expedited data VTPDU can be used to convey urgent data. The virtual sequence space will again comprise of an octet binding and a packet number binding. The receiving TCP gateway generates a segment with the URG flag set, and derives its sequence number from the virtual sequence space.

5.4.3.1 Interoperability Issue with TP4 Expedited Data There is a fundamental difference in the manner in which TCP and TP4 offer this *emergency data* service. TP4 uses a different packet and acknowledgement stream for expedited data. Data transmission (both normal and expedited) are suspended until the expedited data packet is acknowledged. TCP on the other hand embeds the urgent data in the regular data stream, and does not expect any special acknowledgements. A foreseeable performance problem could be where a TP4 entity sends expedited data which is encoded as urgent data for the TCP entity. Since there is no way to solicit an immediate acknowledgement from a TCP entity, any further data transfer from TP4 would be blocked until such time as the TCP entity deems fit to send an ACK segment.

Further the TCP urgent mechanism does not mark any octet as the starting of the urgent information, instead the URG flag is marked true until the last octet of the urgent information is transmitted. Thus if a portion of the data stream is to be mapped into TP4 expedited data, it has to be packetized into 16 byte segments.

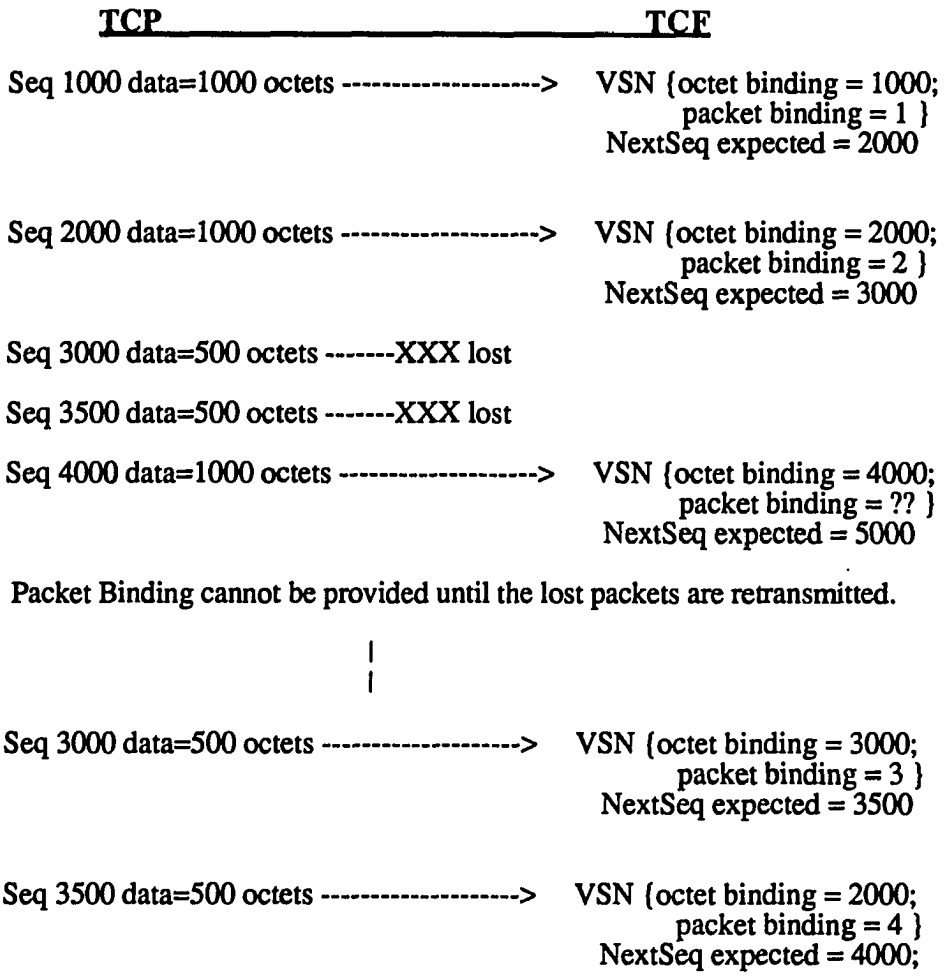
Theoretically interoperability between TCP urgent data and TP4 expedited data can be achieved, but in practice it is cumbersome. It may be favorable to restrict the usage of Urgent/Expedited data.

5.4.3.2 Expedited Acknowledgements TP4 uses an (EA) TPDU to acknowledge an ED TPDU. No sequence number information is carried by the EA TPDU as there can be only one outstanding unacknowledged ED TPDU at any time. TCP, however does not have a separate EA mechanism and expects acknowledgement for urgent data along with other acknowledgements. To accommodate the differing mechanisms, the VTL will use a VSN for expedited acknowledgement VTPDUs. This can be interpreted by gateways which need to generate an acknowledgment with a sequence number.

Another concern for the VTL is that TCP ACK segments may carry information regarding urgent data. The TCP specification states that in case a TCP's send window is closed and an urgent data request is made, then the TCP will send an empty ACK segment with the new urgent information. The VTL must make provisions in acknowledgement VTPDU to carry this optional information.

5.4.4 Use of COTS for Error Recovery

As was explained in the previous section, the TCFs may not be able to associate the correct VSN with a packet. The example below shows how this may happen.



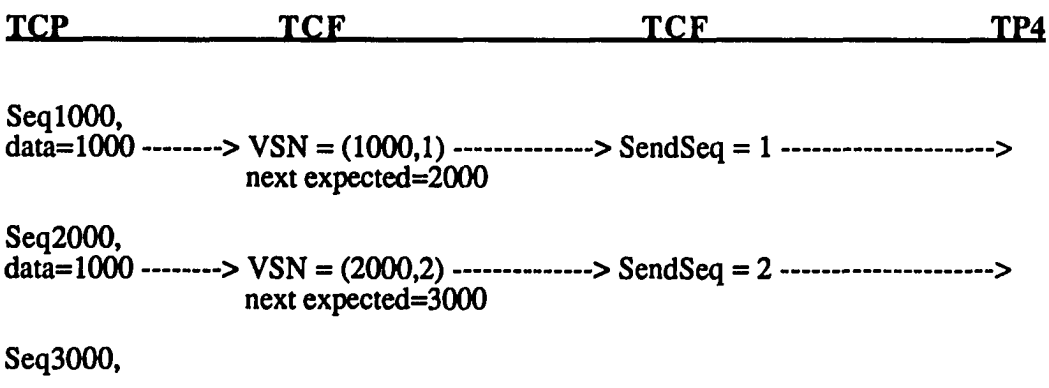
A similar scenario can be recreated for TP4 when it may not be possible to associate an octet binding for the packets in VTL.

To overcome this problem, the following procedure is recommended. Whenever a transmitting TCF loses synchronization of the binding, it shall continue to transmit VTPDUs using what ever the native binding is. This is done with the hope that the remote end point follows the local sequence space representation. This is continued until the transmitting TCF

receives the missing (retransmitted) TPDU and can fill the gaps in its binding-map table. On doing so the TCF conveys those values of VSNs that did not have all the binding information required in a message object, using the Connection Oriented Transport service (COTS), to its peer TCF module instance.

If the receiving TCF can use the partial VSN to generate the local sequence number, then it will do so. Otherwise, the following procedures can be used to recover from this temporary loss of VSN information. The simpler approach involves discarding the packets which do not have the required VSN information. Recovery would then be due to subsequent retransmissions. This approach will cause a performance degradation.

The alternative approach requires the receiving TCFs to buffer those received VTPDUs which do not have the required component of the VSN. When the transmitting TCF is able to fill in the missing values of the VSNs, it conveys the portion of the mapping table to its peer TCF using the COTS. The receiving TCF will then be able to complete the VSN information in the buffered VTPDUs, and hence generate the local sequence number. The example below explains how recovery from loss of VSN synchronization is done.



```
data=500 ----xxx Lost

Seq3500,
data=500 ----xxx Lost

Seq4000,
data=1000 --------> VSN = (4000,?) ----------> SendSeq = ? {buffer  Vsn=4000}
                    next expected=3000
                    Mark This Table entry.
Seq5000,
data=1000 --------> VSN = (5000,?) ----------> SendSeq = ? {buffer Vsn=5000}
                    next expected=3000

Seq6000,
data=1000 --------> VSN = (6000,?) ----------> SendSeq = ? {buffer Vsn=6000}
                    next expected=3000
Retransmissions
    |
    |
Seq3000,
data=500 --------> VSN = (3000,3) ----------> SendSeq = 3 ---------------------->
                    next expected=3500

Seq3500,
data=500 --------> VSN = (3500,4) ----------> SendSeq = 4 ---------------------- >
                    next expected=4000
                    matches with marked table entry.
                    next expected = 7000

          < VSN Map Table message :
                (4000, 5)
                (5000,6)
                (6000,7) >Send to peer TCF over COTS
                                  ---------->
                                                SendSeq = 5 ---------->
                                                SendSeq = 6 ---------->
                                                SendSeq = 7 ---------->

Seq7000,
data=1000 --------> VSN = (7000,8) ----------> SendSeq = 8 ---------------------->
                    next expected=8000
```

A similar recovery process can be visualized for the TP4 gateways. Thus, we see how the chosen architecture aids recovery from transient errors in a transparent manner, and how the connection oriented transport service (COTS) is used by the TCFs to exchange information.

5.5 VTL Specification for the Data Transfer Phase

5.5.1 Rules for the TCFs

Based on the above discussion, the following set of rules specify the data transfer phase of the VTL. The rules cover issues regarding sequence space for data and acknowledgement, provision for expedited data and recovery from loss of synchronization of the sequence space.

(a) A virtual sequence space is employed by the VTL, which provides an octet count based as well as a packet count based binding of the data stream.

(b) The data that an ACK VTPDU acknowledges is represented using the virtual sequence number. The virtual acknowledgement number has an octet count and a packet count binding. It refers to the next packet to be received and the next octet to be received. In case some transport entity has an acceptance mechanism by which data TPDU can be partially accepted and acknowledged, then the packet count component of the VSN will not be incremented. This implies that if the sending entity follows a packet count binding, then it will not receive acknowledgement for a partially received packet.

(c) Data VTPDUs carry optional acknowledgement/credit information. If the end transport entity uses piggybacked acknowledgements, then the TCFs should use this option instead of generating a separate ACK VTPDU. ACK VTPDU will be used if the transport entity uses separate ACK TPDUs.

(d) Acknowledgement VTPDUs carries an optional ACK sequence number which is used if the end system attempts to sequence its ACK TPDU.

(e) Acknowledgement VTPDUs carry optional Flow Control Confirmation information. This may be ignored by the TCFs if the local transport entity does not have a mechanism by which FCC can be provided.

(f) Expedited Data (ED) VTPDUs are used to carry any *emergency* data. The virtual sequence space for the ED VTPDUs carries an octet binding and packet count binding.

(g) Expedited Acknowledgements (EA) VTPDUs are used to acknowledge any ED. An optional field in the EA VTPDU carries any information pertinent to ED.

(h) Credit information is normalized in terms of the maximum virtual packet size that was negotiated during the connection establishment phase.

(i) In case the transmitting TCFs are unable to associate a complete VSN with a VTPDU, due to some inability to generate the derived component of the VSN, then the VTPDU is transmitted with whatever components of the VSN that can be generated.

(j) If a partial value of the VSN is sufficient for a receiving TCF to generate a local sequence number, then it shall do so. Otherwise it will follow any of the following procedures.

1. It can discard any VTPDUs which carry a partial value of the VSN. Recovery will then be due to subsequent retransmissions by the originating end system. This simplicity will be attained at the cost of performance.

2. The receiving TCF buffers the VTPDUs till such time as a mapping information message is received form the peer TCF. This should enable the TCF to associate a sequence number with the buffered VTPDUs. Alternatively the VTPDUs are buffered till such time as duplicate VTPDU with the complete VSN information is received.

(k) Whenever a transmitting TCF is unable to derive the complete VSN due to missing intermediate TPDUs, it shall maintain a table of incomplete VSN values. When the missing information in the incomplete VSN table can be filled, a mapping information message carrying a set of completed VSNs will be transmitted to the peer TCF.

(l) Service data unit boundaries, as indicated by use of the PUSH flag or an EOT flag, are conveyed by the EOSDU flag in the data VTPDU.

Based on these rules, the role of the TCFs during the data transfer phase for TCP and ISO TP4 will be specified in the next section.

5.5.2 Transport Convergence Function for TCP Gateways

As per the architecture definition, the TCF module of the gateways does the conversion between local TPDUs and the VTPDUs. To do so it must follow a set of rules, which are specified in this section for TCP gateways. The set of rules, as described below, can broadly be classified as pertaining to those dealing with the virtual sequence space for data and acknowledgements, generation of acknowledgements and urgent data.

A) **Rules for mapping to and from the virtual sequence space:**

1. Since the virtual sequence space consists of a byte count and a packet count, the TCF has to maintain a packet number to an octet count mapping. The mapping history has to be maintained for the transmitted data stream (to associate retransmissions with a virtual sequence number) and for the received data stream to associate subsequent acknowledgements from the local transport entity with a virtual sequence number.

2. The size of the mapping table containing octet count to packet number mapping is of concern. Mapping of the sequence numbers in the transmitted data stream needs to be maintained to associate the sequence number of a retransmitted segmented with the virtual sequence number (VSN) with which it was originally associated. Ideally the table entries should be dropped as acknowledgements are seen at the gateways, but as this does not guaranty that the end system will get the acknowledgements, the table may have to be maintained on a different criteria. One way of controlling the table size is to keep it in proportion to the send window of the transport entity associated with it by the TCF module. Thus a history of the last (2*Max_Send_Window) number of packets is

maintained. The Max_Send Window variable is maintained by the TCF module, and is updated when a credit allocation greater than any previous credit value is seen. Some maximum limit can be imposed.

3. A table of VSNs of the received Data VTPDUs needs to be maintained so that a VSN can be associated with the subsequent acknowledgements from the receiving TCP entity. The table size is limited to twice the Max_Recv_Window.

4. If the TCF is unable to associate the packet count component of the VSN due to loss of previous segments, it shall store that and subsequent the TCP sequence number in a separate table. VTPDUs are transmitted with a partial value of the VSN. As retransmission from the TCP entity allow for filling in the missing components of the VSNs, a message containing the mapping information will be transmitted to the peer TCF using the COTS interface.

5. If the TCF receives a VTPDU with a missing Octet Count component of the VSN it can take one of the following actions:

   (a) It can reject any VTPDUs that do not have the complete VSN. This approach simplifies the TCF design, but at cost of performance.
   (b) It should buffer the VTPDUs till it receives a mapping information message from its peer TCF, or a duplicate VPTDU with the complete VSN.

B) Rules for generation of Acknowledgements:

1. TCP segments carrying piggybacked acknowledgements will cause the TCFs to use the piggy backed acknowledgement options of the Data VTPDU. Empty ACK segments will cause the TCF to generate an ACK VTPDU.

2. TCP TCFs will ignore any ACK sequence information and Flow Control Confirmation requests carried by the ACK VTPDUs.

3. A TCP entity may partially accept a segment if it straddles the receive window. In this case the acknowledgement number value carried in the segment will reflect the octet up to which data were accepted. While generating the Packet binding for such acknowledgements, the Packet number used for the previous ACK VSN should be used. The example below explains the scenario:

```
        TCP                          TCF

SeqNum=1000 <---------- Data VTPDU
DataLen = 1000          VSN : (1000, 49)
                        DataLen =1000
                        { expected corresponding Ack = 2000 }
                        { next_expected Packet Num = 50 }

AckNum=2000 ----------> Ack VTPDU
                        VSN : (2000, 50)


SeqNum=2000 <---------- Data VTPDU
DataLen=1000            VSN : 2000, 50
                        DataLen = 1000

AckNum=2500 ----------> {segment partially accepted}
                        { expected AckNum was 3000}
                         Ack VTPDU
                         VSN : (2500, 50) { use previous value }
                                          { of packet number   }
```

C) Rules for handling Urgent data:

1. Urgent data is sent as an ED VTPDU. A separate virtual sequence space is employed for ED VTPDUs. The octet binding is embedded in the original octet stream, but a separate packet count is started. A flag in the mapping table indicates that the VSN is used to carry expedited data. The expedited data VTPDUs carry the total amount of expedited data and the amount being carried in the current VTPDU.

2. The urgent pointer information is derived from the received ED VTPDU. If the PUSH flag is set in the TCP segment, then the EOSDU flag is also set in the data or ED VTPDU.

3. Any TCP ACK segments that are recognized as acknowledging an ED VTPDU (through the VSN mapping table) are sent as EA VTPDUs. Any received EA VTPDUs are encoded as TCP ACK segments.

4. Any urgent pointer information carried by an empty ACK segment is encoded as an optional field in the ACK VTPDU.

5. The PUSH flag in the TCP segment is mapped to and from the EOSDU flag in the data VTPDU

Based on the set of rules for the TCP gateways, an ESTELL specification of the role of the TCF during the data transfer phase is presented in Appendix B.

5.5.3 Transport Convergence Function for TP4 Gateways

The set of rules that the TCF component of the TP4 gateways are again classified as those dealing with the virtual sequence space, acknowledgement generation and expedited data handling.

A) **Rules for mapping to and from the virtual sequence space:**

1. The TCF has to maintain a total byte count variable depicting the total number of bytes transmitted and associate it with the sequence number of the packets. This history is needed to map to and from the VSN.

2. A history of the last (2*Max_Send_Window) number of packets is maintained. The Max_Send_Window variable is maintained by the TCF module and is updated when a credit allocation greater than any previous credit value is seen. Some maximum limit can be imposed.

3. A table of VSNs of the received data VTPDUs needs to be maintained so that a VSN can be associated with the subsequent acknowledgements from the receiving TP4 entity. The table size is limited to twice the Max_Recv_Window.

4. If the TCF is unable to associate the octet count component of the VSN due to loss of previous packets, it shall store that and subsequent TP4 packet numbers in a separate table. VTPDUs are transmitted with a partial value of the VSN. As retransmission from the TP4 entity allow for filling in the missing components of the VSNs, a

message containing the mapping information will be transmitted to the peer TCF using the COTS interface.

5. If the TCF receives a VTPDU with a missing packet count component of the VSN it can take one of the following actions:

   (a) It can reject any VTPDUs that do not have the complete VSN. This approach simplifies the TCF design, but at cost of performance.

   (b) It should buffer the VTPDUs until it receives a mapping information message from its peer TCF or a duplicate VPTDU with the complete VSN.

B) **Rules for generation of Acknowledgements:**

1. TP4 ACK segments are mapped to ACK VTPDUs which carry a virtual acknowledgement number. Since the virtual acknowledgement number indicates the next packet number and the next octet number to be received, the TCF should maintain a table of the VSNs and the number of octets of the received data VTPDUs.

2. The subsequence number, if carried by the TP4 ACK, is repeated as such in the ACK VTPDU.

C) **Rules for handling urgent data:**

1. TP4 expedited data are sent as an ED VTPDU. A separate virtual sequence space is employed for ED VTPDUs. The octet binding is embedded in the original octet stream, but a separate packet count is started. A flag in the mapping table indicates that the

VSN is used to carry expedited data. The expedited data VTPDUs carry the total amount of expedited data and the amount being carried in the current VTPDU.

