The Consultative Committee for Space Data Systems

Report Concerning Space Data System Standards

CCSDS FILE DELIVERY PROTOCOL (CFDP)—

PART 2

IMPLEMENTERS GUIDE

INFORMATIONAL REPORT

CCSDS 720.2-G-3

GREEN BOOK

April 2007

AUTHORITY

Issue:	Informational Report, Issue 3
Date:	April 2007
Location:	Washington, DC, USA

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and reflects the consensus of technical panel experts from CCSDS Member Agencies. The procedure for review and authorization of CCSDS Reports is detailed in the *Procedures Manual for the Consultative Committee for Space Data Systems*.

This document is published and maintained by:

CCSDS Secretariat
Office of Space Communication (Code M-3)
National Aeronautics and Space Administration
Washington, DC 20546, USA

FOREWORD

This document is a CCSDS Report, which contains background and explanatory material to support the CCSDS Recommended Standard, *CCSDS File Delivery Protocol* (reference [1]).

Through the process of normal evolution, it is expected that expansion, deletion, or modification to this Report may occur. This Report is therefore subject to CCSDS document management and change control procedures, which are defined in reference [2]. Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this report should be addressed to the CCSDS Secretariat at the address on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were:

Member Agencies

- Agenzia Spaziale Italiana (ASI)/Italy.
- British National Space Centre (BNSC)/United Kingdom.
- Canadian Space Agency (CSA)/Canada.
- Centre National d'Etudes Spatiales (CNES)/France.
- Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Federal Space Agency (Roskosmos)/Russian Federation.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- Japan Aerospace Exploration Agency (JAXA)/Japan.
- National Aeronautics and Space Administration (NASA)/USA.

Observer Agencies

- Austrian Space Agency (ASA)/Austria.
- Belgian Federal Science Policy Office (BFSPO)/Belgium.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- Centro Tecnico Aeroespacial (CTA)/Brazil.
- Chinese Academy of Space Technology (CAST)/China.
- Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
- Danish Space Research Institute (DSRI)/Denmark.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Institute of Space Research (IKI)/Russian Federation.
- KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- Korea Aerospace Research Institute (KARI)/Korea.
- MIKOMTEK: CSIR (CSIR)/Republic of South Africa.
- Ministry of Communications (MOC)/Israel.
- National Institute of Information and Communications Technology (NICT)/Japan.
- National Oceanic & Atmospheric Administration (NOAA)/USA.
- National Space Organization (NSPO)/Taipei.
- Space and Upper Atmosphere Research Commission (SUPARCO)/Pakistan.
- Swedish Space Corporation (SSC)/Sweden.
- United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

Document	Title	Date	Status
CCSDS 720.2-G-1	CCSDS File Delivery Protocol (CFDP)—Part 2: Implementers Guide	January 2002	Original issue, superseded
CCSDS 720.2-G-2	CCSDS File Delivery Protocol (CFDP)—Part 2: Implementers Guide	September 2003	Issue 2, superseded
CCSDS 720.2-G-3	CCSDS File Delivery Protocol (CFDP)—Part 2: Implementers Guide, Informational Report, Issue 3	April 2007	Current issue

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

Figure Page

Table

CONTENTS (continued)

1 INTRODUCTION

1.1 PURPOSE

This report is an adjunct document to the Consultative Committee for Space Data Systems (CCSDS) Recommended Standard for File Delivery Protocol (reference [1]). It contains material which will be helpful in understanding the primary document, and which will assist decision makers and implementers in evaluating the applicability of the protocol to mission needs and in making implementation, option selection, and configuration decisions related to the protocol.

1.2 SCOPE

This report provides supporting descriptive and tutorial material**. This document is not part of the Recommended Standard.** In the event of conflicts between this report and the Recommended Standard, the Recommended Standard shall prevail.

1.3 ORGANIZATION OF THIS REPORT

This report is divided into two parts. Part 1 (reference [3]) provides an introduction to the concepts, features, and characteristics of the CCSDS File Delivery Protocol (CFDP). It is intended for an audience of persons unfamiliar with the CFDP or related protocols. The second part of this report (this document) is an implementers guide. It provides information to assist implementers in understanding the details of the protocol and in the selection of appropriate options, and it contains suggestions and recommendations about implementation-specific subjects. This document also contains implementation reports from various member Agencies, reports on testing of the implementations and protocol, and the requirements upon which the CFDP is based.

1.4 CONVENTIONS AND DEFINITIONS

1.4.1 BIT NUMBERING CONVENTION AND NOMENCLATURE

In this document, the following convention is used to identify each bit in an N-bit field. The first bit in the field to be transmitted (i.e., the most left-justified when drawing a figure) is defined to be ‘Bit 0’; the following bit is defined to be ‘Bit 1’, and so on up to ‘Bit N-1’. When the field is used to express a binary value (such as a counter), the Most Significant Bit (MSB) shall be the first transmitted bit of the field, i.e., ‘Bit 0’, as shown in figure 1-1.

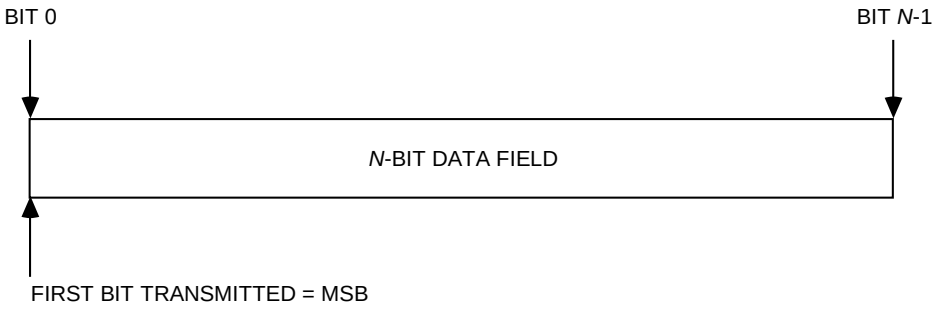


Figure 1-1: Bit Numbering Convention

In accordance with modern data communications practice, spacecraft data fields are often grouped into 8-bit 'words' which conform to the above convention. Throughout this Report, the nomenclature shown in figure 1-2 is used to describe this grouping.

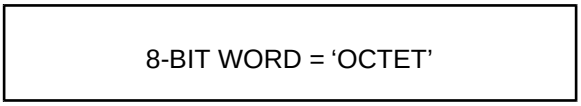


Figure 1-2: Octet Convention

By CCSDS convention, all 'spare' bits shall be permanently set to value 'zero'.

1.4.2 DEFINITIONS

Within the context of this document the following definitions apply:

A *file* is a bounded or unbounded named string of octets that resides on a storage medium.

A *filestore* is a system used to store files; CFDP defines a standard *virtual filestore* interface through which CFDP accesses a filestore and its contents.

A *CFDP protocol entity* (or *CFDP entity*) is a functioning instance of an implementation of the CFDP protocol, roughly analogous to an Internet protocol 'host'. Each CFDP entity has access to exactly one filestore. (It is recognized that the single [logical] filestore of a CFDP entity might encompass multiple physical storage partitions, but any specific reference to such a partition in identifying the location or destination of a file is expected to be encoded as part of the file's name [e.g., 'pathname'].) Each entity also maintains a *Management Information Base* (MIB), which contains such information as default values for user communications requirements (e.g., for address mapping, and for communication timer settings).

The functional concatenation of a file and related *metadata* is termed a *File Delivery Unit* (FDU); in this context the term 'metadata' is used to refer to any data exchanged between

CFDP protocol entities in addition to file content, typically either additional application data (such as a 'message to user') or data that aid the recipient entity in effectively utilizing the file (such as file name).

NOTES

1 An FDU may consist of metadata only.

2 The term 'file' is frequently used in this specification as an abbreviation for 'file delivery unit'; only when the context clearly indicates that actual files are being discussed should the term 'file' not be read as 'file delivery unit'. For example, in the explanation of the record type parameter or the source and destination file name parameters of the CFDP Service Definition, the term 'file' should not be read as 'file delivery unit'.