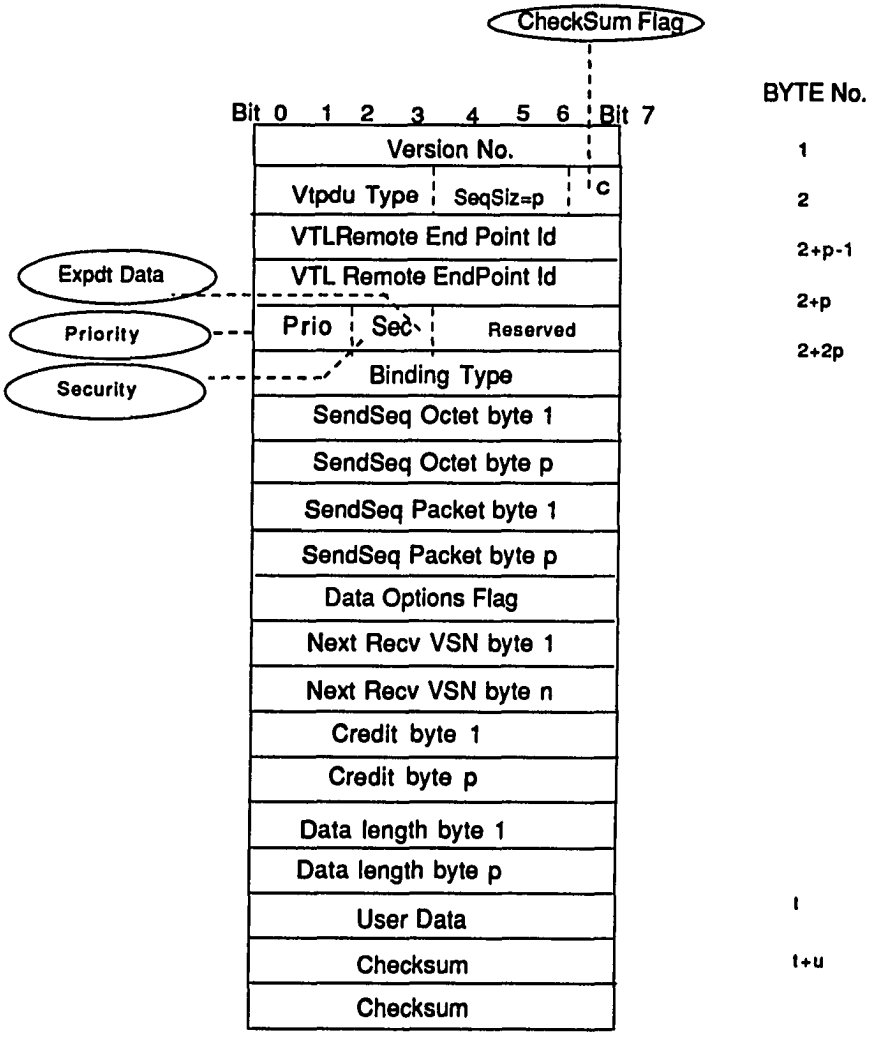


Figure 5.2 - Data VTPDU

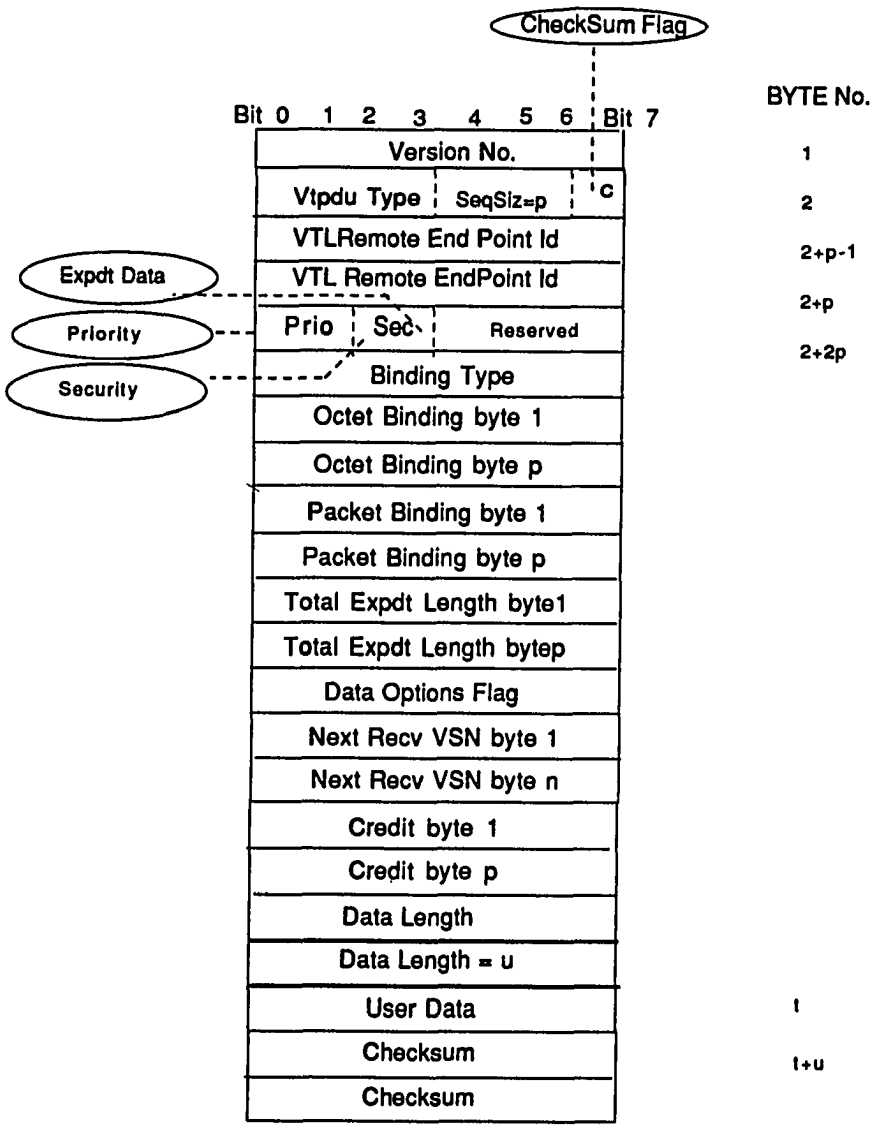


Figure 5.3 - Expedited Data VTPDU

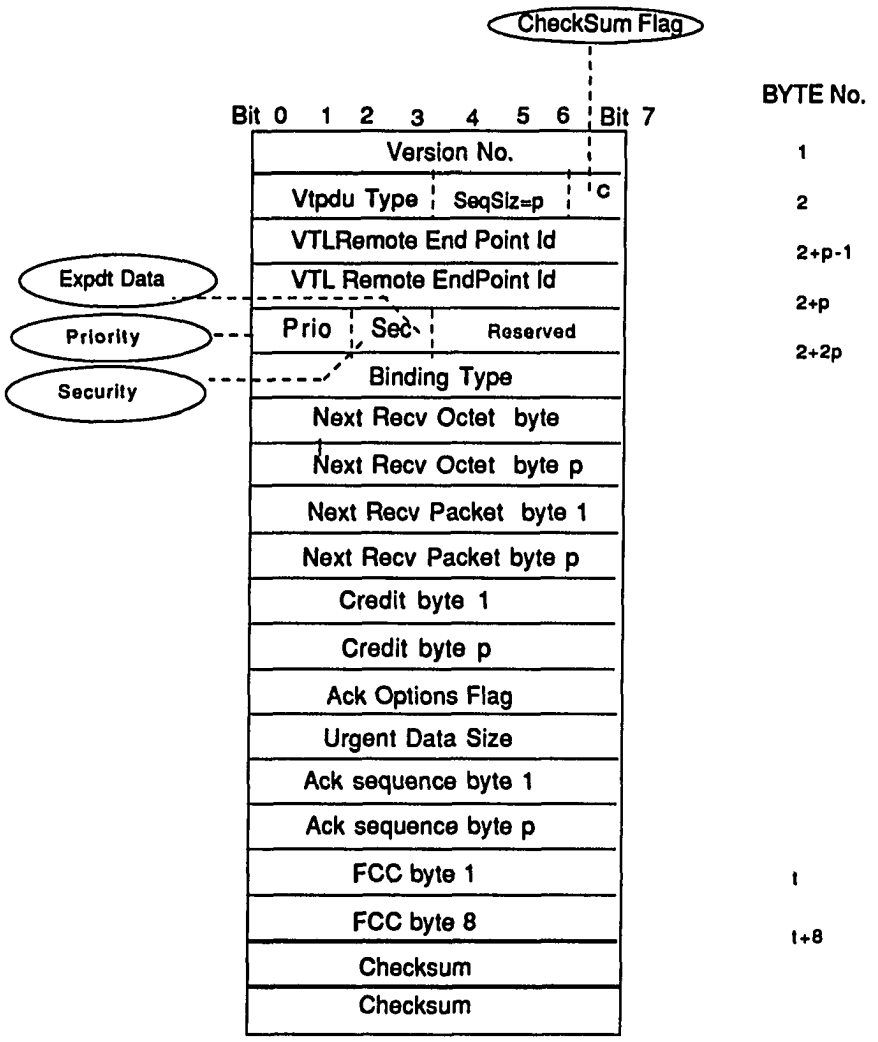


Figure 5.4 - Acknowledgement VTPDU

2. TP4 expedited acknowledgements are sent as expedited acknowledgement VTPDUs which carry the virtual acknowledgement number with a binding in the original octet stream and the separate emergency data packet stream.

3. Any EA VTPDUs received are mapped back to TP4 EA TPDUs.

4. The EOT flag in the TCP segment is mapped to and from the EOSDU flag in the data VTPDU.

Based on the set of rules for the TP4 gateways, an ESTELL specification of the role of the TCF during the data transfer phase is presented in Appendix B.

5.5.4 VTPDU Formats

The VTPDUs to accommodate the data transfer phase of the design are:

(a) Data VTPDU

(b) Expedited Data VTPDU

(c) Acknowledgement VTPDU

Figures 5.2 through 5.4 show the TPDU formats.

5.6 Conclusion

This chapter described how the virtual transport protocol provides transport interoperability during the data transfer phase. The usage of the connection oriented path between the gateways, as provided by the architecture, was demonstrated. Some problems with using TP4 expedited data when the peer transport is a TCP entity were exposed. The next chapter covers the connection termination phase.

6 CONNECTION TERMINATION PHASE

6.1 Introduction

This chapter describes the role of the VTL during the transport connection termination phase. The philosophy behind the VTL approach and the design of the gateway architecture was presented in the earlier chapters, and can be summarized as follows. Interoperability is achieved by translating the local TPDUs into a common format message object. This virtual TPDU is transmitted to the destination gateway using a CLTS. The conversion is done by a sub-module of the gateway, identified as the Transport Convergence Function (TCF), following a set of rules. The conversion rules for the connection established phase and the data transfer phase were formally specified in the previous chapters and in Appendix A and B using the ESTELL FDT.

Sections 6.2 and 6.3 discuss the mechanisms employed by TCP and ISO TP4 to provide the connection termination service. Section 6.4 describes the design issues and the rule set for the TCFs to translate to and from the disconnect phase VTPDU and formally specifies the role of the TCFs during the connection termination phase for ISO TP4 and TCP gateways respectively.

6.2 TCP Connection Termination

TCP provides for both "graceful" as well as "abrupt" closing of a connection. While providing graceful release, the TCP entity will transmit (and retransmit) any data in its internal queues till it is acknowledged. The connection is released only after a three way handshake of FIN segments is complete. This guaranties that the connection is released only after both the entities have finished transmitting any pending data. For non-graceful release, the RST segment is used. This is generated when an ABORT primitive is received from the ULP, or when an error condition is encountered. The initiating TCP transmits a RST segment and changes state to CLOSED. The receiving TCP does a Reset_Self procedure and also changes state to CLOSED. Some feature of the two connection termination procedures are described next.

6.2.1 Reset Generation and Processing

Resets are used to abruptly close established connections, refuse connection attempts and respond to segments not intended for the current incarnation of the connection. A Reset is validated differently depending on the state of the TCP entity. Table 6.1 shows how the RST segment is validated.

Resets are used both for abrupt closing of the connection on the ULPs ABORT request, and also when the TCP entity detects certain error conditions. A valid sequence number for the RST segment is necessary in order for the receiving TCP entity to accept it.

Table 6.1 - Reset Validation

CURRENT STATE	RST VALIDATION PROCEDURE
SYN-SENT	Ack Field should acknowledge the SYN. Change state to CLOSED
LISTEN	Ignore
SYN-RECVD	Seq_Num must be in the Recv_Window. If previous state was LISTEN, return to LISTEN, else CLOSED.
Any Other State	Seq_Num must be in Recv_Window. Abort connection, goto CLOSED state.

Correspondingly, a Reset is generated as shown in Table 6.2.

Table 6.2 - Reset Generation

CURRENT STATE	RST GENRATION PROCEDURE

CLOSED	RST is sent in response to any received segment.
	If TCP_Seg Carries an ACK then
	Seq_Num of Rst = Ack_Num of Recvd_Seg
	Else
	Seq_Num of Rst = 0 and
	Ack_Num = Seq_Num+Length of Recvd_Seg

LISTEN, SYN-SENT or SYN-RECVD	RST is sent if the received segment: 1. Carries unacceptable ACK. 2. Unacceptable Security. The Seq_Num for the Rst_Seg is:

Table 6.2 - (Cont.)

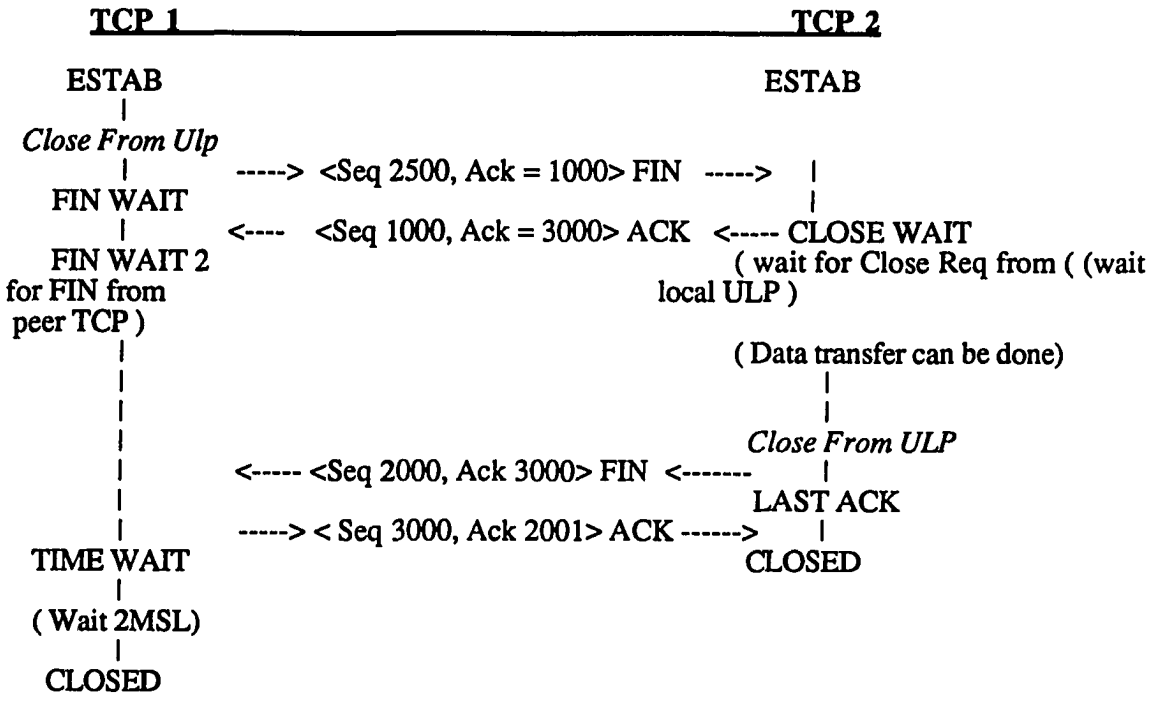
CURRENT STATE	RST GENRATION PROCEDURE

	If Recvd_Seg Carries an ACK then
	Seq_Num of Rst = Ack_Num of Recvd. seg.
	Else
	Seq_Num of Rst = 0 and
	Ack_Num = Seq_Num+Length of Recvd_Seg
	State does not change.

ESTAB, FIN-WAIT1, FIN-WAIT2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT	RST is sent if any unacceptable segment is received. Seq. Num of Rst = Ack Num of recvd. seg. State changes to CLOSED.

6.2.2 Graceful Connection Termination

A Close_Request from the ULP indicates that the user has completed its data transfer. The TCP entity reliably transmits any outstanding data before starting the connection termination procedure. This implies that it is acceptable to make several Send_Requests followed by a Close_Request and expect all the data to be sent to the destination ULP. A ULP should continue to accept data after sending a Close_Request. A three-way handshake of FIN segments is used to provide this graceful close as shown below.



As seen in the sequence of events above, ULP of TCP 1 gives a Close_Request which causes TCP 1 to generate a FIN segment after all previous data have been acknowledged. TCP 2 does not send a FIN segment till it receives a Close_Request from its ULP. Thus TCP 1 goes into the FIN WAIT2 state where it is waiting for a FIN from TCP2. Data can be received in this state. When ULP 2 gives a close request, TCP 2 sends a FIN segment. Now the only remaining interaction is an ACK from TCP1 to finish the three way handshake. The FIN segments may get encoded to carry data, ACKs, urgent and push information.

6.3 TP4 Connection Termination

Unlike TCP, ISO TP4 does not provide a graceful close. That service is provided by the Session Layer. A Disconnect Request from the user results in generation of a DR TPDU. Any data queued up in the transport entity are discarded once a user makes a Disconnect Request. This is equivalent to the ABORT service in TCP. Further if a TP4 entity receives a DR TPDU from a peer, it gives a Disconnect Indication to the User, replies to the peer TP4 with a Disconnect Confirm (DC) TPDU, and closes the connection. The user cannot stop the generation of the Disconnect Confirm. Thus during connection termination there is no three way handshake between the ISO transport entities and the transport service user. The DC TPDU is not acknowledged or retransmitted.

6.3.1 Generation and Acceptance of DR TPDU

To initiate a Connection Release, a transport entity does the following :

(1) If a CC has been sent or received, it shall :

(a) Send a DR TPDU with the correct SRC and DST Reference.

(b) Discard all subsequently received TPDUs other than a DC or a DR.

(c) Consider the connection released on receipt of a DR or a DC.

(2) If a CC has not been received in reply to a previously sent CR, then either send the DR with a zero DST REF or wait for a CC and then send the DR.

On reception of a DR TPDU the following action are taken:

(1) If a CR has been sent, and a CC has not been received then consider the connection to be refused.

(a) If a DR has been sent for the same connection, consider the connection closed.

(b) In any other state, send a DC TPDU and consider the connection closed.

6.3.2 Data in Disconnect Request

TP4 allows for up to 64 bytes of user data in the DR TPDU. There is no guaranty of delivery of data in disconnect request. NIST agreements do not recommend sending data with the disconnect request (DR). This prevents users from importing any significance to data in the DR TPDU. For purpose of this research, NIST compliant transport entities will be assumed.

6.3.3 Disconnect Reason

The DR TPDU carries a one byte disconnect reason code. The various codes are:

- Normal disconnect initiated by Session entity
- Remote Transport entity congestion at connect request time
- Connection negotiation failed
- Duplicate source reference detected for same pair of NSAPs
- Mismatched references

- Protocol error
- Reference overflow
- Connection request refused on this network connection
- Header or parameter length invalid
- Reason Not specified
- Congestion at TSAP
- Session entity not attached to TSAP
- Address unknown

Besides the mandatory Reason Code field, ISO transport also allows a variable length field to convey any other information related to the clearing of the connection.

6.4 VTL Design Issues for the Connection Termination Phase

One of the prime concerns for the connection termination phase is the usage model of the disconnect service as provided by the transport entities. This refers to usage of Graceful Release or Abrupt Release. The other issues of concern are the provision for a three way handshake during connection release, and the mechanism for disassociation of the TCFs and the transport end points. These issues are described below.

6.4.1 Usage Model of Transport Disconnect Service

As described in the preceding sections, TCP allows for both graceful as well as a non-graceful release of the transport connection, where as TP4 allows for only an abrupt connection termination. The abrupt connection termination implies that any data that may be queued in the transport entity is discarded when a Disconnect Request is seen. An external interoperability architecture cannot bridge this gap in the semantics of the disconnect service.

To attain transport interoperability, it then becomes necessary to write applications such that they provide the graceful release function on a higher level and restrict the semantics of the disconnect service to that of a non-graceful release.

The other issue regarding the usage model of the disconnect service is data in the Disconnect Request. Since TP4 does not guaranty the reliable delivery of such data, the users of the service should not import any significance with this data.

6.4.2 Provision for Graceful and Non-Graceful Release

Applications that could potentially operate on diverse transport architectures should use a non graceful transport disconnect. To support this, the VTL should provide a VTPDU that can convey the correct semantics of the disconnect. Since the VTL design philosophy is to provide for an open ended solution to transport interoperability, support for a three-way handshake during connection termination needs to be provided. It is the end users responsibility to use the right disconnect service.

Since TCP FIN segments can carry data along with PUSH or URG information, the VTL must provide support for such scenarios.

6.4.3 TCF Instance Disassociation

At some stage it would be required to recognize that a TCF module instance is no longer associated with an active connection, and the resources allocated with that TCF instance can be freed. This situation can be recognized in case the TCFs handle a unique terminal interaction such as a TP4 DC TPDU or a TCP RST segment. In case a three-way handshake variant of the connection termination service is being used, then the TCF needs to track the Disconnect_Request/Disconnect_Confirm/Acknowledgement sequence before it can be disassociated. This would require the TCF to maintain some state information during the connection termination phase. An alternative approach would be to do the disassociation after a period of inactivity on a particular connection. This approach is complicated by the fact that it may not be possible to statically come up with the optimal time-out value.

An Inactivity Timeout is required however to compensate for the case when one or both the transport end points experience an unusual termination. In this case the TCFs may never see a terminal interaction/sequence, and the disassociation would need to be done on the expiry of an Inactivity Timer.

6.5 VTL Specification for the Connection Termination Phase

Based on the various issues described above, the role of the VTL during the connection termination phase is specified below.

(a) The VTL connection termination VTPDU will carry a Virtual Sequence Number. This is required in case the the target transport entity expects the connection release TPDUs to be in the receive window.

(b) VTL provides a Reset Request and Reset Confirm VTPDU whose semantics are those of a non-graceful release.

(c) VTL also provides for a three way handshake during the connection termination phase. This is accomplished by using the Disconnect Request (DISCON_REQ), Disconnect Confirm (DISCON_CNF) and a subsequent Acknowledgement.

(d) The TCF disassociation should take place after a Reset Confirm VTPDU is received/generated, or when the inactivity timer expires.

(e) In case the three-way handshake variant of the connection release is being used, the the TCFs should go through the state transitions necessary to recognize the completion of the connection release three way handshake. Figure 6.1 shows the transitions that the TCF needs to go through before it can be disassociated.

(f) A disconnect reason code will be carried by the disconnect VTPDU. The TP4 disconnect reasons will be used for this purpose.

(g) The DISCON_REQ / DISCON_CNF VTPDUs can carry data, piggybacked acknowledgements, and options to indicate urgent data or pushed data.

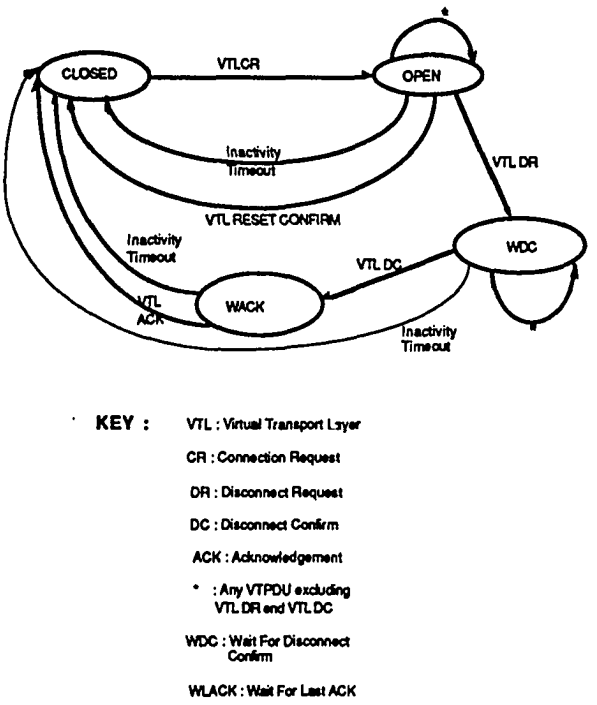


Figure 6.3 - TCF State Transitions

6.5.1 TCF State Transitions

Figure 6.3 shows the state transitions of a TCF instance. A TCF is associated with a connection when a unique initial interaction (a VTL Connection Request) is generated/received. A VTL connection request is generated when an end system request for a new connection is

seen by the TCFs. The end system transport entity guaranties the uniqueness of this request for connection by providing a "birth identifier", in form of an Initial Sequence Number or a Reference Number, in the TPDU. This is sufficient for the TCF to form an association with the requested connection. Once in the OPEN state, all TPDU/VTPDUs pertaining to the remainder of the connection establishment phase and the data transfer phase are translated.

If the "non-graceful" variant of the connection termination is employed, then the VTL Reset Confirm VTPDU takes the form of the unique terminal interaction for the TCFs. At this stage, the TCF can be disassociated from the connection. A disassociation can also be affected if the Inactivity Timer expires. This is a safeguard against one or both of the end systems terminating the connection prematurely.

Provision for a "graceful" connection termination warrants maintaining extra state information. This is so because, unlike the non-graceful release procedure, it is the Disconnect Request - Disconnect Confirm - Acknowledgement sequence which is unique, and the transition to the CLOSED state cannot be affected by any single VTPDU. The transition from OPEN to WAIT_DISCONNECT_CONFIRM takes place when a Disconnect Request VTPDU is received/generated. Data transfer can still take place in this state. A Disconnect Confirm VTPDU causes a state change to WAIT_LAST_ACK. This state signifies the last step in the completion of the three-way handshake. Any other TPDUs/VTPDUs are also handled in this state.

6.5.2 Transport Convergence Function for TCP Gateways

TCP allows for both the graceful and non-graceful termination of the connection. In case of non-graceful termination, a RST segment is transmitted and the connection is considered closed. Nothing is expected to acknowledge this RST segment. For graceful termination, FIN segments are exchanged and acknowledged by both the peers. It is assumed that the TCP application is responsible for using the correct variant of the disconnect service. The TCF does not have the intelligence or the information necessary to use a disconnect procedure different than the one dictated by the semantics of the TCP segment being used. The role of the TCF for TCP gateways during the connection termination phase is formally specified in this section. As shown in Figure 6.2, additional state information is required to support the three-way handshake during connection termination. A description of the states is given below.

WVTLDC: (Wait for VTL Disconnect Confirm) This state is reached from OPEN when a FIN is received from the local TCP entity. In this case the connection termination has been initiated by the local TCP entity.

WFIN: (Wait for FIN) This state is reached from OPEN when a Disconnect Request VTPDU is received and mapped into a FIN for the local TCP.

WLVTLACK: (Wait for Last VTL Ack) This state is reached when a FIN is received in the WFIN state and is mapped to a Disconnect Confirm VTPDU. From this state a transition to CLOSE is made when the corresponding VTL ACK is received.

WLACK: (Wait for Last ACK from TCP) This state is reached when a Disconnect Confirm VTPDU is received in the WVTLDC state. The Disconnect Confirm VTPDU is mapped to a FIN for the local TCP. The corresponding ACK from the TCP causes state change to CLOSE.

To support the graceful release variant of the disconnection complicates the design and subsequent verification of the TCF.

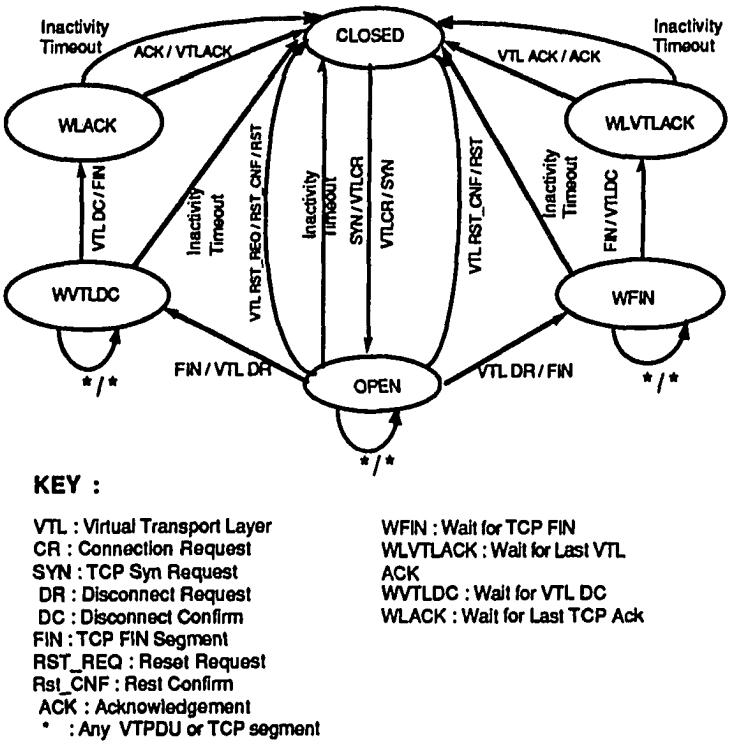
The design issues for the TCP TCF, as listed below, can be categorized as those related to handling RST segments, mapping to and from the virtual sequence number, and providing for the three-way handshake procedure.