The individual, bounded, self-identifying items of CFDP data transmitted between CFDP entities are termed *CFDP Protocol Data Units* (PDUs), or *CFDP PDU*s. Unless otherwise noted, in this document the term 'PDU' always means 'CFDP PDU'. CFDP PDUs are of two general types: *File Data PDUs*, which convey the contents of the files being delivered, and *File Directive PDUs*, which convey only metadata and other non-file information that advance the operation of the protocol.

A *transaction* is the end-to-end transmission of a single FDU between two CFDP entities. A single transaction normally entails the transmission and reception of multiple PDUs. Each transaction is identified by a unique transaction ID; all elements of any single FDU, both file content and metadata, are tagged with the same CFDP transaction ID.

Any single end-to-end file transmission task has two associated entities: the *source* and the *destination*. The source is the entity that has the file at the beginning of the task. The destination is the entity that has a copy of the file when the task is completed.

Each end-to-end file transmission task comprises one or more point-to-point file copy operations. A file copy operation has two associated entities: the entity that has a copy of the file at the beginning of the operation (the *sender* or *sending entity*) and the entity that has a copy of the file when the operation is completed (the *receiver* or *receiving entity*). In the simplest case, the only sender of the file is the source and the only receiver is the destination. In more complex cases (the general case), there are additional *waypoint* entities that receive and send copies of the file; the source is the first sender and the destination is the last receiver.

The term *CFDP user* refers to the software task that causes the local entity to initiate a transaction, or the software task that is notified by the local entity of the progress or completion of a transaction. The CFDP user local to the source entity is referred to as the *source CFDP user*. The CFDP user local to the destination entity is referred to as the *destination CFDP user*. The CFDP user may be operated by a human or by another software process. Unless otherwise noted, the term *user* always refers to the CFDP user.

A *message to user* (or *user message*) allows delivery of information related to a transaction to the destination user in synchronization with the transaction.

A *filestore request* is a request to the remote filestore for service (such as creating a directory, deleting a file, etc.) at the successful completion of a transaction.

Service primitives form the software interface between the CFDP user and its local entity. The user issues *request* service primitives to the local entity to request protocol services, and the local entity issues *indication* service primitives to the user to notify it of the occurrence of significant protocol events.

1.5 REFERENCES

The following documents are referenced in the text of this Report. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Report are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS documents.

[1] *CCSDS File Delivery Protocol (CFDP).* Recommendation for Space Data System Standards, CCSDS 727.0-B-3. Blue Book. Issue 3. Washington, D.C.: CCSDS, June 2005.

[2] *Procedures Manual for the Consultative Committee for Space Data Systems*. CCSDS A00.0-Y-9. Yellow Book. Issue 9. Washington, D.C.: CCSDS, November 2003.

[3] *CCSDS File Delivery Protocol (CFDP)—Part 1: Introduction and Overview*. Report Concerning Space Data System Standards, CCSDS 720.1-G-3. Green Book. Issue 3. Washington, D.C.: CCSDS, January 2007.

[4] *Specification and Description Language (SDL).* ITU Recommendation Z.100. Blue Book. Volume X.1 – X.5. Geneva, Switzerland: ITU General Secretariat, 1988.

[5] D. Comer and D. Stevens. *Internetworking with TCP/IP, Volume II: Design, Implementation, and Internals*. Second edition. Prentice Hall, 1999.

2 CFDP PROTOCOL DATA UNITS

2.1 OVERVIEW

This section presents the formats of the CFDP Protocol Data Units (PDU), as well as the relationships between the PDUs and the CFDP primitives. PDUs are exchanged between CFDP entities and, therefore, both their contents and their formats are defined. Primitives are not exchanged between protocol entities and, therefore, their contents are defined but their formats are not.

The information in this section is provided as an aid to visualizing and understanding the primitives and PDUs, and their relationships. In all cases more detail, and the protocol specifications and procedures, are found in reference [1]. As always, reference [1] is the defining document and in case of any disagreements between it and this Report, reference [1] is the authoritative document.

All PDUs consist of two components: the Fixed PDU Header and the PDU Data Field.

Two PDU types are defined: File Directive and File Data. The PDU type is signaled in the PDU Type field of the Fixed PDU Header, as shown in table 2-1 and subsection 2.2.