(A) **Non-Graceful Release**

(1) A TCP segment carrying with the RST flag set is mapped to a RESET_REQUEST VTPDU. State change to CLOSED is affected by any of the following events.

(2) A RESET_CONFIRM VTPDU or an inactivity time out causes state change to CLOSED.

(3) A RESET_REQUEST VTPDU is mapped to a Reset TCP segment. Since the TCP does not respond to a RST segment, the TCFs generate the corresponding RESET_CONFIRM VTPDU and change state to CLOSED.



KEY :

VTL : Virtual Transport Layer
CR : Connection Request
SYN : TCP Syn Request
DR : Disconnect Request
DC : Disconnect Confirm
FIN : TCP FIN Segment
RST_REQ : Reset Request
Rst_CNF : Rest Confirm
ACK : Acknowledgement
* : Any VTPDU or TCP segment

WFIN : Wait for TCP FIN
WLVTLACK : Wait for Last VTL ACK
WVTLDC : Wait for VTL DC
WLACK : Wait for Last TCP Ack

Figure 6.2 - TCF State Transitions for TCP Gateways

(B) Graceful Release

(1) In the OPEN state, a DISCONNECT_REQUEST VTPDU causes a state change to WFIN. The DISCONNECT_REQUEST VTPDU is mapped to a FIN segment.

(2) FIN segment are handled according to the state of the TCF as described below.

(2.1) In the OPEN state, if a FIN segment is mapped to a DISCONNECT REQUEST VTPDU, and the state is changed to WVTLDC.

(2.2) In the WFIN state, a FIN segment is mapped to a DISCONNECT CONFIRM VTPDU, and state is changed to WLVTLACK

(3) In the WVTLDC state, a DISCONNECT_CONFIRM VTPDU causes a state change to WLACK. the DISCONNECT_CONFIRM VTPDU is mapped to a FIN segment. Any other VTPDUs and TCP segments are translated as per the rule specified in the Data transfer phase of the specification.

(4) In the WLACK, a TCP ACK corresponding to the FIN causes state change to CLOSE.

(5) In the WLVTLACK, a VTL ACK corresponding to a DISCONNECT CONFIRM causes state change to CLOSE.

(6) An Inactivity Timer is maintained, in all active states, whose expiry causes a state transition to the CLOSE state.

(7) The sequence and acknowledgement numbers carried by the TCP segments are mapped to corresponding virtual sequence numbers.

(8) Any disconnect reason code carried by the disconnect or the reset VTPDUs is discarded.

Based on the set of rules for the TCP gateways, an ESTELL specification of the role of the TCF during the data transfer phase is presented Appendix C.

6.5.3 Transport Convergence Function for TP4 Gateways

TP4 provides for non-graceful connection termination. The TP4 DR/DC TPDUs are mapped to RESET_REQUEST/REQUEST_CONFIRM VTPDUs. Since the disconnect VTPDUs carry a virtual sequence number and the corresponding TP4 TPDUs do not, the TCFs have to derive the VSN when generating the VTPDUs. The role of the TCF for TCP gateways during the connection termination phase is formally specified in this section. The design issues are listed below.

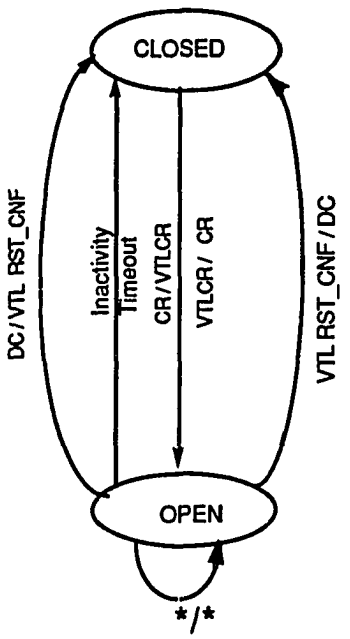
(a) A TP4 DR TPDU is mapped to a RESET_REQUEST VTPDU and a DC TPDU is mapped to a RESET_CONFIRMATION VTPDU. A state change is affected as follows:

(1) A TP4 DC TPDU causes state change to CLOSED.

(2) A VTL RESET_CONFIRM VTPDU causes state change to CLOSED.

(3) An inactivity time out causes state change to CLOSED.

(b) A DISCONNECT_REQUEST VTPDU, which is used to initiate the three-way handshake is mapped to a DR TPDU. The subsequent DC from the TP4 entity is



KEY :

VTL : Virtual Transport Layer
CR : Connection Request
CC : Connection Confirm
DR : Disconnect Request
DC : Disconnect Confirm
RST_REQ : Reset Request
Rst_CNF : Rest Confirm

Figure 6.3 - TCF State Transitions for TP4 Gateways

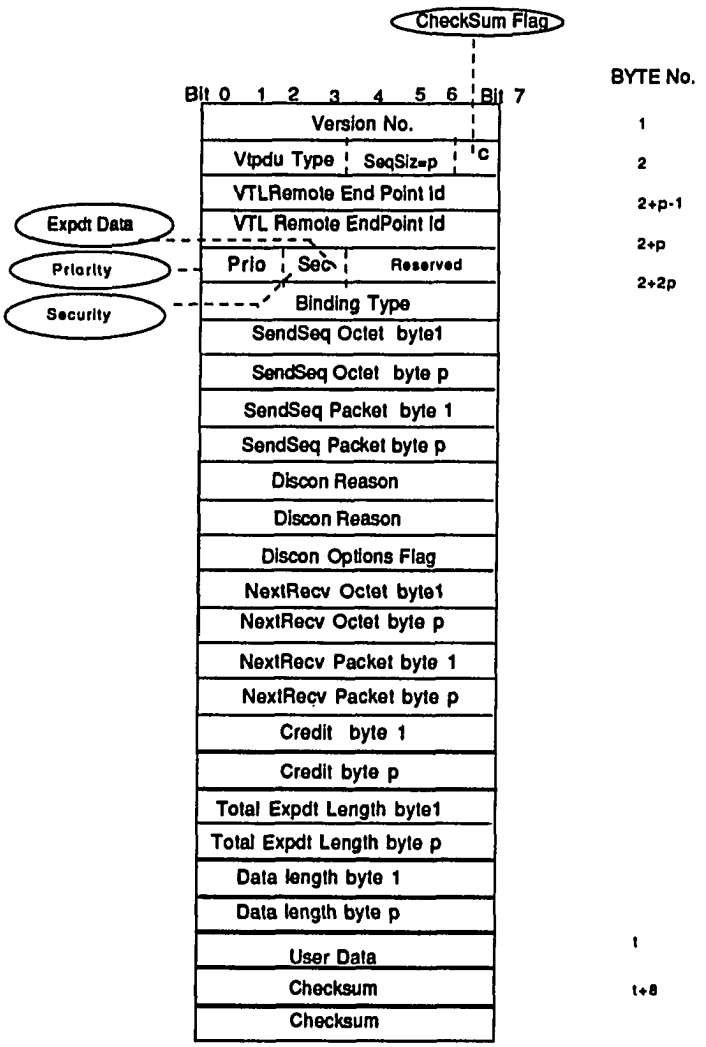


Figure 6.4 - Connection Termination VTPDU

mapped to a RESET_CONFIRM VTPDU and the state is changed to CLOSED. No attempt is made to participate in a three-way handshake during connection release.

(c) If the amount of data in the RESET_REQUEST/RESET_CONFIRM VTPDUs is greater than 64 bytes, the data are rejected.

(d) The TCFs must provide a virtual sequence number for the RESET VTPDUs. This needs to be derived from the mapping tables maintained by the TCF as the ISO TP DR/DC TPDUs do not carry a sequence number.

These rules are formally specified using the ESTELL FDT in Appendix C. Since a three-way handshake for the disconnection is not supported, no additional state information is needed. Figure 6.3 shows the TP4 TCF state transitions.

6.5.4 VTPDU Formats

The VTPDUs required for the connection termination phase are:

(1) Disconnect VTPDU. The VTPDU code identifies:

(a) Reset Request

(b) Reset Confirm

(c) Disconnect Request

(d) Disconnect Confirm

Figure 6.4 shows the format.

6.6 Conclusion

This chapter described how the virtual transport mechanisms provide transport interoperability during the connection termination phase. The usage model of the disconnect service for TCP clients has to be restricted to that of a non-graceful release in order to interoperate with a TP4 client. The VTL provides support for a three-way handshake during the connection termination phase at the cost of the TCFs having to maintain additional state information. The next chapter concludes the dissertation and compares the VTL approach with existing methods for transport interconnection.

7 CONCLUSIONS

7.1 Formal Methodology in Protocol Engineering

This chapter describes briefly the various stages that a protocol design effort goes through and the current status of the VTL design effort in that perspective. Figure 7.1 shows the various stages involved in the design of a Communication Protocol. Although an order has been associated with the cycle, but it is possible to conceive of other valid design sequences. Steps 1 through 4 can be considered as the core of the effort during whgich the new protocol is being formalized. Figure 7.2 provides a conceptual view of the grouping of the various steps of Figure 7.1. A brief description of the various stage is presented next.

7.1.1 Concept and Specification Phase

In general the design phase commences with the realization of the limitations of currently employed solutions, and how those limitations can be removed. A key concept is proposed, at this stage, describing the philosophy of the new approach and how it may overcome the experienced limitations of traditional approaches.

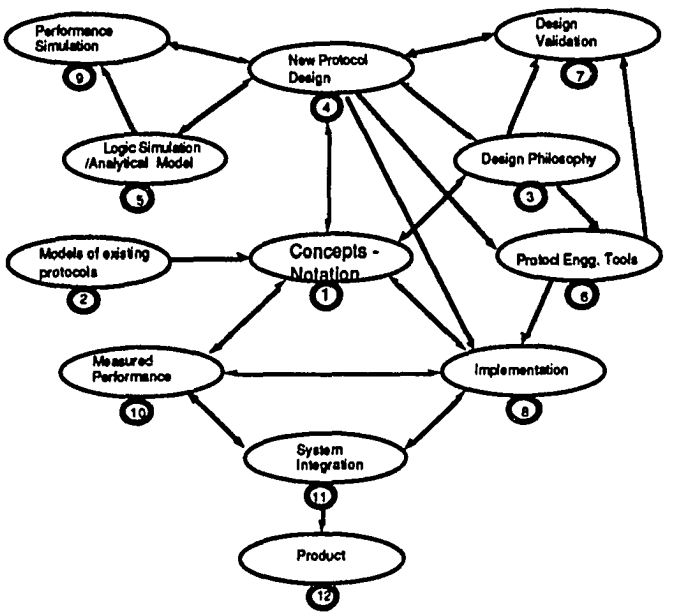


Figure 7.1 - Protocol Engineering Methodology

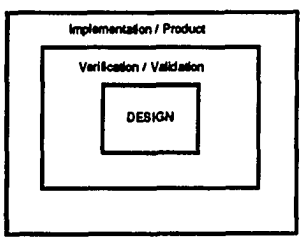


Figure 7.2 - Grouping of Protocol Engineering Tasks

This initial effort normally draws heavily from past experiences and detailed study of existing approaches. From this new concept, a textual (informal) description of the proposed protocol is derived. As the concept matures, a formal specification using a formal description technique (FDT) is produced.A formal specification is absolutely necessary as this communicates unambiguously the procedures and mechanisms of the protocol. The choice of the FDT and the availability of related tools can be critical for the subsequent stages of the protocol engineering effort.

An abstract system architecture that hosts the protocol is also presented at this stage. The envisioned modules and their interactions are identified. The finishing of this stage represents the first breakpoint in the process.

7.1.2 Verification, Validation, Simulation and Modeling Phase

After a formal specification of the protocol is available the next task is one of ascertaining the fact that the proposed approach is correct, complete, and to provide performance figures. Depending on the state set of the protocol, the input/output sequences for each state, the FDT used and the availability of related tools, the effort involved can be anywhere from a few man-weeks to a few man-years. For example trying to verify a specification done using state tables manually would be a far more time consuming task than if the specification were done using a FDT like ESTELL, LOTOUS or SDL on an automated protocol development environment [36].

At any rate, the errors exposed at this stage would require correcting the specification. Methods of protocol verification/validation is fertile research area by itself. For the protocol design effort it would be favorable to use existing tools.

If applicable simulation and/or analytical modeling is also done at this stage to get quantitative performance figures for the protocol. If bottlenecks can be identified they are exposed to the protocol designer. Again availability of computer aided tools can generate simulation models from the FDT.

7.1.3 Implementation and System Integration

This stage involves coding, either manually from the FDT or by an FDT compiler, and the hardware/software integration with the target platform. System dependent constructs like buffer/memory management, timers, access methods (APIs) to the protocol entity, etc., are resolved at this stage.

Looking at the TCP/IP protocol suit, the effort began almost a decade ago with some of the concepts being proposed in the mid 1970s. The DoD TCP specification was published in 1983. Research on TCP/IP protocols has continued till this day, with numerous contributions in the form of performance improvements, internetworking, and implementation techniques.

7.2 The VTL Design Effort in Retrospect

As per the protocol engineering cycle described above, the design of the VTL is guided by the need to provide interoperability at the transport layer in a manner which would overcome the limitations of current approaches. The philosophy behind the approach and an architecture to support the concept were formalized. An interoperability architecture, employing the VTL concept, for DoD TCP and ISO TP4 was formally specified using the ESTELL FDT. The salient feature of the VTL concept are as follows:

(a) Provides interoperability between transports that can operate on unreliable (Class C) networks.

(b) Connectionless in nature. The end systems provide the reliability; the VTL concept provides the transparency and interoperability regardless of the target transport entity as long as they satisfy the property in (a). In some cases a usage model for a particular end system transport service has to be defined in order to provide interoperability. This is necessary when the semantic gap between the service primitives of the end transports cannot be bridged by an external interconnection architecture.

(c) The VTL concept requires the transport convergence functions (TCFs) in the gateways to maintain only two states namely OPEN and CLOSE (when the non-graceful variant of the connection release is used).

(d) The TCFs have to deal with a finite set of well defined external interactions. One set of the interactions are in the form of TPDUs from the local transport entity. The other set is in the form of the VTL VTPDUs. The VTPDUs are also formally specified.

(e) The design was done such that complexity of the next phase, namely the validation/verification task, would be minimized. This manifests in the decision to operate the TCFs on a connectionless transport, thereby shielding it from the vagaries of the network layer management. A set VTPDUs were formalized so that the interactions in the VTL domain are limited and well known in syntax and semantics. The specification was done using the ESTELL FDT. With the availability of an automated development environment generation of the verification/validation test sequences should reduce to an engineering task.

7.2.1 Comparison with Protocol Converters

The approach is different than those taken by protocol converters. Protocol converters translate directly between the end system PDUs and thus need one such device for every architecture that interoperability is desired with. In the VTL the translation is done to a common message format. Now only one translating element is needed to allow interoperability with any other architecture.

7.2.2 Comparison with Service Bridges

Interoperability using Service Bridges requires the end systems to actually establish a connection to the bridge. As a result the end-to-end meaning of the transport service is lost. Thus acknowledgments, credit allocations, and flow control information that the transport

entity receives are from the service bridge. So if there is congestion in one of segments of the connection, that information (in terms of reduced credit) never reaches the end systems on the other segments. This is because there are no service primitives or parameters that convey such protocol specific information. Service bridges are also a performance bottleneck as they must maintain multiple transport connections with the end systems.

The VTL does not establish a connection between the gateways themselves or the gateways and the end systems. The information to be exchanged by the transport entities is conveyed end-to-end by mapping to a common representation of parameters. The connectionless nature of the VTL does not introduce the extra overhead that is associated with maintaining a connection.

7.2.3 Future Work

The scope of this research effort limits itself to the concept and specification phase. Future work can comprise of verification/validation and subsequent implementation of the VTL. The concept can be used to providing interoperability with other transports like those employed by DNA, XTP or the XNS protocol architectures. This would require designing TCFs for those transport protocols, and identifying a usage model of the transport services as dictated by the rules that the VTL prescribes for the various phases of a transport connection lifetime.

8 ACKNOWLEDGEMENT

I am thankful to my major professors Dr. Douglas Jacobson and Dr. James Davis, with whom I have been associated for the past four years as a graduate student at Iowa State University, for suggesting this challenging research topic. Over the years they have been instrumental in instilling in me the problem solving capabilities that are the mark of an engineer. Their technical excellence and foresight has served as a guiding light for my own goals and ambitions. I also extend my gratitude to my committee members: Dr. Arthur Pohm, Dr. Jonny Wong and Dr. Dave Martin who, at various stages of my student life at ISU, have contributed invaluably to my knowledge and experience. Thanks are due to Pam Myers who once again made sure that all my paper work was done in time.

I am grateful to my father, Er. Satinder Paul Singh Ahuja and my mother Mrs. Jaswinder Kaur Ahuja for giving my the opportunity, freedom and peace of mind to pursue higher studies. I am also grateful to my uncle, Dr. S. P. Singh, who has served as a role model for me.

Thanks to the wonderful people at Touch Communications Inc., Campbell, California, where I am currently employed, for their support and for providing an invigorating work environment. In particular I am grateful to Lori Logan who unselfishly sacrificed her

Macintosh so I could write my dissertation in the comfort of my office. I am further indebted to her for painstakingly applying her meticulous, caring and artistic disposition to correct my dissertation in content, form and style. But for her kindness, it would not have been possible for me to deposit my dissertation in time or in a presentable format. I owe an intellectual debt to Tony Vuong, Uppili Srinivasan, Ted Gauthier and Lori Logan for discussing with me the intricacies of communication protocols and architectures. Tapping into their collective knowledge, experience, pragmatic insight and their precious time expedited my research effort. A special thanks to Tony and Uppili for their patience, constant encouragement and mentorship both on and off the job. Thanks are also due to Randall Allsup who showed me how to operate Microsoft Word and Cynthia Jordan who graciously made it possible for me to return to Iowa in order to defend my Ph.D. dissertation.

I am grateful to the training, conditioning and the philosophies imparted to me by the fine martial art Tae Kwon Do (of which I am, and aim to remain, a student) and by my honorable teachers Master Yong Chin Pak (at ISU), Master Scott Coker and Master Garry Nakahama (in California). It was this conditioning that made it possible for me to go through the ordeal of a doctoral program.

Lastly I would like to thank dear friends and peers Vivek and Sonia Mehra, Shelly Coldiron, Don Carr, Dan Jhonson, William Denniger, Muhammad Shafiq, Mansoor Sarwar, Sunil Gaitonde and others for their support, encouragement and friendship.

9 BIBLIOGRAPHY

[1] Zimmerman, H. "OSI Reference Model of Architecture for Open Systems Interconnection." IEEE Transactions on Communications, Com-28, No. 4 (April 1980), 425-432.

[2] Network Working Group. "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods." *Request For Comment 1001,* (March 1987).

[3] MAP/TOP Users Group Technical Report. "Specification of NetBIOS Interface and Name Service by Lower Layer OSI Protocols." (September 1989).

[4] *C Language RPC Tool.* NETWISE, Boulder, Colorado, 1989.

[5] Nye, A. *Xlib Programming Manual. Volume One.* O'Reilly & Associates, Inc., Sebastapol, CA, 1990.

[6] *Unix System V Network Programmers Guide.* Prentice Hall Inc., Englewood Cliffs, New Jersey, 1987.

[7] *X/Open Portability Guide*, Networking Services, X/Open Company, Ltd. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[8] Rose, M. T. and Cass, D. E. "OSI Transport Services on top of the TCP." *Request for Comment 1006*, DDN Network Information Center, SRI International, May 1987.

[9] Piscitello, D. M. "Iternetworking in an OSI Environment." *Data Communications*, 15 (May 1986), 118-136.

[10] Weissberger, A. J. "What the New Internetworking Standards Provide." *Data Communications*, 16 (Feburary 1987), 141-156.

[11] Benhamou, E., Lefebvre, V. and Liu, V. "Practical Consideration in Building Large Internetworks." *Proceedings of the IEEE*, 12 Conferenece on Local Computer Networks, (October 1987), 23-39.

[12] Fluckiger, Francois. "Gateways and Converters in Computer Networks." *Computer Networks and ISDN Systems*, 16, Nos. 1 & 2 (September 1988), 55-59.

[13] Clyne, Les. "LAN/WAN Internetworking." *Computer Networks and ISDN Systems*, 16, Nos. 1 & 2 (September 1988), 34-39.

[14] Tillman, M. A. and Yen, D. "SNA and OSI: Three Strategies for Interconnection" *Communications of the ACM*, 33, No. 2 (Feburary 1990), 214-224.

[15] Groenbaek, I. "Conversion between the TCP and ISO Transport Protocols as a Method of Achieving Interoperability between Data Communication Systems." *IEEE Journal On Selected Areas In Communications*, SAC-4, No. 2 (March 1986), 288-296.

[16] Groenbaek, I. "The TCP and ISO transport service - A brief description and comparison." *SHAPE Techincal Center, Tech. Memo TM-726 (NATO) Unclassified Report,* February 1984.

[17] Okumura, K. "A formal Protocol Conversion Method." *SIGCOM'86 Symposium. Communication Architectures and Protocols*, 16, No. 3 (August 1986), 30-37.

[18] Yao, Y. W. et al. "A Modular Approach to Constructing Protocol Converters." *Proceedings,* IEEE INFOCOM'90 (June 1990), 572-572.

[19] Lam, S. S. "Protocol Conversion - Correctness Problem." *SIGCOM'86 Symposium. Communication Architectures and Protocols*, 16, No. 3 (August 1986), 19-29.

[20] Rose, M. T. " Transport Level Bridges from TCP/IP to OSI/ISO." *Connections,* 2, No. 1 (January 1988), 2-5.

[21] ISO 8348 Information processing systems - Data communications - *Network service definition.*

[22] Cass, D. and Rose, M. "ISO Transport Services on Top of the TCP." *Request for Comment 983.* DDN Network Information Center, SRI International, April 1986.

[23] Shukuya, S. et al. "Study of OSI Subsets from the SNA LU-6.2 Functional Viewpoint." *Papers presented at 30th Meeting of Japan Information processing Society, Tokyo,* March 1985.

[24] Staling, William. "A primer: Understanding Transport Protocols." In *Networking Software. Eds. Colin B. Ungaro, Data Communications Book Series. McGraw-Hill Information Systems Company, New York, New York,* 1987, 29-37.

[25] ISO 9074. Information processing systems - Open Systems Interconnection ESTELLE *-A Formal Description Technique Based on an Extended State transition Model,* (1987).

[26] ISO 8072. Information processing systems - Open Systems Interconnection - *Transport service definition,* (1986).

[27] ISO 8073. Information processing systems - Open Systems Interconnection - *Connection oriented transport protocol specification,* (1986).

[28] ISO 8073 DAD 2. Information processing systems - Open Systems Interconnection - *Connection oriented transport protocol specification Addendum 2: Class four operation over connectionless network service,* (1987).

[29] ISO 8602. Information processing systems - Open Systems Interconnection - *Protocol for providing the connectionless-mode transport service,* (1987).

[30] Postel, J. "Transmission Control Protocol." *Request For Comment 793.* Information Science Institute, University of Southern California, (Spetember 1981).

[31] MIL-STD-1778. *Military Standard Transmission Control Protocol,* (1983).

[32] Zatti, Stefano and Janson, Philippe. "Interconnecting OSI and Non-Osi Networks using an Integrated Directory service." *Computer Networks and ISDN Systems*, 15, No 4 (September 1988), 269-283.

[33] Sunshine, C. A. and Dalal, Y. K. "Connection Management in Transport Protocols." *Computer Networks*, 6, No. 2 (February 1978), 454-473.

[34] *Stable Implementation Agreements for Open Systems Interconnection Protocols.* Version 1 Edition 3. August 1988.

[35] Johns, M. St. "Draft Revised IP Security Option." *Request For Comment 1038.* DDN Network Information Center, SRI International, April 1986.

[36] Sidhu, Deepinder P. and Blumer, Thomas P. "Semi-automatic Implementation of OSI Protocols." *Computer Networks and ISDN Systems*, 18 (1989/90), 221-283.