Table 2-1: PDU Type Code

Field	Values
PDU type	'0' - File Directive '1' - File Data

The format of the data field of File Data PDUs, which are the PDUs used to deliver the actual file data, is shown in 2.3.2.

The data field of File Directive PDUs consists of a Directive Code octet followed by a Directive Parameter field. The File Directive Codes are shown in table 2-2. The formats of each of the different file directive PDUs are shown in subsections 2.3 through 2.5.

Table 2-2: File Directive Codes

Directive Code (hexadecimal)	Action
00	Reserved
01	Reserved
02	Reserved
03	Reserved
04	EOF PDU
05	Finished PDU
06	ACK PDU
07	Metadata PDU
08	NAK PDU
09	Prompt PDU
0C	Keep Alive PDU
0D–FF	Reserved

The relationships between primitives and PDUs are shown in figure 2-1. The figure also shows the relationships of the primitives and PDUs to the operational process from initiation through termination. The MIB is shown on the diagram since its (minimum) contents are defined in the CFDP, and some of those contents are necessary to complete the Metadata PDU initiated by the Put Request. The format of each of the PDUs is presented in the remainder of this section.

In several cases, the Directive Parameter field of a File Directive includes a four-bit Condition Code. The Condition Code shall in each case indicate one of the conditions shown in table 2-3.

Table 2-3: Condition Codes

Condition Code (binary)	Condition
0000	No error
0001	Positive ACK limit reached
0010	Keep alive limit reached
0011	Invalid transmission mode
0100	Filestore rejection
0101	File checksum failure
0110	File size error
0111	NAK limit reached
1000	Inactivity detected
1001	Invalid file structure
1010 – 1101	(reserved)
1110	Suspend.request received
1111	Cancel.request received

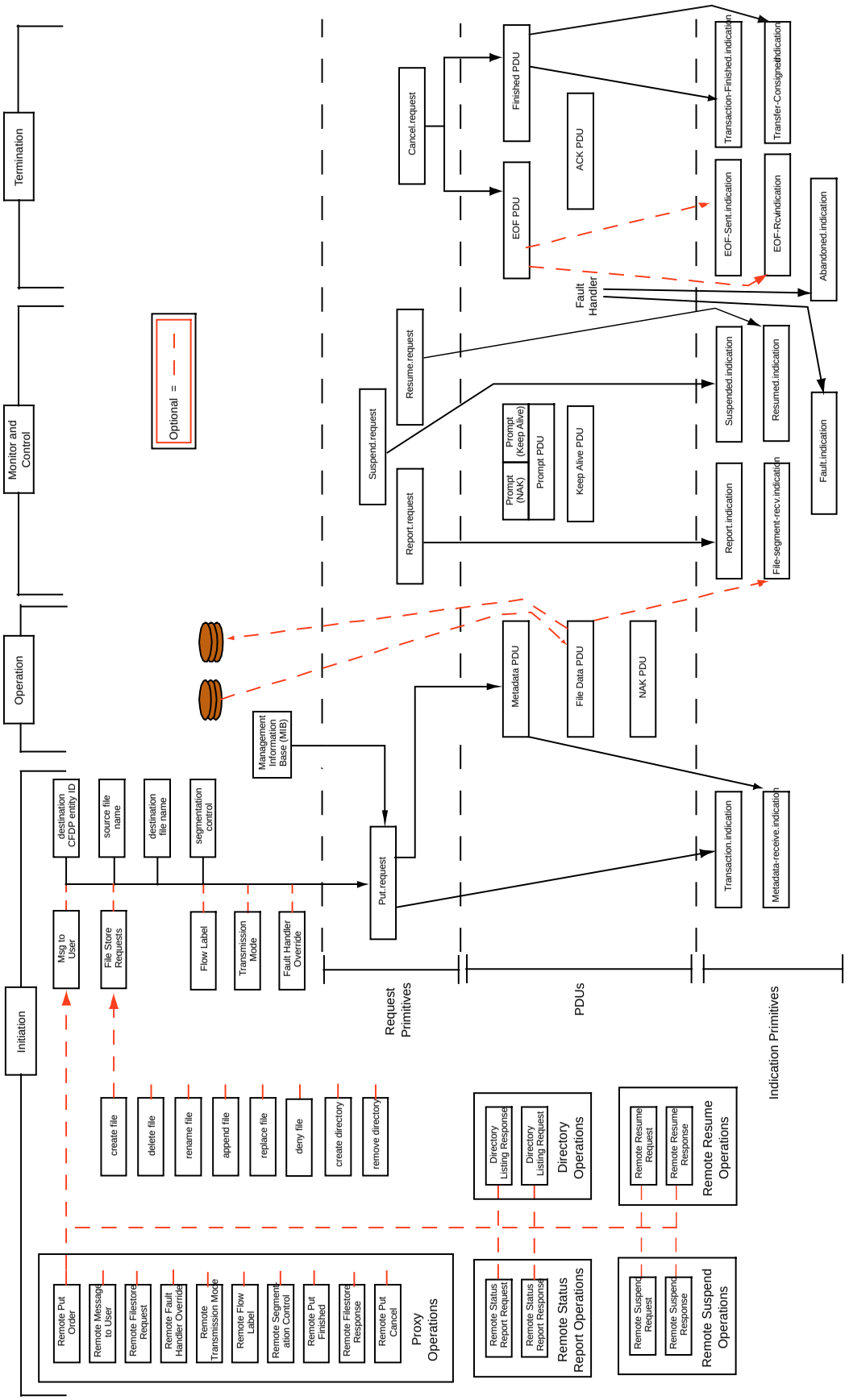


Figure 2-1: Operations View

2.2 FIXED PDU HEADER

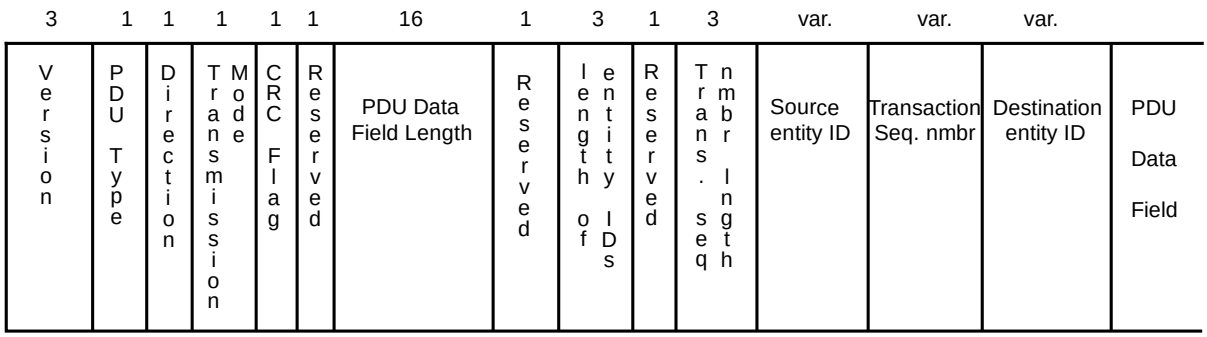
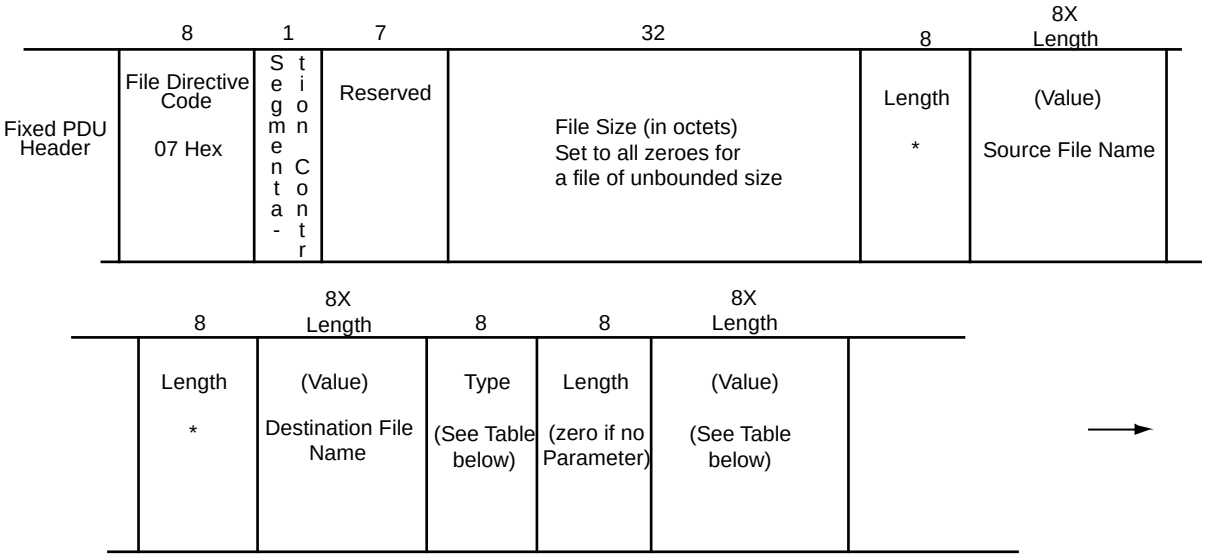