10 APPENDIX A. CONNECTION ESTABLISHMENT PHASE

TCP_TCF Connection Establishment Phase

```
Specification TCP_TCF activity;

Const

    low = 0;
    high = (2**32) -1;
    MAX_NAME_LENGTH = 32;
    MAX_DATA     =  any integer;          { implementation specific }
    MAX_VTL_PACKET = any integer;         { system specific }

Type

    END_POINT_TYPE = ...;  { The end point is uniquely identified
                             by the tuple :
                             <Calling T-Address, Called T-Address  }


    VTPDU_CODE_TYPE = ( VTL_CR, VTL_CC, VTL_ACK);
    LOCAL_REF_TYPE = low..high;
    REMOTE_REF_TYPE = low..high;
    SEQ_TYPE     = low..high;
    OCTET        = 0..255;
    TWO_BYTES    = 0..2**16-1;
    FOUR_BYTES   = 0..2**32-1;
    CREDIT_TYPE = 0..255;
    NAME_LENGTH_TYPE = 1..MAX_NAME_LENGTH;
```

```
DATA_LENGTH_TYPE = 0..MAX_DATA;
BOOLEAN = 0..1;
TRUE = 1;
FALSE = 0;


TCP_PORT_TYPE = ..;      { external }

IP_ADDRESS_TYPE = ..;    { external }

SOCKET_TYPE =
      record
            Port    : TCP_PORT_TYPE;
            IPaddr : IP_ADDRESS_TYPE;
      end;

TP_NAME_TYPE =
      record
            len : MAX_LENGTH_TYPE;
            value : array[1..MAX_NAME_LEN] of OCTET;
      end;

NW_ADDRESS_TYPE =
      record
            Afi = AFI_TYPE;
            IpAddr = IP_ADDRESS_TYPE;
      end;

TP_ADDRESS_TYPE =
      record
            Tp_name : TP_NAME_TYPE;
            Nw_Address : NW_ADDRESS_TYPE;
      end;

DATA_TYPE
      record
            len   : DATA_LENGTH_TYPE;
            data : array[1..MAX_DATA] of OCTETS;
      end;


VTPDU_HEADER_TYPE =
      record
            VersionNo          : VERSION_TYPE;
            VtpduCode          : VTPDU_CODE_TYPE;
            Use_CheckSum       : 1_BIT;
            RemoteVTLendPt     : END_POINT_TYPE;
            Priority           : array[1..2] of 1_BIT;
            Security           : array[1..2] of 1_BIT;
            Reserved           : array [1..4] of 1_BIT;
      end;
```

```
CONREQ_VTPDU_TYPE  =
      record
            VtpduHeader          : VTPDU_HEADER_TYPE;
            SeqSpaceSize         : array[1..3] of 1_BIT;
            Allow_Expdt          : 1_BIT;
            Local_Refrence       : LOCAL_REF_TYPE;
            Remote_Refrence      : REMOTE_REF_TYPE;
            CallingTP            : TP_ADDRESS_TYPE;
            CalledTP             : TP_ADDRESS_TYPE;
            LocalVTLendPt        : END_POINT_TYPE;
            Credit               : CREDIT_TYPE;
            MaxPacketSize        : 0..MAX_DATA;
            Data                 : DATA_TYPE;
            CheckSum             : TWO_BYTES;
      end;


CONCNF_VTPDU_TYPE  =
      record
            VtpduHeader          : VTPDU_HEADER_TYPE;
            SeqSpaceSize         : array[1..3] of 1_BIT;
            Allow_Expdt          : 1_BIT;
            Local_Refrence       : LOCAL_REF_TYPE;
            Remote_Refrence      : REMOTE_REF_TYPE;
            CallingTP            : TP_ADDRESS_TYPE;
            LocalVTLendPt        : END_POINT_TYPE;
            Credit               : CREDIT_TYPE;
            MaxPacketSize        : 0..MAX_DATA;
            Data                 : DATA_TYPE;
            CheckSum             : TWO_BYTES;
      end;

ACK_VTPDU_TYPE  = ...;          { Defined in Appendix 2}


RST_REQ_VTPDU_TYPE  =  ...;          { Defined in Appendix 3}

var

     CrVtpdu      :   CONREQ_VTPDU_TYPE;
     CcVtpdu      :   CONCONF_VTPDU_TYPE;
     AckVtpdu     :   ACK_VTPDU_TYPE;
     RejVtpdu     :   RST_REQ_VTPDU_TYPE;
     TcpSeg       :   TCP_SEMENT_TYPE;


{ Channel Definitions for communication with the TCF Module }
```

```
Channel   IP_Access_Point  ( From_IP,  To_IP );

    by   From_IP  : ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                            Length : SEGMENT_LENGTH_TYPE,
                                            Security : SECURITY_TYPE,
                                            Precedence : PRECEDENCE_TYPE );

    by  To_IP  : SendTPDUrequest ( TcpSeg : TCP_SEGMENT_TYPE,
                                   Security : SECURITY_TYPE,
                                   Precedence : PRECEDENCE_TYPE );
                 ReceiveTPDUrequest ;


Channel   CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF  :  ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );


    { Module Header Definitaions }


Module InternetProtocol_Type  process ;

    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;

Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
          TcfToIp      : IP_Access_Point ( To_IP) individual queue;
          TcfToClts    : CLTS_Access_Point ( From_TCF) individual queue ;
end;


Module CLTS_Type process;

    ip    CltsToTcf  : CLTS_ACCESS_POINT (  To_TCF )

end;

         { Body Definitions for Modules }

Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body  for CLTS_Type; external;
```

```
Body TCF_Body for TCF_Type;


var

    LocalRef        :   LOCAL_REF_TYPE;
    RemoteRef       :   REMOTE_REF_TYPE;
    CallingTep      :   TP_ADDRESS_TYPE;
    CalledTep       :   TP_ADDRESS_TYPE:
    LocalVTL_endPt  :   END_POINT_TYPE;
    RemoteVTL_endPt :   END_POINT_TYPE:


state
    CLOSE, OPEN;       { state set of TCF }

     { Functions and procedures used in the module body }

function        TcfGetVtpdu (Vtpdu_Type : VTPDU_TYPE) : VTPDU_TYPE;
primitive  ;


function        TcfGetTcpSeg (Seg_Type:TCP_SEG_TYPE):TCP_SEGMENT_TYPE;
primitive  ;


procedure       CopyCalledTPname (CalledSocket : SOCKET_TYPE,
                                  var VtpName : TP_NAME_TYPE);
primitive  ;


procedure       CopyCallingTPname (CallingSocket : SOCKET_TYPE,
                                   var VtpName    : TP_NAME_TYPE);
primitive  ;


procedure       ComputeVTLCheckSum( var Vtpdu : VTPDU_TYPE);
primitive  ;


procedure       ComputeTCPCheckSum( var TcpSeg : TCP_SEGMENT_TYPE);
primitive  ;
```

```
function        VerifyTCPCheckSum( TcpSeg : TCP_SEGMENT_TYPE): boolean;
primitive  ;

procedure       CopyUserData (TcpData : array[1..length] of OCTET,
                              var VtpduData : DATA_TYPE,
                              DataLength : DATA_LENGTH_TYPE); primitive ;


function        IsSynReq ( TcpSeg : TCP_SEGMENT_TYPE) :  boolean;

begin
    IsSynReq : =  TcpSeg.Flags.SYN;
end;


function        IsSynAck ( TcpSeg : TCP_SEGMENT_TYPE ) : boolean;

begin
    IsSynAck :=  (TcpSeg.Flags.SYN  and TcpSeg.Flags.ACK);
end;


function        IsAck ( TcpSeg : TCP_SEGMENT_TYPE) : boolean;

begin
    IsAck := TcpSeg.Flags.Ack ;
end;


procedure       MapISNtoLocalRef ( Isn : SEQ_TYPE,
                                   var LocalRef : LOCAL_REF_TYPE);

begin
        LocalRef := Isn;
end;


procedure       StoreLocalRef ( Isn : SEQ_TYPE,
                                   var LocalRef : LOCAL_REF_TYPE);

begin
        LocalRef := Isn;
end;
```

```
procedure      StoreRemoteRef ( Remote_Ref : REMOTE_REF_TYPE,
                                var RemoteRef : REMOTE_REF_TYPE);

begin
          RemoteRef = Remote_Ref;
end;


procedure      MapLocalRefToSYN ( Isn : SEQ_TYPE,
                                  var LocalRef : LOCAL_REF_TYPE);

begin
          Isn := LocalRef;
end;


procedure      MapRemoteRefToAck ( RemoteRef : REMOTE_REF_TYPE,
                                   var Ack    : SEQ_TYPE);

begin
          Ack := RemoteRef + 1;
end;


procedure      MapCreditToTCP ( var TcpCredit : TCP_CREDIT_TYPE ,
                                    Vtpdu : CON_VTPDU_TYPE );

begin
          { TCP credit is expressed in number of octets }
          { VTL carries Credit information as number of }
          { packets, each of MaxVTLpacketSize.          }

          TcpCredit = Vtpdu.Credit * Vtpdu.Options.MaxVTLpacketSize;
end;


procedure      MapCreditToVTL ( TcpSeg : TCP_SEGMENT_TYPE ,
                           var VtpduCredit : VTL_CREDIT_TYPE );

begin
          { Tcp exposes credit in units of octets that the entity is }
          { willing to receive.  In the VTL domain, the unit of data  }
          { transfer is number of Packets.  Thus the equivalent TCP   }
          { windowm size is  TcpWindow / Max_VTL_PacketSize  packets  }

          VtpduCredit = INT( TcpSeg.TcpWindow / MAX_VTL_PACKET_SIZE );

end;
```

```
procedure      MapTcpSegmentSize ( TcpSeg : TCP_SEGMENT_TYPE ,
                                   var VtpduSize : DATA_LENGTH_TYPE );

begin
        VtpduSize = MIN ( TcpSeg.Options.MaxSeg, MAX_VTL_PACKET_SIZE);
end;


procedure      BuildCONREQvtpdu ( TcpSeg : TCP_SEGMENT_TYPE,
                                  Security : SECURITY_TYPE,
                                  Precedence : PRECEDENCE_TYPE,
                                  DataLength : DATA_LENGTH_TYPE,
                                  var Vtpdu : CON_VTPDU_SIZE );

begin
        CopyCallingTPname (TcpSeg.SourcePort, Vtpdu.CallingTPname);
        CopyCalledTPname (TcpSeg.DestinationPort, Vtpdu.CalledTPname);
        Vtpdu.VTLlocalEndpt = LocalVTL_endPt;
        Vtpdu.Priority = DEFAULT_PRECEDENCE;
        Vtpdu.Security = DEFAULT_SECURITY;
        Vtpdu.Use_CheckSum = TRUE;
        Vtpdu.Allow_Expdt = TRUE;
        Vtpdu.SeqSpaceSize = 4;
        CopyUserData(TcpSeg.Data, Vtpdu.Data, DataLenght);
        ComputeVTLCheckSum(Vtpdu);
end;


procedure      BuildCONCNFvtpdu ( TcpSeg : TCP_SEGMENT_TYPE,
                                  Security : SECURITY_TYPE,
                                  Precedence : PRECEDENCE_TYPE,
                                  DataLength : DATA_LENGTH_TYPE,
                                  var Vtpdu : CON_VTPDU_SIZE );

begin
        CopyCallingTPname (TcpSeg.SourcePort, Vtpdu.CallingTPname);
        Vtpdu.VTLlocalEndpt  = LocalVTL_endPt;
        Vtpdu.VTLremoteEndpt = RemoteVTL_endPt;
        Vtpdu.Priority = DEFAULT_PRECEDENCE;
        Vtpdu.Security = DEFAULT_SECURITY;
        Vtpdu.Use_CheckSum = TRUE;
        Vtpdu.Allow_Expdt = TRUE;
        Vtpdu.SeqSpaceSize = 4;
        CopyUserData(TcpSeg.Data, Vtpdu.Data, DataLenght);
        ComputeVTLCheckSum(Vtpdu);
end;
```

```
procedure       BuildACKvtpdu ( TcpSeg : TCP_SEGMENT_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                DataLength : DATA_LENGTH_TYPE,
                                var Vtpdu : CON_VTPDU_SIZE );

extern;         { In appendix 2 }

procedure       BuildRSTvtpdu ( TcpSeg : TCP_SEGMENT_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                DataLength : DATA_LENGTH_TYPE,
                                var Vtpdu : CON_VTPDU_SIZE );

extern;         { In appendix 3 }

procedure       BuildTcpSeg (var TcpSeg : TCP_SEGMENT_TYPE ,
                                  Vtpdu  : CON_VTPDU_TYPE );

begin
          CopyCallingTPname (TcpSeg.SourcePort, Vtpdu.CallingTPname);
          CopyCalledTPname (TcpSeg.DestinationPort, Vtpdu.CalledTPname);
          TcpSeg.Options.MaxSegmentSize = Vtpdu.Options.MaxVTLpacketSize;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Vtpdu.Data.DataLen);
          ComputeTCPCheckSum(TcpSeg);
end;

Initialize

    to CLOSE
          begin
                LcoalRef = 0;
                RemoteRef = 0;
                CallingTep = {0};
                CalledTep  = {0};
                LocalVTL_endPt = 0;
                RemoteVTL_endPt = 0;
          end;

          { transition part of TCF activity }

trans

          { transition due to interactions from IP }
```

```
from CLOSE to OPEN

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

      provided ( IsSynReq(TcpSeg) = TRUE and
                        VerifyTCPChecksum(TcpSeg))
      begin
            CrVtpdu = TcfGetVtpdu(VTL_CR);
            MapISNtoLocalRef (TcpSeg.Seq, CrVtpdu.LocalRef);
            StoreLocalRef(TcpSeg.Seq, LocalRef);
            MapCredit (TcpSeg.Credit, CrVtpdu.Credit);
            BuildCONREQVtpdu (TcpSeg, Security, Precedence, CrVtpdu);
            output TcfToClts.SendTCFdataRequest(CrVtpdu);
      end;


from OPEN to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

      provided ( IsSynAck(TcpSeg) = TRUE and
                        VerifyTCPChecksum(TcpSeg))
      begin
            CcVtpdu = TcfGetVtpdu(VTL_CC);
            MapISNtoLocalTRef (TcpSeg.Seq, CcVtpdu.LocalRef);
            StoreLocalRef(TcpSeg.Seq, LocalRef);
            MapACKtoRemoteRef (TcpSeg.Ack, CcVtpdu.RemoteRef);
            MapCredit (TcpSeg.Credit, CcVtpdu.Credit);
            BuildCONCNFVtpdu (TcpSeg, Security, Precedence, CcVtpdu);
            output TcfToClts.SendTCFdataRequest(CcVtpdu);
      end;


from OPEN to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

      provided ( IsAck(TcpSeg) = TRUE and
                        VerifyTCPChecksum(TcpSeg))
      begin
            AckVtpdu = TcfGetVtpdu(VTL_ACK);
```

```
            MapSeqToLocalRef(TcpSeg.Seq, AckVtpdu.LocalRef);
            MapAckToRemoteRef(Tcp.Ack, AckVtpdu.RemoteRef);
            BuildACKVtpdu (TcpSeg, Security, Precedence, AckVtpdu);
            output TcfToClts.SendTCFdataRequest(AckVtpdu);
      end;


from EITHER to CLOSE

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )


      provided ( IsRst(TcpSeg) = TRUE and
                        VerifyTCPChecksum(TcpSeg))
      begin
            RejVtpdu = TcfGetVtpdu(VTL_RST);
            RejVtpdu.LocalRef := LocalRef;
            BuildACKVtpdu (TcpSeg, Security, Precedence, AckVtpdu);
            output TcfToClts.SendTCFdataRequest(RejVtpdu);
      end;


      { transition due to interactions from CLTS }


from CLOSE to OPEN

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsVTLcr(Vtpdu) = TRUE and
                        VerifyVTLChecksum(Vtpdu))
      begin
            TcpSeg = TcfGetTcpSeg(SYN);
            MapLocalRefToSyn (CrVtpdu.LocalRef, TcpSeg.Seq);
            StoreRemoteRef (CrVtpdu.LocaRef, RemoteRef);
            RemoteVTL_endPt = Vtpdu.VTLlocalEndPt;
            MapCreditToTcp (TcpSeg.Credit, CrVtpdu.Credit);
            BuildTcpSeg (TcpSeg, CrVtpdu);
            output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                           DEFAULT_PRECEDENCE);
      end;


from OPEN to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)
```

```
        provided ( IsVTLcc(Vtpdu) = TRUE and
                        VerifyVTLChecksum(TcpSeg))
        begin
              TcpSeg = TcfGetTcpSeg(SYNACK);
              MapLocalRefToSyn (CcVtpdu.LocalRef, TcpSeg.Seq);
              MapRemoteRefToAck (CcVtpdu.RemoteRef, TcpSef.Ack);
              RemoteVTL_endPt = Vtpdu.VTLlocalEndPt;
              MapCreditToTcp (TcpSeg.Credit, CcVtpdu.Credit);
              BuildTcpSeg (TcpSeg, CcVtpdu);
              output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                              DEFAULT_PRECEDENCE);
        end;

    from OPEN to same

    when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

        provided ( IsVTLack(Vtpdu) = TRUE and
                        VerifyVTLChecksum(TcpSeg))
        begin
              TcpSeg = TcfGetTcpSeg(ACK);
              MapLocalRefToSeq (AckVtpdu.LocalRef, TcpSeg.Seq);
              MapRemoteRefToAck (AckVtpdu.RemoteRef, TcpSef.Ack);
              MapCreditToTcp (TcpSeg.Credit, AckVtpdu.Credit);
              BuildTcpSeg (TcpSeg, AckVtpdu);
              output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                              DEFAULT_PRECEDENCE);

        end;

    from EITHER to CLOSE

    when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

        provided ( IsVTLrst(Vtpdu) = TRUE and
                        VerifyVTLChecksum(TcpSeg))
        begin
              TcpSeg = TcfGetTcpSeg(RST);
              MapLocalRefToSeq (RstVtpdu.LocalRef, TcpSeg.Seq);
              BuildTcpSeg (TcpSeg, RstVtpdu);
              output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                              DEFAULT_PRECEDENCE);

        end;

end; { end of TCP_TCP_BODY }

    { Module-variable-declaration-part of specification }
```

```
modvar

     InternetProtocol   : InternetProtocol_Type;
     TCP_TCF            : array[END_POINT_TYPE] of TCF_Type;
     CLTS               : CLTS_TYPE;

     { Initialization Part Of specification }

Initialize

     begin   { module initialization }

          init InternetProtocol with InternetProtocol_Body;
          init CLTS with CLTS_Body;

          all end_point : END_POINT_TYPE do
          begin
               init TCP_TCF with TCF_Body(end_point);

                    { connect interaction points }

               connect InternetProtocol.IptoTCF to
                       TCP_TCF.TcfToIP[end_point];

               connect TCP_TCF[end_point].TcfToCLTS to
                       CLTS.CltsTpTcf;
          end;

end. { end of specification }
```