Table 2-4: Fixed PDU Header Fields

Field	Length (bits)	Values	Comment
Version	3	'000'	For the first version.
PDU type	1	'0' — File Directive '1' — File Data	
Direction	1	'0' — toward file receiver '1' — toward file sender	Used to perform PDU forwarding.
Transmission Mode	1	'0' — acknowledged '1' — unacknowledged	
CRC Flag	1	'0' — CRC not present '1' — CRC present	
Reserved for future use	1	set to '0'	
PDU Data field length	16		In octets.
Reserved for future use	1	set to '0'	
Length of entity IDs	3		Number of octets in entity ID less one; i.e., '0' means that entity ID is one octet. Applies to all entity IDs in the PDU header.
Reserved for future use	1	set to '0'	
Length of Transaction sequence number	3		Number of octets in sequence number less one; i.e., '0' means that sequence number is one octet.
Source entity ID	variable		Uniquely identifies the entity that originated the transaction.
Transaction sequence number	variable		Uniquely identifies the transaction, among all transactions originated by this entity.
Destination entity ID	variable		Uniquely identifies the entity that is the final destination of the transaction's metadata and file data.

2.3 OPERATION PDUs

2.3.1 METADATA PDU



* LV Length field indicates zero length and LV value field omitted when there is no associated file, e.g. messages used for Proxy operations

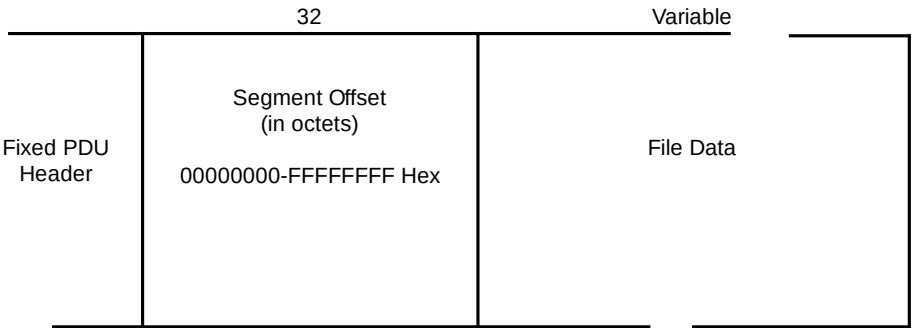
Table 2-5: Metadata Segmentation Control Field Contents

Segmentation Control
'0' - Record boundaries respected
'1' - Record boundaries not respected

Table 2-6: Metadata Type-Length-Value (TLV) Field Codes

Type Field Code	Contents of Value Field
00 Hex	Filestore Request
02 Hex	Message to User
04 Hex	Fault Handler Overrides
05 Hex	Flow Label