TCP_TCF Connection Establishment Phase

```
Specification TP4_TCF activity;

Const

    low = 0;
    high = (2**32) -1;
    MAX_NAME_LENGTH = 32;
    MAX_DATA    =  any integer;          { implementation specific }
    MAX_VTL_PACKET = any integer;        { system specific }

Type

    END_POINT_TYPE = ...;   { The end point is uniquely identified }
                            { by the 3 tuple <SrcRefId, DestRefId, Nsap>}
    TP4_CR = ..;
    TP4_CC = ..;
    TP4_AK = ..;
    TP4_TPDU_TYPE = (TP4_CR, TP4_CC, TP4_AK);
    VTPDU_CODE_TYPE = ( VTL_CR, VTL_CC, VTL_ACK);
    LOCAL_REF_TYPE = low..high;
    REMOTE_REF_TYPE = low..high;
    SEQ_TYPE    = low..high;
    OCTET       = 0..255;
    1_BIT       = ..;
    TWO_BYTES   = 0..2**16-1;
    FOUR_BYTES  = 0..2**32-1;
    CREDIT_TYPE = 0..255;
    NAME_LENGTH_TYPE = 1..MAX_NAME_LENGTH;
    DATA_LENGTH_TYPE = 0..MAX_DATA;
    BOOLEAN = 0..1;
    TRUE = 1;
    FALSE = 0;


    NSAP_ADDRESS_TYPE = ..;      { external as specified by ISO IP }

    TSEL_TYPE = ..;               { external; as specified by ISO TP }

    TSAP_ADDRESS_TYPE
          record
                Tselector     : TSEL_TYPE;
                NsapAddress   : NSAP_ADDRESS_TYPE;
          end
```

```
DATA_TYPE
      record
            len   : DATA_LENGTH_TYPE;
            data : array[1..MAX_DATA] of OCTETS;
      end;


VTPDU_HEADER_TYPE =
      record
            VersionNo          : VERSION_TYPE;
            VtpduCode          : VTPDU_CODE_TYPE;
            Use_CheckSum       : 1_BIT;
            RemoteVTLendPt     : END_POINT_TYPE;
            Priority           : array[1..2] of 1_BIT;
            Security           : array[1..2] of 1_BIT;
            Reserved           : array [1..4] of 1_BIT;
      end;


CONREQ_VTPDU_TYPE  =
      record
            VtpduHeader        : VTPDU_HEADER_TYPE;
            SeqSpaceSize       : array[1..3] of 1_BIT;
            Allow_Expdt        : 1_BIT;
            Local_Refrence     : LOCAL_REF_TYPE;
            Remote_Refrence    : REMOTE_REF_TYPE;
            CallingTP          : TP_ADDRESS_TYPE;
            CalledTP           : TP_ADDRESS_TYPE;
            LocalVTLendPt      : END_POINT_TYPE;
            Credit             : CREDIT_TYPE;
            MaxPacketSize      : 0..MAX_DATA;
            Data               : DATA_TYPE;
            CheckSum           : TWO_BYTES;
      end;


CONCNF_VTPDU_TYPE  =
      record
            VtpduHeader        : VTPDU_HEADER_TYPE;
            SeqSpaceSize       : array[1..3] of 1_BIT;
            Allow_Expdt        : 1_BIT;
            Local_Refrence     : LOCAL_REF_TYPE;
            Remote_Refrence    : REMOTE_REF_TYPE;
            CallingTP          : TP_ADDRESS_TYPE;
            LocalVTLendPt      : END_POINT_TYPE;
            Credit             : CREDIT_TYPE;
            MaxPacketSize      : 0..MAX_DATA;
            Data               : DATA_TYPE;
            CheckSum           : TWO_BYTES;
      end;

ACK_VTPDU_TYPE  = ...;          { Defined in Appendix 2}
```

```
    RST_REQ_VTPDU_TYPE  =  ...;          { Defined in Appendix 3}

var

    CrVtpdu      :   CONREQ_VTPDU_TYPE;
    CcVtpdu      :   CONCONF_VTPDU_TYPE;
    AckVtpdu     :   ACK_VTPDU_TYPE;
    RejVtpdu     :   RST_REQ_VTPDU_TYPE;

    Tp4CR        :   ISO_TP4_CR_TYPE;
    Tp4CC        :   ISO_TP4_CC_TYPE;
    Tp4AK        :   ISO_TP4_AK_TYPE

{ Channel Definitions for communication with the TCF Module }

Channel  IP_Access_Point  ( From_IP,  To_IP );

    by   From_IP  : ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE );

    by   To_IP    : SendTPDUrequest ( Tp4Tpdu : TP4_TPDU_TYPE );

                    ReceiveTPDUrequest ;

Channel  CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF   :  ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );


    { Module Header Definitaions }

Module InternetProtocol_Type  process ;

    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;

Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
```

```
            TcfToIp     : IP_Access_Point ( To_IP) individual queue;
            TcfToClts   : CLTS_Access_Point ( From_TCF) individual queue ;
end;

Module CLTS_Type process;

    ip    CltsToTcf  : CLTS_ACCESS_POINT (  To_TCF )

end;

          { Body Definitions for Modules }

Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body  for CLTS_Type; external;


Body TCF_Body for TCF_Type;

var

    LocalRef            :  LOCAL_REF_TYPE;
    RemoteRef           :  REMOTE_REF_TYPE;
    CallingTSAPaddress  :  TSAP_ADDRESS_TYPE;
    CalledTSAPaddress   :  TSAP_ADDRESS_TYPE;
    LocalVTL_endPt      :  END_POINT_TYPE;
    RemoteVTL_endPt     :  END_POINT_TYPE;

{The following variables define the behaviour of the local
transport}

    LocalSeqSize        :  ONE_BYTE;
    UseLocalCheckSum    :  boolean;
    UseLocalExpedt      :  boolean;


state
     EITHER = (CLOSE, OPEN);       { state set of TCF }

     { Functions and procedures used in the mnodule body }

function       TcfGetVtpdu (Vtpdu_Type : VTPDU_TYPE) : VTPDU_TYPE;
primitive  ;


function       TcfGetTp4Pdu ( Tpdu_Type : ISO_TPDU_TYPE) : ISO_TPDU_TYPE;
primitive  ;
```

```
function        IsNormalFormat (Tp4ConReq : ISO_TP4_CR_TYPE) : boolean;
primitive  ;

procedure       CopyCalledTPname (CalledTSAPaddress : TSAP_ADDRESS_TYPE,
                                  var VtpName : TP_NAME_TYPE);
primitive  ;

procedure       CopyCallingTPname (CallingTSAPaddress:TSAP_ADDRESS_TYPE,
                                  var VtpName : TP_NAME_TYPE);
primitive  ;

procedure       ComputeVTLCheckSum( var Vtpdu : VTPDU_TYPE);
primitive  ;

procedure       ComputeTP4CheckSum( var Tp4Tpdu : ISO_TPDU_TYPE);
primitive  ;

function        VerifyVTLCheckSum( Vtpdu : VTPDU_TYPE): boolean;
primitive  ;

function        VerifyTP4CheckSum( Tp4Pdu : ISO_TPDU_TYPE): boolean;
primitive  ;

procedure       CopyUserData (ConData : array[1..length] of OCTET,
                              var VtpduData : DATA_TYPE,
                              DataLength : DATA_LENGTH_TYPE);
primitive  ;

procedure       MapSrcRefToLocalRef ( SrcRef : TP4_REFRENCE_TYPE,,
                                      var LocalRef : LOCAL_REF_TYPE);

begin
        LocalRef := SrcRef;
end;

procedure       StoreLocalRef ( SrcRef : TP4_REFRENCE_TYPE,,
                                var LocalRef : LOCAL_REF_TYPE);

begin
        LocalRef := SrcRef;
end;
```

```
procedure       StoreRemoteRef ( Remote_Ref : REMOTE_REF_TYPE,
                                 var RemoteRef : REMOTE_REF_TYPE);

begin
          RemoteRef := Remote_Ref;
end;


procedure       MapLocalRefToSrcRef ( var SrcRef : TP4_REFRENCE_TYPE,
                                       LocalRef   : LOCAL_REF_TYPE);
primitive;

          { A local mapping is done to compensate for the size }
          { difference between the VTL refrence field size (4 bytes) }
          { and the ISO TP4 Refrence_ID which is 2 Bytes


procedure       MapRemoteRefToDstRef ( RemoteRef : REMOTE_REF_TYPE,
                                       var DestRef    : TP4_REFRENCE_TYPE);

begin
          DstRef := RemoteRef;
end;

procedure       MapCreditToTP4 ( var TP4Credit : TP4_CREDIT_TYPE ,
                                     Vtpdu     : CON_VTPDU_TYPE );

begin

          Tp4Credit = Vtpdu.Credit;
end;


procedure       MapCreditToVTL ( Tp4Credit : TP4_CREDIT_TYPE ,
                            var VtlCredit : VTL_CREDIT_TYPE );

begin
          VtlCredit = Tp4Credit;
end;


procedure       MapTp4TpduSize ( Tp4TpduSize    : DATA_LENGTH_TYPE ,
                                 var VtpduSize : DATA_LENGTH_TYPE );

begin
          VtpduSize = MIN ( Tp4TpduSize, MAX_VTL_PACKET_SIZE);
```

```
end;


procedure      BuildCONREQvtpdu ( Tp4Tpdu : ISO_TPDU_TYPE,
                                  DataLength : DATA_LENGTH_TYPE,
                                  var Vtpdu  : CON_VTPDU_SIZE );

begin
          CopyCallingTPname (Tp4Tpdu.CallingTsapAddr, Vtpdu.CallingTPname);
          CopyCalledTPname  (Tp4Tpdu.CalledTsapAddr, Vtpdu.CalledTPname);
          Vtpdu.VTLlocalEndpt = LocalVTL_endPt;
          Vtpdu.Priority = DEFAULT_PRECEDENCE;
          Vtpdu.Security = DEFAULT_SECURITY;
          If(Tp4Tpdu.OPtions.Use_CheckSum = TRUE) {
                Vtpdu.Options.Use_CheckSum = TRUE;
          } else {
                Vtpdu.Options.Use_CheckSum = FALSE;
          }

          if(Tp4Tpdu.Options.Expdt_data = TRUE){
                Vtpdu.Options.Allow_Expdt = TRUE;
          } else {
                Vtpdu.Options.Allow_Expdt = TRUE;
          }

          if(Tp4Tpdu.Options.NormalFormat = TRUE) {
                Vtpdu.Options.SeqSpaceSize = 2;
          } else {
                Vtpdu.Options.SeqSpaceSize = 4;
          }

          CopyUserData(TcpSeg.Data, Vtpdu.Data, DataLenght);
          ComputeVTLCheckSum(Vtpdu);
end;


procedure      BuildCONCNFvtpdu ( Tp4Tpdu : ISO_TPDU_TYPE,
                                  Security : SECURITY_TYPE,
                                  Precedence : PRECEDENCE_TYPE,
                                  DataLength : DATA_LENGTH_TYPE,
                                  var Vtpdu : CON_VTPDU_SIZE );

begin
          CopyCallingTPname (Tp4Tpdu.CallingTsapAddr, Vtpdu.CallingTPname);
          Vtpdu.VTLlocalEndpt  = LocalVTL_endPt;
          Vtpdu.VTLremoteEndpt = RemoteVTL_endPt;
          Vtpdu.Priority = DEFAULT_PRECEDENCE;
          Vtpdu.Security = DEFAULT_SECURITY;
          If(Tp4Tpdu.OPtions.Use_CheckSum = TRUE) {
                Vtpdu.Options.Use_CheckSum = TRUE;
```

```
        } else {
              Vtpdu.Options.Use_CheckSum = FALSE;
        }

        if(Tp4Tpdu.Options.Expdt_data = TRUE){
              Vtpdu.Options.Allow_Expdt = TRUE;
        } else {
              Vtpdu.Options.Allow_Expdt = TRUE;
        }

        if(Tp4Tpdu.Options.NormalFormat = TRUE) {
              Vtpdu.Options.SeqSpaceSize = 2;
        } else {
              Vtpdu.Options.SeqSpaceSize = 4;
        }

        CopyUserData(TcpSeg.Data, Vtpdu.Data, DataLenght);
        ComputeVTLCheckSum(Vtpdu);
end;


procedure     BuildACKvtpdu ( Tp4Tpdu : ISO_TPDU_TYPE,
                              var Vtpdu : CON_VTPDU_SIZE );

extern;       { in appendix 2 }


procedure     BuildRSTvtpdu ( Tp4Tpdu : ISO_TPDU_TYPE,
                              var Vtpdu : CON_VTPDU_SIZE );

extern;       { in appendix 3 }


procedure     BuildTp4Tpdu (var Tp4Tpdu : TP4_TPDU_TYPE ,
                                Vtpdu   : CON_VTPDU_TYPE );

begin

        CopyCallingTPname (Tp4Tpdu.CallingTsapAdr, Vtpdu.CallingTPname);
        CopyCalledTPname  (Tp4Tpdu.CalledTsapAdr, Vtpdu.CalledTPname);
        Tp4Tpdu.Options.MaxTpduSize = Vtpdu.Options.MaxVTLpacketSize;

        If(Vtpdu.OPtions.Use_CheckSum = TRUE) {
              Tp4Tpdu.Options.UseCheckSum = TRUE;
        } else {
              Tp4Tpdu.Options.UseCheckSum = FALSE;
        }

        if(Vtpdu.Options.Expdt_data = TRUE){
              Tp4Tpdu.Options.Allow_Expdt = TRUE;
        } else {
```

```
                Tp4Tpdu.Options.Allow_Expdt = TRUE;
        }

        if(Vtpdu.Options.SeqSpaceSize = 2) {
                Tp4Tpdu.Options.NormalFormat = TRUE;
        } else {
                Tp4Tpdu.Options.NormalFormat = FALSE;
        }

        CopyUserData(Tp4Tpdu.Data, Vtpdu.Data, Vtpdu.Data.DataLen);

        If(UseLocalCheckSum) {
                ComputeTP4CheckSum(Tp4Tpdu);
        }
end;

Initialize

    to CLOSE
        begin

                LcoalRef = 0;
                RemoteRef =0;
                CallingTSAPaddress = {0};
                CalledTSAPaddress  = {0};
                LocalVTL_endPt = 0;
                RemoteVTL_endPt = 0;
                UseLocalCheckSum = TRUE ;
                LocalSeqSize = 4;
                UseLocalExpdt = FALSE;

        end;

        { transition part of TCF activity }

trans

        { transitions due to interactions from IP module }

    from CLOSE to OPEN

    when  IPtoTCF.ReceiveTPDUindication (Tp4Tpdu : TP4_TPDU_TYPE )

        provided ( IsConReq(Tp4Tpdu) = TRUE and
                   VerifyTp4CheckSum(Tp4Tpdu = TRUE))
        begin
                CrVtpdu = TcfGetVtpdu(VTL_CR);
                MapSrcRefToLocalRef (Tp4Tpdu.SrcRef, CrVtpdu.LocalRef);
```

```
            StoreLocalRef(Tp4Tpdu.SrcRef, LocalREf);
                                    { Lcoal variable for module }
            CrVtpdu.RemoteRef = 0;
            MapCredit (Tp4Tpdu.Credit, CrVtpdu.Credit);
            BuildCONREQVtpdu (Tp4Tpdu, CrVtpdu);
            output TcfToClts.SendTCFdataRequest(CrVtpdu);
      end;

from OPEN to same

when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE )

      provided ( IsConCnf(Tp4Tpdu) = TRUE and
                  VerifyTp4CheckSum(Tp4Tpdu = TRUE))
      begin
            CcVtpdu = TcfGetVtpdu(VTL_CC);
            MapSrcRefToLocalTRef (Tp4Tpdu.SrcRef, CcVtpdu.LocalRef);
            MapDestRefToRemoteRef (Tp4Tpdu.DestRef, CcVtpdu.RemoteRef);
            StoreRemoteRef(CcVtpdu.RemoteRef, RemoteRef);
            MapCredit (Tp4Tpdu.Credit, CcVtpdu.Credit);

            if(Tp4Tpdu.Options.UseCheckSum = TRUE){
                  UseLocalCheckSum := TRUE;
            }

            if(Tp4Tpdu.Options.NormalFormat = TRUE){
                  LocaSeqSize = 2;
            } else {
                  LocaSeqSize = 4;
            }

            If(Tp4Tpdu.Options.UseExpdt = TRUE) {
                  UseLocalExpdt = TRUE;
            } else {
                  UseLocalExpdt = FALSE;
            }

            BuildCONCNFVtpdu (Tp4Tpdu, CcVtpdu);
            output TcfToClts.SendTCFdataRequest(CcVtpdu);
      end;

from OPEN to SAME

when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE)

      provided ( IsAck(Tp4Tpdu) = TRUE and
                  VerifyTp4CheckSum(Tp4Tpdu = TRUE))
      begin
            AckVtpdu = TcfGetVtpdu(VTL_ACK);
```

```
            AckVtpdu.LocalRef = Localref;
            MapDestRefToRemoteRef(Tp4Tpdu.DestRef, AckVtpdu.RemoteRef);
            BuildACKVtpdu (Tp4Tpdu, AckVtpdu);
            output TcfToClts.SendTCFdataRequest(AckVtpdu);
      end;

from EITHER to CLOSE

when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE)

      provided ( IsDR(Tp4Tpdu) = TRUE and
                 VerifyTp4CheckSum(Tp4Tpdu = TRUE))
      begin
            RstVtpdu = TcfGetVtpdu(VTL_RST);
            MapDestRefToRemoteRef(Tp4Tpdu.DestRef, RstVtpdu.RemoteRef);
            MapSrcRefToLocaRef(Tp4Tpdu.SrcRef, RstVtpdu.LocaRef);
            BuildRSTVtpdu (Tp4Tpdu, RstVtpdu);
            output TcfToClts.SendTCFdataRequest(RstVtpdu);
      end;


      { Transitions due interactions from CLTS }

from CLOSE to OPEN

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsVTLcr(Vtpdu) = TRUE and
                 VerifyVTLCheckSum(Vtpdu = TRUE))
      begin
            Tp4Vtpdu = TcfGetTp4Tpdu ( CR );
            MapLocalRefToSrcRef (CrVtpdu.LocalRef, Tp4Tpdu.SrcRef);
            StoreRemoteRef(CrVtpdu.LocalRef, Remoteref);
            Tp4Tpdu.DestRef = 0;
            MapCreditToTp4 (Tp4Tpdu.Credit, CrVtpdu.Credit);
            BuildTcpSeg (Tp4Tpdu, CrVtpdu);
            output TcfToIp.SendTPDUrequest(Tp4Tpdu );
      end;


from OPEN to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsVTLcc(Vtpdu) = TRUE and
                 VerifyVTLCheckSum(Vtpdu = TRUE))
      begin
            Tp4Tpdu = TcfGetTp4Tpdu(CC);
```

```
                MapLocalRefToSrcRef (CcVtpdu.LocalRef, Tp4Tpdu.SrcRef);
                MapRemoteRefToDestRef (CcVtpdu.RemoteRef, Tp4Tpdu.DestRef)
                MapCreditToTp4 (Tp4Tpdu.Credit, CcVtpdu.Credit);

                if(Vtpdu.Use_CheckSum = TRUE){
                      UseLocalCheckSum := TRUE;
                }

                LocalSeqSize = Vtpdu.SeqSpaceSize;

                If(UseExpdt = TRUE) {
                      UseLocalExpdt = TRUE;
                } else {
                      UseLocalExpdt = FALSE;
                }

                BuildTp4Tpdu (Tp4Tpdu, CrVtpdu);
                output TcfToIp.SendTPDUrequest( Tp4Tpdu );
          end;


     from OPEN to same

     when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

          provided ( IsVTLack(Vtpdu) = TRUE and
                      VerifyVTLCheckSum(Vtpdu = TRUE))
          begin
                Tp4Tpdu = TcfGetTp4Tpdu(AK);
                MapRemoteRefToDestRef (AckVtpdu.RemoteRef, Tp4Tpdu.DestRef);
                MapCreditToTp4 (Tp4Tpdu.Credit, AckVtpdu.Credit);
                BuildTp4Tpdu (Tp4Tpdu, AckVtpdu);
                output TcfToIp.SendTPDUrequest( Tp4Tpdu);

          end;

     from EITHER to CLOSE

     when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

          provided ( IsVTLrst(Vtpdu) = TRUE and
                     VerifyVTLCheckSum(Vtpdu = TRUE))
          begin
                Tp4Tpdu = TcfGetTp4Tpdu(DR);
                MapRemoteRefToDestRef (RstVtpdu.RemoteRef, Tp4Tpdu.DestRef);
                MapLocalRefToSrcRef (RstVtpdu.LocalRef, Tp4Tpdu.SrcRef);
                BuildTp4Tpdu (Tp4Tpdu, RstVtpdu);
                output TcfToIp.SendTPDUrequest( Tp4Tpdu);
          end;

end; { end of TP4_TCP_BODY }
```

```
    { Module-variable-declaration-part  of  specification }

modvar

    InternetProtocol  : InternetProtocol_Type;
    ISO_TP4_TCF       : array[END_POINT_TYPE] of TCF_Type;
    CLTS              : CLTS_TYPE;

    { Initialization  Part  Of  specification }

Initialize

    begin  { module initialization }

        init InternetProtocol with InternetProtocol_Body;
        init CLTS with CLTS_Body;

        all end_point : END_POINT_TYPE do
        begin
            init ISO_TP4_TCF with TCF_Body(end_point);

                { connect  interaction  points }

            connect InternetProtocol.IptoTCF to
                    ISO_TP4_TCF.TcfToIP[end_point];

            connect ISO_TP4_TCF[end_point].TcfToCLTS to
                    CLTS.CltsTpTcf;
        end;

end. { end of specification }
```

11 APPENDIX B. DATA TRANSFER PHASE

TCP_TCF Data Transfer Phase Specification

type

```
VSN_TYPE =
      record
            BindingType          : ONE_BYTE;
            OctetBinding         : FOUR_BYTES;
            PacketBinding        : TWO_BYTES;
      end

OCTET_BINDING  = 0x01;
PACKET_BINDING = 0x02;

MSG_HEADER_TYPE =
      record
            VersionNo            : VERSION_TYPE;
            RemoteVTLendPt       : END_POINT_TYPE;
            MsgCode              : MSG_CODE_TYPE;
      end;

TCF_VSNMSG_TYPE =
      record
            MsgHdr               : MSG_HEADER_TYPE;
            NumVsnRec            : ONE_BYTE;
            VsnRecord            : array[1..NumRec] of VSN_TYPE;
      end;
```

```
            VsnRecord           : array[1..NumRec] of VSN_TYPE;
      end;

PIGGYBACK_ACK_TYPE =
      record
            NextRecvVSN         : VSN_TYPE;
            Credit              : CREDIT_TYPE;
      end;

DATA_OPTIONS_TYPE =
      record
            OptionsFlags        : ONE_BYTE;
            PiggyBackAck        : PIGGYBACK_ACK_TYPE;
      end;

{ The encoding of the OptionsFlag in the data options is as : }

PIGGYBACK_ACK = 0x01;

VTPDU_HEADER_TYPE =
      record
            VersionNo           : VERSION_TYPE;
            VtpduCode           : VTPDU_CODE_TYPE;
            Use_CheckSum        : 1_BIT;
            RemoteVTLendPt      : END_POINT_TYPE;
            Priority            : array[1..2] of 1_BIT;
            Security            : array[1..2] of 1_BIT;
            Reserved            : array [1..4] of 1_BIT;
      end;


DATA_TPDU_TYPE  =
      record
            VtpduHdr            : VTPDU_HEADER_TYPE;
            SendVSN             : VSN_TYPE;
            OptionsFlags        : ONE_BYTE;
            DataOptions         : DATA_OPTIONS_TYPE;
            Data                : DATA_TYPE;
            CheckSum            : TWO_BYTES;
      end;


{ ACK_OPTIONS is a variant record, with Mask field describing which }
{ of the options are carried in the VTPDU                           }

ACK_OPTIONS_TYPE =
      record
            OptionsMask         : ONE_BYTE;
            UrgentDataSize      : TWO_BYTES;
            AckSequence         : TWO_BYTES;
            FccInfo             : array[1..8] of ONE_OCTET;
```

```
      end;

{ The encoding of the ACK OptionsMask is as :                    }


URG_DATA_LENGTH    = 0x01;
ACK_SEQ            = 0x02;
FCC_REQ            = 0x04;
EXPDT_ACK          = 0x08;


DATA_ACK_VTPDU_TYPE  =
      record
            VtpduHdr           : VTPDU_HEADER_TYPE;
            NextRecv           : VSN_TYPE;
            Credit             : CREDIT_TYPE;
            AckOptions         : ACK_OPTIONS_TYPE;
            CheckSum           : TWO_BYTES;
      end;


EXPEDITED_DATA_TPDU_TYPE  =
      record
            VtpduHdr           : VTPDU_HEADER_TYPE;
            ExpdtSendSeq       : VSN_TYPE;
            TotalEDlen         : DATA_LENGTH_TYPE;
            DataOptions        : DATA_OPTIONS_TYPE;
            Data               : DATA_TYPE;
            CheckSum           : TWO_BYTES;
      end;


RST_REQ_VTPDU_TYPE  = ...;          { defined in appendix 3 }


{ TCP Sequence number to VSN mapping table element       }

SEND_MAP_TABLE_ELEMENT_TYPE =
      record
            Vsn          :  VSN_TYPE;
            DataLength   :  DATA_LENGTH_TYPE;
            PacketType   :  1_BIT;          { Set to 1 if mapping is    }
      end;                                  { done to/from EXPDT VTPDU }


RECV_MAP_TABLE_ELEMENT_TYPE =
      record
            Vsn          :  VSN_TYPE;
            DataLength   :  DATA_LENGTH_TYPE;
            ExpectedAck  :  TCP_SEQNUM_TYPE;  { = octet binding + length}
            PacketType   :  1_BIT;          { Set to 1 if mapping is    }
```

```
        end;                                    { done to/from EXPDT VTPDU }

var

    DataVtpdu          :   DATA_VTPDU_TYPE;
    DataAckVtpdu       :   DATA_ACK_VTPDU_TYPE;
    ExpdtDataVtpdu     :   EXPEDITED_DATA_VTPDU_TYPE;
    ExpdtAckVtpdu      :   EXPEDITED_ACK_VTPDU_TYPE;
    RejVtpdu           :   RST_REQ_VTPDU_TYPE;
    TcpSeg             :   TCP_SEMENT_TYPE;
    MappingInfoMsg     :   TCF_MSG_TYPE;
    SendListElement    :   SEND_MAP_TABLE_ELEMENT_TYPE;
    RecvListElement    :   RECV_MAP_TABLE_ELEMENT_TYPE;


{ Channel Definitions for communication with the TCF Module }

Channel  IP_Access_Point  ( From_IP,  To_IP );

    by   From_IP  : ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                            Length : SEGMENT_LENGTH_TYPE,
                                            Security : SECURITY_TYPE,
                                            Precedence : PRECEDENCE_TYPE );

    by  To_IP   : SendTPDUrequest ( TcpSeg : TCP_SEGMENT_TYPE,
                                    Security : SECURITY_TYPE,
                                    Precedence : PRECEDENCE_TYPE );
                  ReceiveTPDUrequest ;


Channel  CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF  :  ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );


Channel  COTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFMsgRequest;
                    SendTCFMsgRequest ( TcfMsg : TCF_MESSAGE_TYPE );

    by  To_TCF  :  ReceiveTCFMsgIndication ( TcfMsg : TCF_MESSAGE_TYPE );


              { Module Header Definitaions }
```

```
Module InternetProtocol_Type  process ;

    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;

Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
          TcfToIp      : IP_Access_Point ( To_IP) individual queue;
          TcfToClts    : CLTS_Access_Point ( From_TCF) individual queue ;
          TcfToCots    : COTS_Access_Point ( From_TCF) individual queue ;
end;


Module CLTS_Type process;

    ip    CltsToTcf  : CLTS_ACCESS_POINT ( To_TCF )

end;


Module COTS_Type process;

    ip    CotsToTcf  : COTS_ACCESS_POINT ( To_TCF )

end;

                { Body Definitions for Modules }


Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body  for CLTS_Type; external;

Body      Cots_Body  for COTS_Type; external;


Body TCF_Body for TCF_Type

var

    SendMapList  : ..;      { Actual data structure is left as }
                            { implementation dependent        }
    SendMapListIndex : ...;

    PartialVsnIndex : ...;

    RecvMapList  : ..;      { Actual data structure is left as }
```

```
                        { implementation dependent          }
    SendPacketSize : DATA_LENGTH_TYPE;

    RecvPacketSize : DATA_LENGTH_TYPE;

    SendSeqNum     : TCP_SEQNUM_TYPE;          { current TCP send seq }

    Next_SendSeqNum : TCP_SEQNUM_TYPE:         { The next expected send
                                 seq that will be carried by the TCP
                                 segment = SendSeqNum + seg.length }
    SendPacketNum  : integer;

    PreviousNextRecvVSN : VSN_TYPE;

    PreviousCreditValue : CREDIT_TYPE;

    RecvSeqNum     : TCP_SEQNUM_TYPE;

    Next_RecvSeqNum     : TCP_SEQNUM_TYPE;

    RecvPacketNum  : integer;


     { Functions and procedures used in the module body }

function       TcfGetVtpdu (Vtpdu_Type : VTPDU_TYPE) : VTPDU_TYPE;
primitive  ;


function       TcfGetTcpSeg (Seg_Type : TCP_SEG_TYPE) : TCP_SEGMENT_TYPE;
primitive  ;


function       IsUrg (Seg_Type : TCP_SEG_TYPE) : boolean;
primitive  ;


procedure      ClearSendMapList();
primitive  ;

procedure      ClearRecvMapList();
primitive  ;

procedure      UpdateSendMapList(VtlSeq : VSN_TYPE,
                                 SegLength : DATA_LENGTH_TYPE,
                                 ExpdtFlag : char);
```

```
          { allocate a SendListElement and add to head of FIFO send queue }
primitive  ;


procedure      UpdateRecvMapList(Vtpdu : VTPDU_TYPE,
                                 SegLength : DATA_LENGTH,
                                 ExpdtFlag : char);

          { If (Vsn < Next_RecvVsn ) Then it is a retransmission.
               check the Recvmaplist to see if the octet binding was
               present.  If not then free the buffers in the recv buffer
               list.  Fill in the Octet binding info in the RecvMapList }
          { Else Allocate a RecvListElement and add to head of FIFO
            recv queue }
primitive  ;


procedure      SetIncompleteVsnIndex(PartialVsnIndex);
primitive  ;

procedure      SetPiggyBackOption(DataVtpdu);
primitive  ;


function       BufferVtpduPolicy()   : boolean;
primitive  ;

procedure      BufferVtpdus(DataVtpdu : DATA_VTPDU_TYPE);
primitive  ;

function       IsOctetBinding(Vsn : VSN_TYPE)   : boolean;
primitive  ;

function       IsMissingSeg(TcpSeqNum : SEQ_NUM_TYPE,
                            Vsn : VSN_TYPE ) : boolean ;

               {
                     check the SendMapList for this TcpSeqNum;
                     check if VSN entry is incomplete
                           return true if it is
                      else return false
               }
 primitive  ;



procedure      UpdateVsnInfo(TcpSeq, Vsn, SegLength, ExpdtFlag);

           { Update the VSN map table with the missing packet number
             portion. }
primitive  ;
```

```
function        BuildMapInfo(TcpSeq : TCP_SEQ_TYPE,
                             Vsn : VSN_TYPE)   : TCF_MESSAGE_TYPE,
begin
          { the VSN Map Message has the following format :
                Remote_End_Point_Id,
                Number of VSNs, < VSN1, ...., VSNp>
          }
          return (MapMsg);
end;


procedure       MapSeqToVSN(TcpSeqNum : SEQ_NUM_TYPE,
                            SegLength : DATA_LENGTH_TYPE,
                            var VtlSeq : VSN_TYPE,
                            ExpdtFlag : char);

begin

var PacketNum;


    if (TcpSeqNum = Next_SendSeqNum ) then
      begin
          { This is the next in sequence segment }
          VtlSeq.VsnType := (OCTET_BINDING logical OR
                             PACKET_BINDING );
          VtlSeq.OctetBinding := TcpSeqNum;
          Next_SendSeqNum = TcpSeqNum + SegLength;
          if(ExpdtFlag = TRUE) Then
                PacketNum := SendExpdtNum := SendExpdtNum +1;
          else
                PacketNum := SendPacketNum := SendPacketNum + 1;
          VtlSeq.PacketBinding := PacketNum;
          UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
      end;

    if (TcpSeqNum < Next_SendSeqNum) then
      begin
          { A retransmission; get the VSN from the SendMapList }

           VsnFromSendMap(VtlSeq, SegLength, ExpdtFlag);
      end;


end;
```

```
procedure       MapRecvVSNtoSeq( var TcpSeqNum : SEQ_NUM_TYPE,
                                 VtlSeq : VSN_TYPE )

begin
                TcpSeqNum := VtlSeq;
end;

procedure       MapAckToVACK(TcpAckNum : SEQ_NUM_TYPE,
                        var VtlAck : VSN_TYPE);

begin
          { Check the receive map list for a VSN which matches this}
          { TcpAckNum, Get the packet number from this List element }
          { fill the VtlAck as :
                VtlAck.BindingType = OCTET_BINDING Logical OR
                                     PACKET_BINIDING;
                VtlAck.OctetBinding = TcpAckNum;
                VtlAck.PacketBinding = PacketNum;                    }
          { If a match is not found then :
                VtlAck.BindingType = OCTET_BINDING;
                VtlAck.OctetBinding = TcpAckNum;                     }
end;


procedure       SetPiggyBackOption(var DataVtpdu : DATA_VTPDU_TYPE)

begin
        DataVtpdu.DataOptions.OptionsFlag := 0x01;
end;

procedure       MapVACKtoAck(var TcpAckNum : SEQ_NUM_TYPE,
                                 VtlAck : VSN_TYPE);

begin
                TcpAckNum = VtlAck.OctetBinding;
end;

procedure       FillVtpduHdr (  Security     : SECURITY_TYPE,
                                Precedence   : PRECEDENCE_TYPE,
                                var VtpduHdr : VTPDU_HEADER_TYPE)

begin
                VtpduHdr.VersionNo := 1;
                VtpduHdr.Use_CheckSum := 1;
                VtpduHdr.RemoteVTLendPt := RemoteVTL_endPt;
```

```
                VtpduHdr.Priority := DEFAULT_PRECEDENCE;
                VtpduHdr.Security := DEFAULT_SECURITY;
end;

procedure       BuildDATAVtpdu (TcpSeg : TCP_SEG_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var DataVtpdu : DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(DataVtpdu.VtpduHdr, Security, Precedence);
          DataVtpdu.VtpduHdr.VtpduCode := DATA_VTPDU;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);

end;

procedure       BuildEDATAVtpdu (TcpSeg : TCP_SEG_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var EDataVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := EDATA_VTPDU;
          EDataVtpdu.TotalEDlen := TcpSeg.UrgPtr;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);

end;

procedure       BuildACKVtpdu (TcpSeg : TCP_SEG_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := DATA_ACK_VTPDU;
          if(IsUrg(TcpSeg) = TRUE) then
            begin
                AckVtpdu.AckOptionsMask := URG_DATA_LENGTH;
                AckVtpdu.AckOptions.UrgentDataSize := TcpSeg.UrgPtr;
            end;
          ComputeVTLCheckSum(Vtpdu);
```

```
end;

trans


    to CLOSE
          begin
                { Add this part to the initialization }
                { Clear recv and send map table       }
                     ClearSendMapList();
                     ClearRecvMapList();
          end


          { transition due to interactions from IP }

    from OPEN to same

    when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                          length : SEGMENT_LENGTH_TYPE,
                                          Security : SECURITY_TYPE,
                                          Precedence : PRECEDENCE_TYPE )

    { A TCP segment can carry the following information :              }
    {
          1. Data
          2. Ack
          3. PUSH
          4. URG
    }


    { A DATA VTPDU is generated if length > 0 and URG flag is not set
      Any ACK information is piggybacked with data                    }

          provided ((length > 0) and NOT IsUrg(TcpSeg)
                                 and NOT IsRst(TcpSeg)
                                 and VerifyTCPChecksum(TcpSeg))
          begin

                DataVtpdu := TcfGetVtpdu(VTL_DT);
                ExpdtFlag := FALSE;
                { Use the Seq Number to get the VSN }
                If(TcpSeg.SeqNum > Next_SendSeqNum) then
                 begin
                      { If this condition happens, then some of the  }
                      { intermediate segments are lost.  It not possible }
                      { to associate a packet count in the VSN }
                     VtlSeq.VsnType := OCTET_BINDING;
```

```
                VtlSeq.OctetBinding := TcpSeqNum;
                UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
                SetIncompleteVsnIndex(PartialVsnIndex);
             end;
          else if (IsMissingSeg(TcpSeq, Vsn) = TRUE) then
             begin
                { build a map information message and send to peer }
                { TCF over COTS interaction point                  }
                UpdateVsnInfo(TcpSeq, Vsn, SegLength, ExpdtFlag);
                MapInfoMsg = BuildMapInfo(TcpSeq, Vsn);
                output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
             end;
          else MapSeqToVSN (TcpSeg.SeqNum, DataVtpdu.SendVSN);
          if(IsAck(TcpSeg) = TRUE) then
             begin
                SetPiggyBackOption(DataVtpdu);
                MapAckToVACK ( TcpSeg.Acknum, DataVtpdu);
                MapCredit (TcpSeg.Credit, DataVtpdu);
             end;
          BuildDATAVtpdu (TcpSeg, length, Security,
                          Precedence, AckVtpdu);
          output TcfToClts.SendTCFdataRequest(DataVtpdu);
    end;


{ An Expedited DATA VTPDU is generated if length > 0 and URG flag  }
{ is set.  Any ACK information is piggybacked                      }

    provided ((length > 0) and IsUrg(TcpSeg)
                           and NOT IsRst(TcpSeg)
                           and VerifyTCPChecksum(TcpSeg))
    begin

          EDataVtpdu = TcfGetVtpdu(VTL_DT);
          ExpdtFlag := TRUE;
          { Use the Seq Number to get the VSN }
          If(TcpSeg.SeqNum > Next_SendSeqNum) then
           begin
                { If this condition happens, then some of the  }
                { intermediate segments are lost.  It not possible }
                { to associate a packet count in the VSN }
                VtlSeq.VsnType := OCTET_BINDING;
                VtlSeq.OctetBinding := TcpSeqNum;
                UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
                SetIncompleteVsnIndex(PartialVsnIndex);
             end;
          else if (IsMissingSeg(TcpSeq, Vsn) = TRUE) then
             begin
                { build a map information message and send to peer }
                { TCF over COTS interaction point                  }
                UpdateVsnInfo(TcpSeq, Vsn, SegLength, ExpdtFlag);
                MapInfoMsg = BuildMapInfo(TcpSeq, Vsn);
```

```
                output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
             end;
          { Use the Seq Number to get the Expedited Data VSN }
          else MapSeqToVSN (TcpSeg.SeqNum, EDataVtpdu.ExpdtSendVSN,
                            ExpdtFlag);
          if(IsAck(TcpSeg) = TRUE) then
             begin
                SetPiggyBackOption(EDataVtpdu);
                MapAckToVACK ( TcpSeg.Acknum, EDataVtpdu);
                MapCredit (TcpSeg.Credit, EDataVtpdu);
             end;
          BuildEDATAVtpdu (TcpSeg, length, Security,
                           Precedence, EDataVtpdu);
          output TcfToClts.SendTCFdataRequest(EDataVtpdu);
    end;


{ An empty ACK Segment is mapped to an ACK VTPDU  }

    provided ((length = 0) and IsAck(TcpSeg)
                           and NOT IsRst(TcpSeg)
                           and VerifyTCPChecksum(TcpSeg))
    begin

          AckVtpdu = TcfGetVtpdu(VTL_ACK);
          {Use the AckNum in the TCP seg to get the VSN }
          MapAckToVACK(TcpSeg.Acknum, AckVtpdu);
          MapCredit (TcpSeg.Credit, DataVtpdu.Credit);
          BuildACKVtpdu (TcpSeg, length, Security,
                         Precedence, AckVtpdu);
          output TcfToClts.SendTCFdataRequest(AckVtpdu);
    end;


{  ******* transition due to interactions from CLTS ********  }


    { The finite set of VTPDUs that can be received are
          1. Data with or without PiggyBack ACK info.
          2. Expedited Data VTPDU with or without Piggy back info
          3. An ACK VTPDU

from OPEN to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

    provided ( IsVTLdata(Vtpdu) = TRUE and
                    VerifyVTLChecksum(Vtpdu) = TRUE)
    begin
          TcpSeg = TcfGetTcpSeg();
          ExpdtFlag := FALSE
```

```
            { If the required VSN binding is available then :}
            if(IsOctetBinding(Vtpdu.SendVSN) = TRUE) then
             begin
                  UpdateRecvMapList(Vtpdu, SegLength, ExpdtFlag);
                  { The encoding of the VSN is such that the }
                  { Octet binding is present, i.e. does not  }
                  { have to be derived                       }
            MapRecvVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
                  If(PiggyBackAck(Vtpdu) = TRUE and
                    (IsOctetBinding(Vtpdu.NextRecvVSN) = TRUE)) then
                   begin
                        MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
                        RecordVACK( Vtpdu.NextRecvVSN,
                              PreviousNextRecvVSN);
                        MapCreditToTcp (TcpSeg.Credit, Vtpdu);
                        RecordCredit (Vtpdu, PreviousCreditValue);
                   end
                  else
                   begin
                        MapVACKtoAck(TcpSeg.AckNum,
                                    PreviousNextRecvVSN);
                        MapCreditToTcp(TcpSeg.Credit,
                                    PreviousCreditValue);
                   end

                  BuildTcpSeg (TcpSeg, Vtpdu);
                  output TcfToIp.SendTPDUrequest(TcpSeg,
                  NORMAL_SECURITY, DEFAULT_PRECEDENCE);
             end; { if OctetBinding }
            else
             begin
                  if ( BufferVtpduPolicy() = TRUE ) then
                    { Buffer VTPDUs with incomplete VSNs }
                       BufferVtpdus(Vtpdu);
                  else { discard the VTPDU }
             end;
      end;


{ Expedited VTPDU is mapped to a TCP segment with URG flag set  }

      provided ( IsVTLexpdtdata(Vtpdu) = TRUE and
                       VerifyVTLChecksum(Vtpdu))
      begin
            TcpSeg = TcfGetTcpSeg();
            MapURGptr(TcpSeg, Vtpdu);
            ExpdtFlag := FALSE
            { If the required VSN binding is available then :}
            if(IsOctetBinding(Vtpdu.SendVSN) = TRUE) then
             begin
                  UpdateRecvMapList(Vtpdu, SegLength, ExpdtFlag);
                  { The encoding of the VSN is such that the }
```

```
                    { Octet binding is present, i.e. does not  }
                    { have to be derived                       }
              MapRecvVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
                    If(PiggyBackAck(Vtpdu) = TRUE and
                      (IsOctetBinding(Vtpdu.NextRecvVSN) = TRUE)) then
                     begin
                          MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
                          RecordVACK( Vtpdu.NextRecvVSN,
                                PreviousNextRecvVSN);
                          MapCreditToTcp (TcpSeg.Credit, Vtpdu);
                          RecordCredit (Vtpdu, PreviousCreditValue);
                     end
                    else
                     begin
                          MapVACKtoAck(TcpSeg.AckNum,
                                       PreviousNextRecvVSN);
                          MapCreditToTcp(TcpSeg.Credit,
                                       PreviousCreditValue);
                     end

                    BuildTcpSeg (TcpSeg, Vtpdu);
                    output TcfToIp.SendTPDUrequest(TcpSeg,
                    NORMAL_SECURITY, DEFAULT_PRECEDENCE);
               end; { if OctetBinding }
              else
               begin
                    if ( BufferVtpduPolicy() = TRUE ) then
                      { Buffer VTPDUs with incomplete VSNs }
                         BufferVtpdus(Vtpdu);
                    else { discard the VTPDU }
               end;

          end;


          provided ( IsVTLack(Vtpdu) = TRUE and
                          VerifyVTLChecksum(Vtpdu) = TRUE)
          begin
               TcpSeg = TcfGetTcpSeg();
               MapVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
               MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
               MapCreditToTcp (TcpSeg.Credit, Vtpdu);
               BuildTcpSeg (TcpSeg, Vtpdu);
               output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                              DEFAULT_PRECEDENCE);
          end;


{  ******* transition due to interactions from COTS ********  }
```

```
{ Peer TCF messages arive over the Connection Oriented Transport }
{ between the gateways                                            }

   { the message types are :

              1. VSN Mapping Information

   }


   from OPEN to same

   when  CotsToTcf.ReceiveTCFMsgIndication (TcfMsg : TCF_MSG_TYPE)

         provided ( IsVsnMap(TcfMsg) = TRUE )

         begin
               { The mapping information can be used if the  }
               { received VTPDUs with missing Octet binding  }
               { were buffered                               }
               { If so then - The RecvMapList is updated     }
               {            - The corresponding VTPDUs from  }
               {              the recv buffer are mapped to  }
               {              local TPDUs                    }

               If (BufferPolicy() = TRUE) then
                begin
                     FillMissingVsn(TcfMsg);
                     do
                       Vtpdu = GetBufferedVtpdu();
                       TcpSeg = GetTcpSeg();
                       GenerateTcpSeg(TcpSeg, Vtpdu);
                        output TcfToIp.SendTPDUrequest(TcpSeg,
                                   NORMAL_SECURITY, DEFAULT_PRECEDENCE);
                      while (MoreVtpdus();
                 end;
         end;


end; { of TCP TCF body }
```

type

```
VSN_TYPE =
      record
            BindingType       : ONE_BYTE;
            OctetBinding      : FOUR_BYTES;
            PacketBinding     : TWO_BYTES;
      end

OCTET_BINDING  = 0x01;
PACKET_BINDING = 0x02;

PIGGYBACK_ACK_TYPE =
      record
            NextRecvVSN       : VSN_TYPE;
            Credit            : CREDIT_TYPE;
      end;

DATA_OPTIONS_TYPE =
      record
            OptionsFlags      : ONE_BYTE;
            PiggyBackAck      : PIGGYBACK_ACK_TYPE;
      end;

{ The encoding of the OptionsFlag in the data options is as : }

PIGGYBACK_ACK = 0x01;

VTPDU_HEADER_TYPE =
      record
            VersionNo         : VERSION_TYPE;
            VtpduCode         : VTPDU_CODE_TYPE;
            Use_CheckSum      : 1_BIT;
            RemoteVTLendPt    : END_POINT_TYPE;
            Priority          : array[1..2] of 1_BIT;
            Security          : array[1..2] of 1_BIT;
            Reserved          : array [1..4] of 1_BIT;
      end;


DATA_TPDU_TYPE  =
      record
            VtpduHdr          : VTPDU_HEADER_TYPE;
            SendVSN           : VSN_TYPE;
```

```
            OptionsFlags        : ONE_BYTE;
            DataOptions         : DATA_OPTIONS_TYPE;
            Data                : DATA_TYPE;
            CheckSum            : TWO_BYTES;
      end;


{ ACK_OPTIONS is a variant record, with Mask field describing which }
{ of the options are carried in the VTPDU                           }

 ACK_OPTIONS_TYPE =
      record
            OptionsMask         : ONE_BYTE;
            UrgentDataSize      : TWO_BYTES;
            AckSequence         : TWO_BYTES;
            FccInfo             : array[1..8] of ONE_OCTET;
      end;

 { The encoding of the ACK OptionsMask is as :                          }


 URG_DATA_LENGTH    = 0x01;
 ACK_SEQ            = 0x02;
 FCC_REQ            = 0x04;
 EXPDT_ACK          = 0x08;


 DATA_ACK_VTPDU_TYPE  =
      record
            VtpduHdr            : VTPDU_HEADER_TYPE;
            NextRecv            : VSN_TYPE;
            Credit              : CREDIT_TYPE;
            AckOptions          : ACK_OPTIONS_TYPE;
            CheckSum            : TWO_BYTES;
      end;


EXPEDITED_DATA_TPDU_TYPE  =
      record
            VtpduHdr            : VTPDU_HEADER_TYPE;
            ExpdtSendSeq        : VSN_TYPE;
            TotalEDlen          : DATA_LENGTH_TYPE;
            DataOptions         : DATA_OPTIONS_TYPE;
            Data                : DATA_TYPE;
            CheckSum            : TWO_BYTES;
      end;


RST_REQ_VTPDU_TYPE  = ...;          { defined in appendix 3 }
```

```
{ TP4 Sequence number to VSN mapping table element                    }

SEND_MAP_TABLE_ELEMENT_TYPE =
      record
            Vsn          :  VSN_TYPE;
            DataLength   :  DATA_LENGTH_TYPE;
            PacketType   :  1_BIT;          { Set to 1 if mapping is   }
      end;                                  { done to/from EXPDT VTPDU }


RECV_MAP_TABLE_ELEMENT_TYPE =
      record
            Vsn          :  VSN_TYPE;
            DataLength   :  DATA_LENGTH_TYPE;
            ExpectedAck  :  TP4_SEQ_TYPE;   { packet binding + 1 }
            PacketType   :  1_BIT;          { Set to 1 if mapping is   }
      end;                                  { done to/from EXPDT VTPDU }

var

    DataVtpdu          :   DATA_VTPDU_TYPE;
    DataAckVtpdu       :   DATA_ACK_VTPDU_TYPE;
    ExpdtDataVtpdu     :   EXPEDITED_DATA_VTPDU_TYPE;
    ExpdtAckVtpdu      :   EXPEDITED_ACK_VTPDU_TYPE;
    RejVtpdu           :   RST_REQ_VTPDU_TYPE;

    Tp4Data            :   ISO_TP4_DT_TYPE;
    Tp4EData           :   ISO_TP4_ED_TYPE;
    Tp4AK              :   ISO_TP4_AK_TYPE;
    Tp4EA              :   ISO_TP4_EA_TYPE;

    MappingInfoMsg     :   TCF_MSG_TYPE;
    SendListElement    :   SEND_MAP_TABLE_ELEMENT_TYPE;
    RecvListElement    :   RECV_MAP_TABLE_ELEMENT_TYPE;
```

{ **Channel Definitions for communication with the TCF Module** }

```
Channel   IP_Access_Point  ( From_IP,  To_IP );

    by    From_IP   : ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE );

    by    To_IP     : SendTPDUrequest ( Tp4Tpdu : TP4_TPDU_TYPE );

                      ReceiveTPDUrequest ;


Channel   CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
```

```
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF  :   ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );

Channel  COTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFMsgRequest;
                    SendTCFMsgRequest ( TcfMsg : TCF_MESSAGE_TYPE );

    by  To_TCF  :  ReceiveTCFMsgIndication ( TcfMsg : TCF_MESSAGE_TYPE );
```

{ **Module Header Definitions** }

```
Module InternetProtocol_Type  process ;

    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;

Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
          TcfToIp      : IP_Access_Point ( To_IP) individual queue;
          TcfToClts  : CLTS_Access_Point ( From_TCF)
                                              individual  queue ;
          TcfToCots  : COTS_Access_Point ( From_TCF)
                                              individual  queue ;
end;


Module CLTS_Type process;

    ip    CltsToTcf  : CLTS_ACCESS_POINT ( To_TCF )

end;


Module COTS_Type process;

    ip    CotsToTcf  : COTS_ACCESS_POINT ( To_TCF )

end;
```