2.3.2 FILE DATA PDU



2.3.3 NEGATIVE ACKNOWLEDGMENT (NAK) PDU

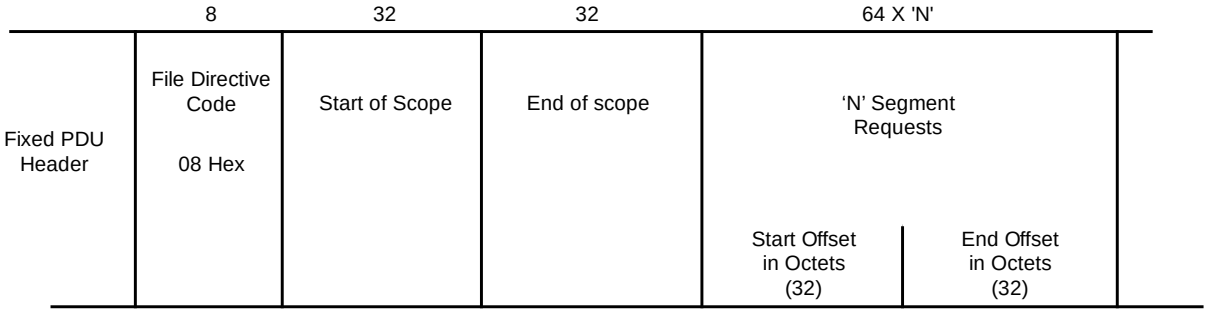


Table 2-7: Segment Request Form

Parameter	Length (bits)	Values	Comments
Start offset	32	Data — Offset of start of requested segment Metadata — 00000000 (hex)	In octets
End Offset	32	Data — Offset of first octet after end of requested segment Metadata — 00000000 (hex)	In octets

2.4 MONITOR AND CONTROL PDUs

2.4.1 PROMPT PDU

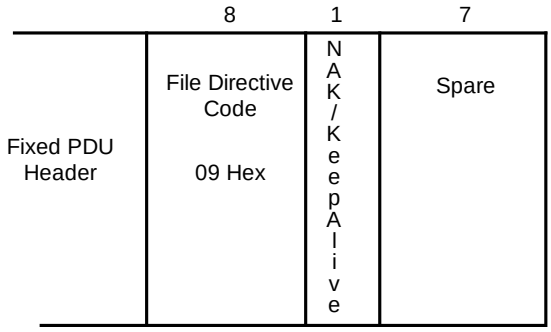
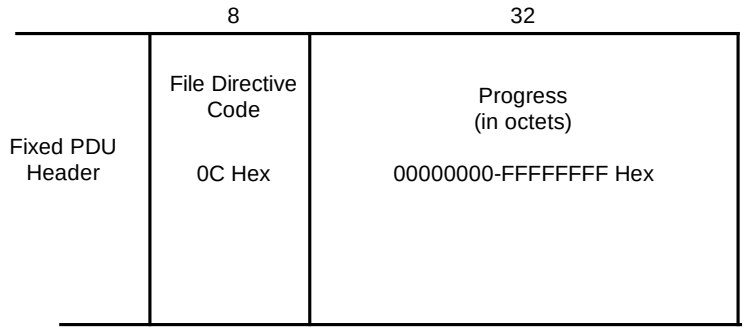


Table 2-8: Prompt PDU NAK/Keep Alive Field Contents

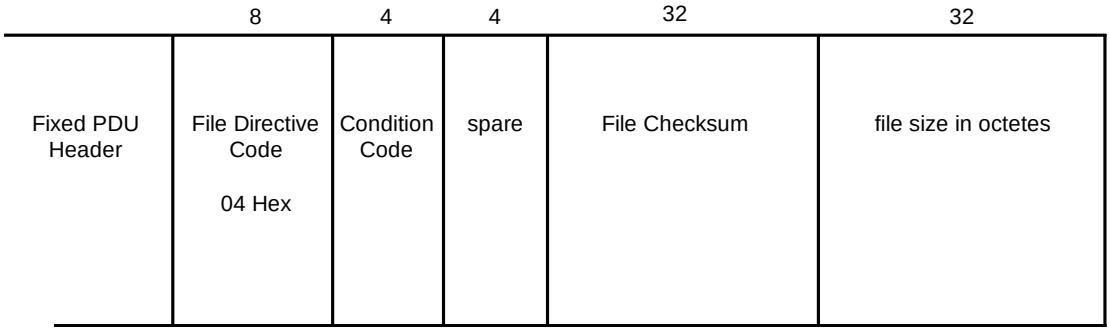
NAK/Keep Alive Code
'0' - NAK
'1' - Keep Alive

2.4.2 KEEP ALIVE PDU



2.5 TERMINATION PDUs

2.5.1 END OF FILE (EOF) PDU



NOTES

1 File Checksum: Modulo 2^{32} word-wide addition (where 'word' is defined as 4 octets) of all file segment data transmitted by the sender (regardless of the condition code, i.e., even if the condition code is other than 'No error'), aligned with reference to the start of file.