{ **Body Definitions for Modules** }

```
Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body  for CLTS_Type; external;

Body      Cots_Body  for COTS_Type; external;


Body TCF_Body for TCF_Type

var

    SendMapList  : ..;       { Actual data structure is left as }
                             { implementation dependent         }
    SendMapListIndex : ...;

    PartialVsnIndex : ...;   { points to List Element having
                               incomplete information}

    RecvMapList  : ..;       { Actual data structure is left as }
                             { implementation dependent         }

    SendPacketSize : DATA_LENGTH_TYPE;

    RecvPacketSize : DATA_LENGTH_TYPE;

    SendPacketNum      : TP4_SEQ_TYPE;   { current TP4 send seq }

    Next_SendPacketNum : TP4_SEQ_TYPE: { The next expected send seq that
                                         will be carried by the TP4 TPDU }
    NextSendSeqNum  : FOUR_BYTES;

    RecvSeqNum  : FOUR_BYTES;

    Next_RecvSeqNum      : FOUR_BYTES;

    RecvPacketNum  : TP4_SEQ_TYPE;


     { Functions and procedures used in the module body }

function       TcfGetVtpdu (Vtpdu_Type : VTPDU_TYPE) : VTPDU_TYPE;
primitive  ;


function       TcfGetTp4TPDU (TpduType : TP4_TPDU_CODE) : TP4_TPDU_TYPE;
primitive  ;


procedure      ClearSendMapList();
```

```
primitive  ;

procedure       ClearRecvMapList();
primitive  ;

procedure       UpdateSendMapList(VtlSeq : VSN_TYPE,
                                  Length : DATA_LENGTH_TYPE,
                                  ExpdtFlag : char);

          { allocate a SendListElement and add to head of FIFO send queue }
primitive  ;


procedure       UpdateRecvMapList(Vtpdu : VTPDU_TYPE,
                                  ExpdtFlag : char);

          { If (Vsn < Next_RecvVsn ) Then it is a retransmission.
               check the Recvmaplist to see if the packet binding was
               present.  If not then free the buffers in the recv buffer
               list.  Fill in the packet binding info in the RecvMapList }
          { Else Allocate a RecvListElement and add to head of FIFO
            recv queue }
primitive  ;


procedure       SetIncompleteVsnIndex( var PartialVsnIndex);
primitive  ;


function        BufferVtpduPolicy()   : boolean;
primitive  ;

procedure       BufferVtpdus(DataVtpdu : DATA_VTPDU_TYPE);
primitive  ;

function        IsPacketBinding(Vsn : VSN_TYPE)   : boolean;
primitive  ;

function        IsMissingSeg(Tp4Seq : TP4_SEQ_TYPE,
                             Vsn : VSN_TYPE ) : boolean ;

                {
                      check the SendMapList for this Tp4SeqNum;
                      check if VSN entry is incomplete
                            return true if it is
                       else return false
                }
 primitive  ;
```

```
procedure       UpdateVsnInfo(Tp4Seq, Vsn, Length, ExpdtFlag);

              { Update the VSN map table with the missing octet number
                portion. }
primitive  ;



function        BuildMapInfo(Tp4Seq : TP4_SEQ_TYPE,
                             Vsn : VSN_TYPE)   : TCF_MESSAGE_TYPE,
begin
           { the VSN Map Message has the following format :
                Remote_End_Point_Id,
                Number of VSNs, < VSN1, ...., VSNp>
           }
           return (MapMsg);
end;



procedure       MapSeqToVSN(Tp4PacketNum : SEQ_NUM_TYPE,
                            Length : DATA_LENGTH_TYPE,
                            var VtlSeq : VSN_TYPE,
                            ExpdtFlag : char);

begin

var PacketNum;


     if (Tp4PacketNum = Next_SendPacketNum ) then
       begin
           { This is the next in sequence packet}
           VtlSeq.VsnType := (OCTET_BINDING Bit OR
                              PACKET_BINDING );
           VtlSeq.PacketBinding := Tp4PacketNum;
           VtlSeq.OctetBinding := NextSendSeq;
           UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
           NextSendSeq := NextSendSeq + Length;
        end;
     if (Tp4PacketNum < Next_SendPacketNum) then
       begin
           { A retransmission; get the VSN from the SendMapList }

            VsnFromSendMap(VtlSeq, Length, ExpdtFlag);
       end;


end;
```

```
procedure       MapRecvVSNtoSeq( var Tp4PacketNum : TP4_SEQ_TYPE,
                                 VtlSeq : VSN_TYPE )

begin
                Tp4PacketNum := VtlSeq.PacketBinding;
end;


procedure       MapAckToVACK(TcpAckNum : TP4_SEQ_TYPE,
                        var VtlAck : VSN_TYPE);

begin
          { Check the receive map list for a VSN which matches this}
          { Tp4AckNum, Get the octet number from this List element }
          { fill the VtlAck as :
                VtlAck.BindingType := OCTET_BINDING  Bit OR
                                      PACKET_BINIDING;
                VtlAck.OctetBinding := RecvListElement.Vsn.OctetBinding +
                                       RecvListElement.DataLength;
                VtlAck.PacketBinding := Tp4AckNum;                    }
          { If a match is not found then :
                VtlAck.BindingType = PACKET_BINDING;
                VtlAck.PacketBinding = Tp4AckNum;                     }

end;


procedure       MapVACKtoAck(var Tp4AckNum : TP4_SEQ_TYPE,
                                VtlAck : VSN_TYPE);

begin
                Tp4AckNum = VtlAck.PacketBinding;

end;


procedure       FillVtpduHdr (  Security     : SECURITY_TYPE,
                                Precedence   : PRECEDENCE_TYPE,
                                var VtpduHdr : VTPDU_HEADER_TYPE)

begin
                VtpduHdr.VersionNo := 1;
                VtpduHdr.Use_CheckSum := 1;
                VtpduHdr.RemoteVTLendPt := RemoteVTL_endPt;
                VtpduHdr.Priority := DEFAULT_PRECEDENCE;
                VtpduHdr.Security := DEFAULT_SECURITY;
end;


procedure       BuildDATAVtpdu (Tp4Data : ISO_TP4_DT_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
```

```
                                Precedence : PRECEDENCE_TYPE,
                                var DataVtpdu : DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(DataVtpdu.VtpduHdr, Security, Precedence);
          DataVtpdu.VtpduHdr.VtpduCode := DATA_VTPDU;
          CopyUserData(Tp4Data.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildEDATAVtpdu (Tp4EData : ISO_TP4_ED_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var EDataVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := EDATA_VTPDU;
          EDataVtpdu.TotalEDlen := Length;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildACKVtpdu (Tp4Ack : ISO_TP4_AK_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := DATA_ACK_VTPDU;
          { If FCC information is present in the Tp4Ack, then fill in the
            optional fields of the AckVtpdu    }
          ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildEACKVtpdu (Tp4EAck : ISO_TP4_EA_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
```

```
        EDataVtpdu.VtpduHdr.VtpduCode := EDATA_ACK_VTPDU;
        { Mark Expedited Ack option in the ACK VTPDU }
        ComputeVTLCheckSum(Vtpdu);

end;


trans


    to CLOSE
        begin
            { Add this part to the initialization }
            { Clear recv and send map table       }
                ClearSendMapList();
                ClearRecvMapList();
        end


        { transition due to interactions from IP }

    from OPEN to same

    when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : ISO_TP4_TPDU_TYPE)

    { A TP4 TPDU can carry the following information :              }
    {
        1. Data
        2. Ack
        3. Expdt Data
        4. Expdt Ack
        5. RST
    }

        provided (IsData(Tp4Tpdu) and VerifyTp4Checksum(TcpSeg))
        begin

            DataVtpdu := TcfGetVtpdu(VTL_DT);
            ExpdtFlag := FALSE;
            { Use the Packet Number to get the VSN }
            If(Tp4Tpdu.PacketNum > Next_SendPacketNum) then
             begin
                { If this condition happens, then some of the  }
                { intermediate packets are lost.  It not possible }
                { to associate a octet count in the VSN }
                VtlSeq.VsnType := PACKET_BINDING;
                VtlSeq.PacketBinding := Tp4Tpdu.PacketNum;
                UpdateSendMapList(VtlSeq, Tp4Tpdu.DataLength,
                                  ExpdtFlag);
                SetIncompleteVsnIndex(PartialVsnIndex);
              end;
```

```
      else if (IsMissingSeg(Tp4Tpdu.PacketNum, Vsn) = TRUE) then
         begin
            { build a map information message and send to peer }
            { TCF over COTS interaction point                  }
            UpdateVsnInfo(Tp4Tpdu.PacketNum, Vsn, DataLength,
                          ExpdtFlag);
            MapInfoMsg = BuildMapInfo(Tp4Tpdu.PacketNum, Vsn);
            output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
         end;
      else MapSeqToVSN (Tp4tpdu.PacketNum, DataVtpdu.SendVSN,
                        ExpdtFlag);
      BuildDATAVtpdu (TcpSeg, length, Security,
                      Precedence, DataVtpdu);
      output TcfToClts.SendTCFdataRequest(DataVtpdu);
end;


provided (IsEData(Tp4Tpdu) and VerifyTp4Checksum(TcpSeg))
begin

      EDataVtpdu = TcfGetVtpdu(VTL_DT);
      ExpdtFlag := TRUE;
      { Use the Packet Number to get the VSN }
      If(Tp4Tpdu.PacketNum > Next_SendPacketNum) then
       begin
            { If this condition happens, then some of the  }
            { intermediate packets are lost.  It not possible }
            { to associate a octet count in the VSN }
            VtlSeq.VsnType := PACKET_BINDING;
            VtlSeq.PacketBinding := Tp4Tpdu.PacketNum;
            UpdateSendMapList(VtlSeq, Tp4Tpdu.DataLength,
                              ExpdtFlag);
            SetIncompleteVsnIndex(PartialVsnIndex);
        end;
      else if (IsMissingSeg(Tp4Tpdu.PacketNum, Vsn) = TRUE) then
         begin
            { build a map information message and send to peer }
            { TCF over COTS interaction point                  }
            UpdateVsnInfo(Tp4Tpdu.PacketNum, Vsn, DataLength,
                          ExpdtFlag);
            MapInfoMsg = BuildMapInfo(Tp4Tpdu.PacketNum, Vsn);
            output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
         end;
      else MapSeqToVSN (Tp4tpdu.PacketNum, DataVtpdu.SendVSN,
                        ExpdtFlag);
      BuildEDATAVtpdu (TcpSeg, length, Security,
                       Precedence, DataVtpdu);
      output TcfToClts.SendTCFdataRequest(DataVtpdu);
end;


provided (IsAck(Tp4Tpdu) and VerifyTp4Checksum(TcpSeg))
```

```
        begin

              AckVtpdu = TcfGetVtpdu(VTL_ACK);
              {Use the NextSend in the Tp4 Ack to get the VSN }
              MapAckToVACK(Tp4Tp, AckVtpdu);
              MapCredit (Tp4Tpdu.Credit, AckVtpdu.Credit);
              BuildACKVtpdu (Tp4Tpdu, Security,
                             Precedence, AckVtpdu);
              output TcfToClts.SendTCFdataRequest(AckVtpdu);
        end;


        provided (IsEAck(Tp4Tpdu) and VerifyTp4Checksum(TcpSeg))
        begin

              AckVtpdu = TcfGetVtpdu(VTL_ACK);
              {Use the NextSend in the Tp4 Ack to get the VSN }
              MapAckToVACK(Tp4Tpdu, AckVtpdu);
              MapCredit (Tp4Tpdu.Credit, AckVtpdu.Credit);
              BuildEACKVtpdu (Tp4Tpdu, Security,
                             Precedence, AckVtpdu);
              output TcfToClts.SendTCFdataRequest(AckVtpdu);
        end;



{  ******* transition due to interactions from CLTS ********  }


        { The finite set of VTPDUs that can be received are
              1. Data with or without PiggyBack ACK info.
              2. Expedited Data VTPDU with or without Piggyback info.
              3. An ACK VTPDU with or without the Expedited Option

    from OPEN to same

    when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

        provided ( IsVTLdata(Vtpdu) = TRUE and
                        VerifyVTLChecksum(Vtpdu) = TRUE)
        begin
              Tp4Data := TcfGetTp4Tpdu(DT);
              ExpdtFlag := FALSE;
              { If the required VSN binding is available then :}
              if(IsPacketBinding(Vtpdu.SendVSN) = TRUE) then
               begin
                    UpdateRecvMapList(Vtpdu, ExpdtFlag);
                    { The encoding of the VSN is such that the }
                    { Packet binding is present, i.e. does not  }
```

```
            { have to be derived                          }
        MapRecvVSNtoSeq(Tp4Data.SendSeq, Vtpdu.SendVSN);
            If(PiggyBackAck(Vtpdu) = TRUE and
              (IsPacketBinding(Vtpdu.NextRecvVSN) = TRUE)) then
             begin
                  If(IsExpdtAck(Vtpdu) then
                   begin
                        Tp4EAk := TcfGetTp4Tpdu(EA);
                        MapVACKtoAck(Tp4EAk.NextRecv,
                                     Vtpdu.NextRecvVSN);
                        BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                        output TcfToIp.SendTPDUrequest(TpE4Ak);
                   end;
                  else   { normal data ACK }
                    begin
                        Tp4Ak := TcfGetTp4Tpdu(AK);
                        MapVACKtoAck(Tp4Ak.NextRecv,
                               Vtpdu.NextRecvVSN);
                        MapCreditToTp4 (Tp4Ak.Credit,Vtpdu);
                        BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                        output TcfToIp.SendTPDUrequest(Tp4Ak);
                   end;
             end; { piggyback ACK }
            BuildTp4DTPDU (Tp4Data, Vtpdu);
            output TcfToIp.SendTPDUrequest(Tp4Data);
end; { if OctetBinding }
     else
      begin
            if ( BufferVtpduPolicy() = TRUE ) then
              { Buffer VTPDUs with incomplete VSNs }
                BufferVtpdus(Vtpdu);
      { see if the piggy back ACK, if preset, has packet binding }
            If(PiggyBackAck(Vtpdu) = TRUE and
              (IsPacketBinding(Vtpdu.NextRecvVSN) = TRUE)) then
             begin
                  If(IsExpdtAck(Vtpdu) then
                   begin
                        Tp4EAk := TcfGetTp4Tpdu(EA);
                        MapVACKtoAck(Tp4EAk.NextRecv,
                                     Vtpdu.NextRecvVSN);
                        BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                        output TcfToIp.SendTPDUrequest(TpE4Ak);
                   end;
                  else   { normal data ACK }
                    begin
                        Tp4Ak := TcfGetTp4Tpdu(AK);
                        MapVACKtoAck(Tp4Ak.NextRecv,
                               Vtpdu.NextRecvVSN);
                        MapCreditToTp4 (Tp4Ak.Credit,Vtpdu);
                        BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                        output TcfToIp.SendTPDUrequest(Tp4Ak);
                   end;
```

```
                end; { piggybacked ACK }
              else { discard the VTPDU }
        end;
  end;


{ Expedited VTPDU is mapped to a TP4 Expedited Data TPDU  }

  provided ( IsVTLexpdtdata(Vtpdu) = TRUE and
                VerifyVTLChecksum(Vtpdu))
  begin
        Tp4EData := TcfGetTp4Tpdu(ED);
        ExpdtFlag := TRUE;
        { If the required VSN binding is available then :}
        if(IsPacketBinding(Vtpdu.SendVSN) = TRUE) then
         begin
              UpdateRecvMapList(Vtpdu, ExpdtFlag);
              { The encoding of the VSN is such that the }
              { Packet binding is present, i.e. does not  }
              { have to be derived                        }
        MapRecvVSNtoSeq(Tp4EData.SendSeq, Vtpdu.SendVSN);
              If(PiggyBackAck(Vtpdu) = TRUE and
                (IsPacketBinding(Vtpdu.NextRecvVSN) = TRUE)) then
               begin
                    If(IsExpdtAck(Vtpdu) then
                     begin
                          Tp4EAk := TcfGetTp4Tpdu(EA);
                          MapVACKtoAck(Tp4EAk.NextRecv,
                                       Vtpdu.NextRecvVSN);
                          BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                          output TcfToIp.SendTPDUrequest(TpE4Ak);
                     end;
                    else   { normal data ACK }
                      begin
                          Tp4Ak := TcfGetTp4Tpdu(AK);
                          MapVACKtoAck(Tp4Ak.NextRecv,
                                 Vtpdu.NextRecvVSN);
                          MapCreditToTp4 (Tp4Ak.Credit,Vtpdu);
                          BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                          output TcfToIp.SendTPDUrequest(Tp4Ak);
                     end;
               end; { piggybacked ACK }
              BuildTp4DTPDU (Tp4Data, Vtpdu);
              output TcfToIp.SendTPDUrequest(Tp4EData);
  end; { if OctetBinding }
       else
        begin
              if ( BufferVtpduPolicy() = TRUE ) then
                { Buffer VTPDUs with incomplete VSNs }
                  BufferVtpdus(Vtpdu);
        { see if the piggy back ACK, if preset, has packet binding }
              If(PiggyBackAck(Vtpdu) = TRUE and
```

```
            (IsPacketBinding(Vtpdu.NextRecvVSN) = TRUE)) then
           begin
                If(IsExpdtAck(Vtpdu) then
                 begin
                      Tp4EAk := TcfGetTp4Tpdu(EA);
                      MapVACKtoAck(Tp4EAk.NextRecv,
                                   Vtpdu.NextRecvVSN);
                      BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                      output TcfToIp.SendTPDUrequest(TpE4Ak);
                 end;
                else   { normal data ACK }
                  begin
                      Tp4Ak := TcfGetTp4Tpdu(AK);
                      MapVACKtoAck(Tp4Ak.NextRecv,
                             Vtpdu.NextRecvVSN);
                      MapCreditToTp4 (Tp4Ak.Credit,Vtpdu);
                      BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                      output TcfToIp.SendTPDUrequest(Tp4Ak);
                 end;
           end; { piggybacked ACK }
          else { discard the VTPDU }
      end;

end;


provided ( IsVTLack(Vtpdu) = TRUE and
                VerifyVTLChecksum(Vtpdu) = TRUE)
begin
      if(IsPacketBinding(Vtpdu.NextRecvVSN) = TRUE)) then
       begin
            If(IsExpdtAck(Vtpdu) then
             begin
                  Tp4EAk := TcfGetTp4Tpdu(EA);
                  MapVACKtoAck(Tp4EAk.NextRecv,
                                   Vtpdu.NextRecvVSN);
                  BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                  output TcfToIp.SendTPDUrequest(TpE4Ak);
             end;
            else   { normal data ACK }
             begin
                  Tp4Ak := TcfGetTp4Tpdu(AK);
                  MapVACKtoAck(Tp4Ak.NextRecv,
                               Vtpdu.NextRecvVSN);
                  MapCreditToTp4 (Tp4Ak.Credit,Vtpdu);
                  BuildTp4EAKTPDU (Tp4EAk, Vtpdu);
                  output TcfToIp.SendTPDUrequest(Tp4Ak);
             end;
      end;
```

```
{  ******* transition due to interactions from COTS ********  }

 { Peer TCF messages arive over the Connectio Oriented Transport }
 { between the gateways                                           }

    { the message types are :
                1. VSN Mapping Information
    }


    from OPEN to same

    when  CotsToTcf.ReceiveTCFMsgIndication (TcfMsg : TCF_MSG_TYPE)

          provided ( IsVsnMap(TcfMsg) = TRUE )

          begin
                { The mapping information can be used if the  }
                { received VTPDUs with missing packet binding  }
                { were buffered                                }
                { If so then - The RecvMapList is updated      }
                {            - The corresponding VTPDUs from   }
                {              the recv buffer are mapped to   }
                {              local TPDUs                     }

               If (BufferPolicy() = TRUE) then
                begin
                     FillMissingVsn(TcfMsg);
                     do
                       Vtpdu = GetBufferedVtpdu();
                       Tp4Tpdu = GenerateTp4Tpdu(Vtpdu);
                       output TcfToIp.SendTPDUrequest(Tp4Tpdu);
                     while (MoreVtpdus() );
                 end;
          end;


end; { end of TP4 TCF body }
```

12 APPENDIX C. CONNECTION TERMINATION PHASE

TCP_TCF Data Transfer Phase Specification

type

```
DISC_DATA_OPTIONS_TYPE =
      record
            OptionsFlag       : ONE_BYTE;
            PiggyBackAck      : PIGGY_BACK_ACK_TYPE;
            TotalEDlen        : DATA_LENGTH_TYPE;
      end;

{ The encoding of the Options Flag is as : }

PIGGYBACK_ACK     = 0x01;
EXPDT_DATA        = 0x02;
PUSH_DATA         = 0x04;

DISCONNECT_VTPDU_TYPE  =
      record
            VtpduHdr          : VTPDU_HEADER_TYPE;
            SendVSN           : VSN_TYPE;
            Reason            : TWO_BYTES;
            DataOptions       : DISC_DATA_OPTIONS_TYPE;
            Data              : DATA_TYPE;
            CheckSum          : TWO_BYTES;
      end;

RESET_REQUEST_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
RESET_CONFIRM_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
```

```
    RESET_CONFIRM_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;

    DISCONNECT_REQUEST_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
    DISCONNECT_CONFIRM_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;

var

    DataVtpdu          :   DATA_VTPDU_TYPE;
    DataAckVtpdu       :   DATA_ACK_VTPDU_TYPE;
    ExpdtDataVtpdu     :   EXPEDITED_DATA_VTPDU_TYPE;
    ExpdtAckVtpdu      :   EXPEDITED_ACK_VTPDU_TYPE;

    RstReqVtpdu        :   RESET_REQUEST_VTPDU_TYPE;
    RstCnfVtpdu        :   RESET_CONFIRM_VTPDU_TYPE;
    DiscReqVtpdu       :   DISCONNECT_REQUEST_VTPDU_TYPE;
    DiscCnfVtpdu       :   DISCONNECT_CONFIRM_VTPDU_TYPE;

    TcpSeg             :   TCP_SEMENT_TYPE;

    MappingInfoMsg     :   TCF_MSG_TYPE;
    SendListElement    :   SEND_MAP_TABLE_ELEMENT_TYPE;
    RecvListElement    :   RECV_MAP_TABLE_ELEMENT_TYPE;


{ Channel Definitions for communication with the TCF Module }

Channel   IP_Access_Point  ( From_IP,  To_IP );

    by   From_IP  : ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                            Length : SEGMENT_LENGTH_TYPE,
                                            Security : SECURITY_TYPE,
                                            Precedence : PRECEDENCE_TYPE );

    by  To_IP   : SendTPDUrequest ( TcpSeg : TCP_SEGMENT_TYPE,
                                    Security : SECURITY_TYPE,
                                    Precedence : PRECEDENCE_TYPE );
                  ReceiveTPDUrequest ;


Channel   CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF   :  ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );


Channel   COTS_Access_Point ( From_TCF,  To_TCF );
```

```
    by  From_TCF :  ReceiveTCFMsgRequest;
                    SendTCFMsgRequest ( TcfMsg : TCF_MESSAGE_TYPE );

    by  To_TCF   :  ReceiveTCFMsgIndication ( TcfMsg : TCF_MESSAGE_TYPE );
```

{ **Module Header Definitions** }

```
Module InternetProtocol_Type  process ;

    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;


Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
          TcfToIp     : IP_Access_Point ( To_IP) individual queue;
          TcfToClts   : CLTS_Access_Point ( From_TCF) individual queue ;
          TcfToCots   : COTS_Access_Point ( From_TCF) individual queue ;
end;


Module CLTS_Type process;

    ip    CltsToTcf  : CLTS_ACCESS_POINT ( To_TCF )

end;


Module COTS_Type process;

    ip    CotsToTcf  : COTS_ACCESS_POINT ( To_TCF )

end;
```

{ **Body Definitions for Modules** }

```
Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body  for CLTS_Type; external;

Body      Cots_Body  for COTS_Type; external;
```

```
var

state

    ANY : (CLOSE, OPEN, WVTLDC, WFIN, WLACK, WLVTLACK);

    ACTIVE_STATE :  (OPEN, WVTLDC, WFIN, WLACK, WLVTLACK);

    { Where :
      WVTLDC : Wait for DISCONNECT_CONFIRM VTPDU
      WFIN : Wait for TCP FIN
      WLACK : Wait for TCP Last Ack
      WLVTLACK : Wait for VTL last Ack
    }

     { Functions and procedures used in the module body }

         { Support functions and procedures are as     }
         { specified in the Connection Establishment   }
         { phase and Data Transfer phase               }

procedure      BuildDATAVtpdu (TcpSeg : TCP_SEG_TYPE,
                               Length : DATA_LENGTH_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var DataVtpdu : DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(DataVtpdu.VtpduHdr, Security, Precedence);
          DataVtpdu.VtpduHdr.VtpduCode := DATA_VTPDU;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildEDATAVtpdu (TcpSeg : TCP_SEG_TYPE,
                               Length : DATA_LENGTH_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var EDataVtpdu:EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := EDATA_VTPDU;
          EDataVtpdu.TotalEDlen := TcpSeg.UrgPtr;
          CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
          ComputeVTLCheckSum(Vtpdu);
```

```
end;

procedure       BuildACKVtpdu (TcpSeg : TCP_SEG_TYPE,
                               Length : DATA_LENGTH_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := DATA_ACK_VTPDU;
          if(IsUrg(TcpSeg) = TRUE) then
            begin
                AckVtpdu.AckOptionsMask := URG_DATA_LENGTH;
                AckVtpdu.AckOptions.UrgentDataSize := TcpSeg.UrgPtr;
            end;
          ComputeVTLCheckSum(Vtpdu);

end;

trans


    to CLOSE
          begin

          end

          { transition due to interactions from IP }

    from ACTIVE_STATE to same

    when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                          length : SEGMENT_LENGTH_TYPE,
                                          Security : SECURITY_TYPE,
                                          Precedence : PRECEDENCE_TYPE )


          provided ( IsRst(TcpSeg)
                     and VerifyTCPChecksum(TcpSeg))
          begin

                { A RST segment doesnot carry any data, however it can }
                { can carry Ack Num                                    }

                RstReqVtpdu := TcfGetVtpdu(VTL_RST_REQ);
                ExpdtFlag := FALSE;
```

```
            MapSeqToVSN (TcpSeg.SeqNum, RstReqVtpdu.SendVSN);
            if(IsAck(TcpSeg) = TRUE) then
               begin
                  SetPiggyBackOption(RstReqVtpdu);
                  MapAckToVACK ( TcpSeg.Acknum, RstReqVtpdu);
                  MapCredit (TcpSeg.Credit, RstReqVtpdu);
               end;
            BuildRstReqVtpdu (TcpSeg, length, Security,
                              Precedence, RstReqVtpdu);
            output TcfToClts.SendTCFdataRequest(RstReqVtpdu);
      end;


from OPEN to WVTLDC

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

      provided ( IsFin(TcpSeg)
                 and VerifyTCPChecksum(TcpSeg))

{ A FIN segment is mapped to a DiscReqVtpdu and state is changed to }
{ WVTLDC - Wait For VTL Disconnect Confirm.                          }
{ A FIN segment may carry normal, URG or PUSHed data                 }

      begin

            DiscReqVtpdu := TcfGetVtpdu(VTL_DISC_REQ);

            if (IsUrg(TcpSeg)) then
             begin
                  ExpdtFlag := TRUE;
                  SetExpdtOption(DiscReqVtpdu);
             end

            if (IsPush(TcpSeg)) then SetPushOption(DiscReqVtpdu);

            { Use the Seq Number to get the VSN }
            If(TcpSeg.SeqNum > Next_SendSeqNum) then
             begin
                  { If this condition happens, then some of the  }
                  { intermediate segments are lost.  It not possible }
                  { to associate a packet count in the VSN }
                  VtlSeq.VsnType := OCTET_BINDING;
                  VtlSeq.OctetBinding := TcpSeqNum;
                  UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
                  SetIncompleteVsnIndex(PartialVsnIndex);
              end;
            else if (IsMissingSeg(TcpSeq, Vsn) = TRUE) then
               begin
```

```
                { TCF over COTS interaction point                    }
                UpdateVsnInfo(TcpSeq, Vsn, SegLength, ExpdtFlag);
                MapInfoMsg = BuildMapInfo(TcpSeq, Vsn);
                output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
             end;
          { Use the Seq Number to get the Expedited Data VSN }
          else MapSeqToVSN (TcpSeg.SeqNum, DiscReqVtpdu.SendVSN,
                            ExpdtFlag);
          if(IsAck(TcpSeg) = TRUE) then
             begin
                SetPiggyBackOption(DiscReqVtpduVtpdu);
                MapAckToVACK ( TcpSeg.Acknum, DiscReqVtpdu);
                MapCredit (TcpSeg.Credit, DiscReqVtpdu);
             end;
          BuildDiscReqVtpdu (TcpSeg, length, Security,
                             Precedence, DiscReqVtpdu);
          output TcfToClts.SendTCFdataRequest(DiscReqVtpdu);
    end;

from WVTLDC to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE)

      provided ( NOT IsRst(TcpSeg)
                 and VerifyTCPChecksum(TcpSeg))

begin

   { In the WVTLDC state, any segment from the TCP is mapped to the
     corresponding VTPDU as specified for the data transfer phase }

end;


from WFIN to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE)

      provided ( NOT IsFin(TcpSeg)
                 and NOT IsRst(TcpSeg)
                 and VerifyTCPChecksum(TcpSeg))

begin

{ WFIN - Wait for FIN is reached when a DISCON_REQ VTPDU is received}
```

```
{ The TCF waits for corresponding FIN to arrive from the local TCP  }
{ In the WFIN state, any segment from the TCP is mapped to the      }
{  corresponding VTPDU as specified for the data transfer phase     }


end;

from WFIN to WLVTLACK

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE)

      provided ( IsFin(TcpSeg)
                 and VerifyTCPChecksum(TcpSeg))

{ The FIN segment may carry normal, URG or PUSHed data        }

      begin

            DiscReqVtpdu := TcfGetVtpdu(VTL_DISC_REQ);

            if (IsUrg(TcpSeg)) then
             begin
                  ExpdtFlag := TRUE;
                  SetExpdtOption(DiscReqVtpdu);
             end

            if (IsPush(TcpSeg)) then SetPushOption(DiscReqVtpdu);

            { Use the Seq Number to get the VSN }
            If(TcpSeg.SeqNum > Next_SendSeqNum) then
             begin
                  { If this condition happens, then some of the}
                  { intermediate segments are lost. It is not  }
                  { possible to associate a packet count in the
                    VSN }
                  VtlSeq.VsnType := OCTET_BINDING;
                  VtlSeq.OctetBinding := TcpSeqNum;
                  UpdateSendMapList(VtlSeq, SegLength, ExpdtFlag);
                  SetIncompleteVsnIndex(PartialVsnIndex);
              end;
            else if (IsMissingSeg(TcpSeq, Vsn) = TRUE) then
               begin
                  { build a map information message and send to peer}
                  { TCF over COTS interaction point                  }
                  UpdateVsnInfo(TcpSeq, Vsn, SegLength, ExpdtFlag);
                  MapInfoMsg = BuildMapInfo(TcpSeq, Vsn);
                  output TcfToCots.SendTCFMsgRequest(MapInfoMsg);
               end;
            { Use the Seq Number to get the Expedited Data VSN }
```

```
                MapCredit (TcpSeg.Credit, DiscReqVtpdu);
            end;
        BuildDiscReqVtpdu (TcpSeg, length, Security,
                           Precedence, DiscReqVtpdu);
        output TcfToClts.SendTCFdataRequest(DiscReqVtpdu);
    end;

from WLVTLACK to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE)

    provided ( NOT IsRst(TcpSeg)
               and VerifyTCPChecksum(TcpSeg))

begin

{ In the WLVTLACK state, any segment from the TCP is mapped to the
corresponding VTPDU as specified for the data transfer phase }

end;


from WLACK to same

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

    provided ( NOT IsRst(TcpSeg)
               and NOT IslastAck(TcpSeg)
               and VerifyTCPChecksum(TcpSeg))
begin

{ The WLACK state is reached when a Discon Cnf VTPDU is received }
{ Any TCP segments are handled as specified in the data transfer }
{ phase }


end;


from WLACK to CLOSE

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )
```

```
from WLACK to CLOSE

when  IPtoTCF.ReceiveTPDUindication ( TcpSeg : TCP_SEGMENT_TYPE,
                                      length : SEGMENT_LENGTH_TYPE,
                                      Security : SECURITY_TYPE,
                                      Precedence : PRECEDENCE_TYPE )

      provided ( IsRst(TcpSeg)
                 or IslastAck(TcpSeg)
                 and VerifyTCPChecksum(TcpSeg))
begin

{ The ACK corresponding to the FIN segment derived from a Discon }
{ Confirm VTPDU causes a state change to closed, and subsequent  }
{ disassociation }


end;



{  ******* transition due to interactions from CLTS ********  }


      { The finite set of VTPDUs that can be received are
            1. Reset_Request VTPDU.
            2. Reset_Confirm VTPDU
            3. Disconnect_Request VTPDU
            4. Disconnect_Confirm VTPDU
            5. other VTPDUs as specified in the data
               transfer phase.
      }


from ACTIVE_STATE to CLOSE

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsResetReq(Vtpdu) = TRUE and
                 VerifyVTLChecksum(Vtpdu) = TRUE)

      begin

      { A Reset Request in any active state is mapped to a TCP RST }
      { Since the TCP does not reply to a RST, the Reset Confirm   }
      { is generated and trasnmitted                               }

            RstReqVtpdu = Vtpdu;
            TcpSeg = TcfGetTcpSeg();
            SetTcpFin(TcpSeg);
```

```
        if(IsExpdtOption(RstReqVtpdu.DataOptions)) then
         begin
              MapURGptr(TcpSeg, Vtpdu);
              ExpdtFlag := TRUE ;
         end;
        else  ExpdtFlag := FLASE ;

        IsPushOption(RstReqVtpdu.DataOptions) then
              {Set PSH Flag in TCP Segment };

        { If the required VSN binding is available then :}
        if(IsOctetBinding(Vtpdu.SendVSN) = TRUE) then
         begin
              UpdateRecvMapList(Vtpdu, SegLength, ExpdtFlag);
              { The encoding of the VSN is such that the }
              { Octet binding is present, i.e. does not  }
              { have to be derived                       }
        MapRecvVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
         end;
        else
         begin
              { If Octet binding is missing then use local}
              { knowledge of next expected sequence number}
              TcpSeg.SeqNum := Next_RecvSeqNum;
         end;
        If(PiggyBackAck(Vtpdu) = TRUE and
          (IsOctetBinding(Vtpdu.NextRecvVSN) = TRUE)) then
              begin
                   MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
                   RecordVACK( Vtpdu.NextRecvVSN,
                         PreviousNextRecvVSN);
                   MapCreditToTcp (TcpSeg.Credit, Vtpdu);
                   RecordCredit (Vtpdu, PreviousCreditValue);
              end
        else {No piggybacked Ack Num }
              begin
                   MapVACKtoAck(TcpSeg.AckNum,
                               PreviousNextRecvVSN);
                   MapCreditToTcp(TcpSeg.Credit,
                               PreviousCreditValue);
              end

        BuildTcpSeg (TcpSeg, Vtpdu);
        output TcfToIp.SendTPDUrequest(TcpSeg,
        NORMAL_SECURITY, DEFAULT_PRECEDENCE);

        { Build a Disconnect Confirm VTPDu        }

        BuildDiscCnfVtpdu (DiscCnfVtpdu);
        output TcfToClts.SendTCFdataRequest(DiscCnfVtpdu);
end;
```

```
from ACTIVE_STATE to CLOSE

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsResetCnf(Vtpdu) = TRUE and
                 VerifyVTLChecksum(Vtpdu) = TRUE)

      begin

      { A Reset Confirm in any active state is used to close the   }
      { TCF association. Nothing is output                         }


      end;


from OPEN to WFIN

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsDisconReq(Vtpdu) = TRUE and
                 VerifyVTLChecksum(Vtpdu) = TRUE)

      begin

      { A Disconnect Request results in generation of a FIN segment}
      { and state is changed to WFIN - Wait for corresponding FIN  }

            RstReqVtpdu = Vtpdu;
            TcpSeg = TcfGetTcpSeg();
            SetTcpFin(TcpSeg);

            if (IsExpdtOption(RstReqVtpdu.DataOptions)) then
             begin
                  MapURGptr(TcpSeg, Vtpdu);
                  ExpdtFlag := TRUE ;
             end;
            else  ExpdtFlag := FALSE ;

            if(IsPushOption(RstReqVtpdu.DataOptions)) then
             begin
                  {Set PSH Flag in TCP Segment };
             end;


            { If the required VSN binding is available then :}
            if(IsOctetBinding(Vtpdu.SendVSN) = TRUE) then
             begin
                  UpdateRecvMapList(Vtpdu, SegLength, ExpdtFlag);
                  { The encoding of the VSN is such that the }
                  { Octet binding is present, i.e. does not  }
```

```
                { have to be derived                          }
            MapRecvVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
                If(PiggyBackAck(Vtpdu) = TRUE and
                  (IsOctetBinding(Vtpdu.NextRecvVSN) = TRUE)) then
                 begin
                      MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
                      RecordVACK( Vtpdu.NextRecvVSN,
                            PreviousNextRecvVSN);
                      MapCreditToTcp (TcpSeg.Credit, Vtpdu);
                      RecordCredit (Vtpdu, PreviousCreditValue);
                 end
                else
                 begin
                      MapVACKtoAck(TcpSeg.AckNum,
                                   PreviousNextRecvVSN);
                      MapCreditToTcp(TcpSeg.Credit,
                                   PreviousCreditValue);
                 end

                BuildTcpSeg (TcpSeg, Vtpdu);
                output TcfToIp.SendTPDUrequest(TcpSeg,
                NORMAL_SECURITY, DEFAULT_PRECEDENCE);
            end; { if OctetBinding }
           else
            begin
                if ( BufferVtpduPolicy() = TRUE ) then
                  { Buffer VTPDUs with incomplete VSNs }
                    BufferVtpdus(Vtpdu);
                else { discard the VTPDU }
            end;

      end;


from WFIN to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( NOT IsReset(Vtpdu) = TRUE and
                 VerifyVTLChecksum(Vtpdu) = TRUE)
      begin

      { In WFIN state, any received VTPDUs are handled as specified}
      { in data tansfer phase                                       }


      end;

from WVTLDC to WLACK

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)
```

```
provided ( IsDisconCnf(Vtpdu) = TRUE and
           VerifyVTLChecksum(Vtpdu) = TRUE)

begin

{ A Disconnect Confirm results in generation of a FIN segment}
{ and state is changed to WLACK - Wait for corresponding ACK }

      RstCnfVtpdu = Vtpdu;
      TcpSeg = TcfGetTcpSeg();
      SetTcpFin(TcpSeg);

      if (IsExpdtOption(RstReqVtpdu.DataOptions)) then
       begin
            MapURGptr(TcpSeg, Vtpdu);
            ExpdtFlag := TRUE ;
       end;
      else  ExpdtFlag := FALSE ;

      if(IsPushOption(RstReqVtpdu.DataOptions)) then
       begin
            {Set PSH Flag in TCP Segment };
       end;


      { If the required VSN binding is available then :}
      if(IsOctetBinding(Vtpdu.SendVSN) = TRUE) then
       begin
            UpdateRecvMapList(Vtpdu, SegLength, ExpdtFlag);
            { The encoding of the VSN is such that the }
            { Octet binding is present, i.e. does not  }
            { have to be derived                       }
     MapRecvVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
            If(PiggyBackAck(Vtpdu) = TRUE and
              (IsOctetBinding(Vtpdu.NextRecvVSN) = TRUE)) then
             begin
                  MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
                  RecordVACK( Vtpdu.NextRecvVSN,
                        PreviousNextRecvVSN);
                  MapCreditToTcp (TcpSeg.Credit, Vtpdu);
                  RecordCredit (Vtpdu, PreviousCreditValue);
             end
            else
             begin
                  MapVACKtoAck(TcpSeg.AckNum,
                              PreviousNextRecvVSN);
                  MapCreditToTcp(TcpSeg.Credit,
                              PreviousCreditValue);
             end

            BuildTcpSeg (TcpSeg, Vtpdu);
```

```
                output TcfToIp.SendTPDUrequest(TcpSeg,
                NORMAL_SECURITY, DEFAULT_PRECEDENCE);
            end; { if OctetBinding }
           else
            begin
                if ( BufferVtpduPolicy() = TRUE ) then
                  { Buffer VTPDUs with incomplete VSNs }
                     BufferVtpdus(Vtpdu);
                else { discard the VTPDU }
            end;

      end;


from WLACK to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( NOT IsReset(Vtpdu)   and
                 VerifyVTLChecksum(Vtpdu) = TRUE)

      begin

      { In WLACK state, any received VTPDUs are handled as specified}
      { in data tansfer phase                                       }


      end;


from WLVTLACK to CLOSE

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsReset(Vtpdu) = TRUE
                 OR IsLastVtlAck(Vtpdu) = TRUE
                 and VerifyVTLChecksum(Vtpdu) = TRUE)


      begin

      { An Ack corresponding to a previously sent Disconnect Confirm }
      { results in the corresponding TCP ACK and a change to CLOSE   }

            TcpSeg = TcfGetTcpSeg();
            MapVSNtoSeq(TcpSeg.SeqNum, Vtpdu.SendVSN);
            MapVACKtoAck(TcpSeg.AckNum, Vtpdu.NextRecvVSN);
            MapCreditToTcp (TcpSeg.Credit, Vtpdu);
            BuildTcpSeg (TcpSeg, Vtpdu);
            output TcfToIp.SendTPDUrequest(TcpSeg, NORMAL_SECURITY
                                           DEFAULT_PRECEDENCE);
```

```
            end;

end; { of TCP TCF body }
```

TP4_TCF Specification For Connection Termination

type

```
      DISC_DATA_OPTIONS_TYPE =
            record
                  OptionsFlag          : ONE_BYTE;
                  PiggyBackAck         : PIGGY_BACK_ACK_TYPE;
                  TotalEDlen           : DATA_LENGTH_TYPE;
            end;

      { The encoding of the Options Flag is as : }

      PIGGYBACK_ACK       = 0x01;
      EXPDT_DATA          = 0x02;
      PUSH_DATA           = 0x04;

      DISCONNECT_VTPDU_TYPE  =
            record
                  VtpduHdr             : VTPDU_HEADER_TYPE;
                  SendVSN              : VSN_TYPE;
                  Reason               : TWO_BYTES;
                  DataOptions          : DISC_DATA_OPTIONS_TYPE;
                  Data                 : DATA_TYPE;
                  CheckSum             : TWO_BYTES;
            end;

      RESET_REQUEST_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
      RESET_CONFIRM_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;

      DISCONNECT_REQUEST_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
      DISCONNECT_CONFIRM_VTPDU_TYPE : DISCONNECT_TPDU_TYPE;
```

var

```
      DataVtpdu           :    DATA_VTPDU_TYPE;
      DataAckVtpdu        :    DATA_ACK_VTPDU_TYPE;
```

```
    ExpdtDataVtpdu      :   EXPEDITED_DATA_VTPDU_TYPE;
    ExpdtAckVtpdu       :   EXPEDITED_ACK_VTPDU_TYPE;

    RstReqVtpdu         :   RESET_REQUEST_VTPDU_TYPE;
    RstCnfVtpdu         :   RESET_CONFIRM_VTPDU_TYPE;
    DiscReqVtpdu        :   DISCONNECT_REQUEST_VTPDU_TYPE;
    DiscCnfVtpdu        :   DISCONNECT_CONFIRM_VTPDU_TYPE;

    Tp4Data             :   ISO_TP4_DT_TYPE;
    Tp4EData            :   ISO_TP4_ED_TYPE;
    Tp4AK               :   ISO_TP4_AK_TYPE;
    Tp4EA               :   ISO_TP4_EA_TYPE;

    MappingInfoMsg      :   TCF_MSG_TYPE;
    SendListElement     :   SEND_MAP_TABLE_ELEMENT_TYPE;
    RecvListElement     :   RECV_MAP_TABLE_ELEMENT_TYPE;


( Channel Definitions for communication with the TCF Module )

Channel   IP_Access_Point  ( From_IP,  To_IP );

    by   From_IP  : ReceiveTPDUindication ( Tp4Tpdu : TP4_TPDU_TYPE );

    by   To_IP    : SendTPDUrequest ( Tp4Tpdu : TP4_TPDU_TYPE );

                    ReceiveTPDUrequest ;


Channel   CLTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFdataRequest;
                    SendTCFdataRequest ( Vtpdu : VTPDU_TYPE );

    by  To_TCF   :  ReceiveTCFdataIndication ( Vtpdu : VTPDU_TYPE );


Channel   COTS_Access_Point ( From_TCF,  To_TCF );

    by  From_TCF :  ReceiveTCFMsgRequest;
                    SendTCFMsgRequest ( TcfMsg : TCF_MESSAGE_TYPE );

    by  To_TCF   :  ReceiveTCFMsgIndication ( TcfMsg : TCF_MESSAGE_TYPE );


                ( Module Header Definitions )


Module InternetProtocol_Type  process ;
```

```
    ip  IPtoTCF :  IP_Access_Point (From_IP) ;

end;

Module TCF_Type activity
          (End_Point_Id : END_POINT_TYPE) { parameter to TCF }

    ip    { list of interaction points }
          TcfToIp      : IP_Access_Point ( To_IP) individual queue;
          TcfToClts    : CLTS_Access_Point ( From_TCF) individual queue ;
          TcfToCots    : COTS_Access_Point ( From_TCF) individual queue ;
end;


Module CLTS_Type process;

    ip    CltsToTcf   : CLTS_ACCESS_POINT ( To_TCF )

end;


Module COTS_Type process;

    ip    CotsToTcf   : COTS_ACCESS_POINT ( To_TCF )

end;

                { Body Definitions for Modules }


Body      InternetProtocol_Body for InternetProtocol_Type; external;

Body      Clts_Body   for CLTS_Type; external;

Body      Cots_Body   for COTS_Type; external;


Body TCF_Body for TCF_Type

state

    ANY : (CLOSE, OPEN);

    ACTIVE_STATE : (OPEN);

var
```

```
{ Functions and procedures used in the module body }

    { Suppurt functions and proceduers are as     }
    { specified in the Connection Establishment   }
    { phase and Data Transfer phase               }

procedure      BuildDATAVtpdu (Tp4Data : ISO_TP4_DT_TYPE,,
                               Length : DATA_LENGTH_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var DataVtpdu : DATA_VTPDU_TYPE)

begin
        FillVTPDUhdr(DataVtpdu.VtpduHdr, Security, Precedence);
        DataVtpdu.VtpduHdr.VtpduCode := DATA_VTPDU;
        CopyUserData(Tp4Data.Data, Vtpdu.Data, Lenght);
        ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildEDATAVtpdu (Tp4EData : ISO_TP4_ED_TYPE,
                               Length : DATA_LENGTH_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var EDataVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
        FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
        EDataVtpdu.VtpduHdr.VtpduCode := EDATA_VTPDU;
        EDataVtpdu.TotalEDlen := Length;
        CopyUserData(TcpSeg.Data, Vtpdu.Data, Lenght);
        ComputeVTLCheckSum(Vtpdu);

end;


procedure      BuildACKVtpdu (Tp4Ack : ISO_TP4_AK_TYPE,
                               Security : SECURITY_TYPE,
                               Precedence : PRECEDENCE_TYPE,
                               var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
        FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
        EDataVtpdu.VtpduHdr.VtpduCode := DATA_ACK_VTPDU;
        { If FCC information is present in the Tp4Ack, then fill in the
          optional fields of the AckVtpdu    }
        ComputeVTLCheckSum(Vtpdu);
```

```
end;

procedure       BuildEACKVtpdu (Tp4EAck : ISO_TP4_EA_TYPE,
                                Length : DATA_LENGTH_TYPE,
                                Security : SECURITY_TYPE,
                                Precedence : PRECEDENCE_TYPE,
                                var AckVtpdu : EXPEDITED_DATA_VTPDU_TYPE)

begin
          FillVTPDUhdr(EDataVtpdu.VtpduHdr, Security, Precedence);
          EDataVtpdu.VtpduHdr.VtpduCode := EDATA_ACK_VTPDU;
          { Mark Expedited Ack option in the ACK VTPDU }
          ComputeVTLCheckSum(Vtpdu);

end;


trans


    to CLOSE
          begin

          end


          { transition due to interactions from IP }

    { A TP4 TPDU can carry the following information during the }
    { connection termination phase
          1. DR
          2. DC
    }

    from OPEN to same

    when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : ISO_TP4_TPDU_TYPE)


          provided (IsDr(Tp4Tpdu) and VerifyTp4Checksum(Tp4Tpdu))
          begin

                RstReqVtpdu := TcfGetVtpdu(VTL_RST_CNF);
                { Since the DR VTPDU does not carry a sequence number }
                { The VSN in the RstReq is generated from localy kept }
                { information. A reason code if used is also included }
                { vtpdu }

                VtlSeq.VsnType := PACKET_BINDING BIT OR
                                  OCTET_BINDING;
```

```
            VtlSeq.PacketBinding := Next_SendPacketNum;
            VtlSeq.OctetBinding := NextSendSeqNum;
            SetVsn(RstReqVtpdu, VtlSeq);
            BuildRstVtpdu (T4Tpdu, RstReqVtpdu);
            output TcfToClts.SendTCFdataRequest(RstReqVtpdu);
      end;

from OPEN to CLOSE

when  IPtoTCF.ReceiveTPDUindication ( Tp4Tpdu : ISO_TP4_TPDU_TYPE)

      provided (IsDc(Tp4Tpdu) and VerifyTp4Checksum(Tp4Tpdu))
      begin

            RstCnfVtpdu := TcfGetVtpdu(VTL_RST_REQ);
            { Since the DC VTPDU does not carry a sequence number }
            { The VSN in the RstCnf is generated from localy kept }
            { information }

            VtlSeq.VsnType := PACKET_BINDING BIT OR
                              OCTET_BINDING;
            VtlSeq.PacketBinding := Next_SendPacketNum;
            VtlSeq.OctetBinding := NextSendSeqNum;
            SetVsn(RstReqVtpdu, VtlSeq);
            BuildRstReqVtpdu (T4Tpdu, RstCnfVtpdu);
            output TcfToClts.SendTCFdataRequest(RstCnfVtpdu);
      end;

{  ******* transition due to interactions from CLTS ********  }

      { The finite set of VTPDUs that can be received during
        connection termination are :
            1. Reset_Request VTPDU.
            2. Reset_Confirm VTPDU.
            3. Disconnect_Request VTPDU
            4. VTPUDs used for Data Transfer

from OPEN to same

when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

      provided ( IsVTLRstReq(Vtpdu) = TRUE and
                        VerifyVTLChecksum(Vtpdu) = TRUE)
      begin

            { A ResetReq VTPDU is mapped to a DR TPDU }
            { If the VTPDU carries more than 64 bytes }
            { of data, then the data is discarded. A  }
```

```
                { reason code if present is used as such  }

                Tp4Dr := TcfGetTp4Tpdu(DR);
                BuildDrTpdu(Tp4Dr, Vtpdu);
                output TcfToIp.SendTPDUrequest(Tp4Dr);
          end;



     from OPEN to CLOSE

     when  CltsToTcf.ReceiveTCFdataIndication (Vtpdu : VTPDU_TYPE)

          provided ( IsVTLRstCnf(Vtpdu) = TRUE and
                          VerifyVTLChecksum(Vtpdu) = TRUE)
          begin

                { A ResetCnf VTPDU is mapped to a DC TPDU }

                Tp4Dc := TcfGetTp4Tpdu(DC);
                BuildDrTpdu(Tp4Dc, Vtpdu);
                output TcfToIp.SendTPDUrequest(Tp4Dc);
          end;



end; { end of TP4 TCF body }
```