2 File Size: Expressed in octets. This value shall be the total number of file data octets transmitted by the sender, regardless of the condition code (i.e., it shall be supplied even if the condition code is other than 'No error').

3 Unacknowledged-mode transactions always terminate on receipt of the EOF (No error) PDU; therefore, any Metadata or file data PDU received after the EOF (No error) PDU for the same transaction may be ignored.

2.5.2 FINISHED PDU

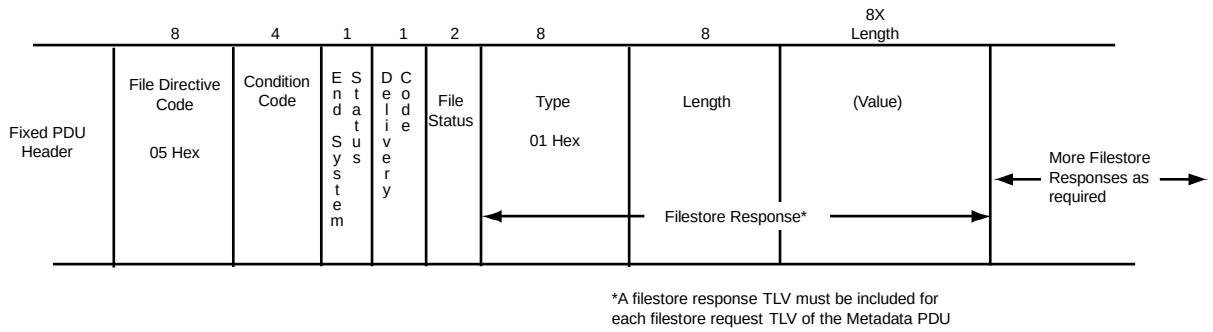
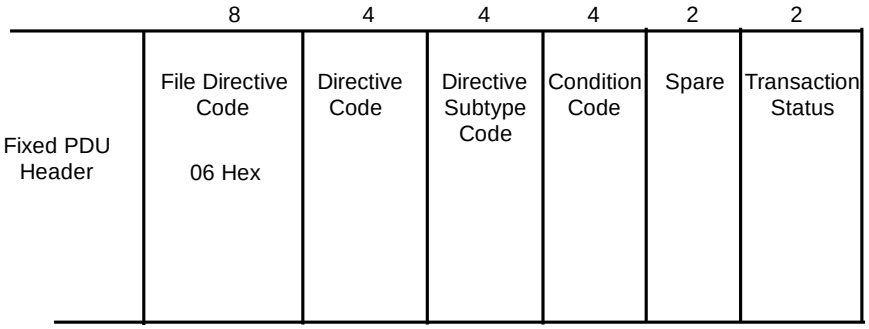


Table 2-9: Finished PDU Field Codes

Parameter	Values	Comment
End System Status	'0' - Generated by Waypoint '1' - Generated by End System	
Delivery Code	'0' - Data Complete '1' - Data Incomplete	
File Status Codes	'00' — Delivered file discarded deliberately '01' — Delivered file discarded due to filestore rejection '10' — Delivered file retained in filestore successfully '11' — Delivered file status unreported	File status is meaningful only when the transaction includes the transmission of file data.

2.5.3 POSITIVE ACKNOWLEDGMENT (ACK) PDU



NOTE – Transaction Status parameter:

00 – Undefined: The transaction to which the acknowledged PDU belongs is not currently active at this entity, and the CFDP implementation **does not** retain transaction history. The transaction might be one that was formerly active and has been terminated, or it might be one that has never been active at this entity.

01 – Active: The transaction to which the acknowledged PDU belongs is currently active at this entity.

10 – Terminated: The transaction to which the acknowledged PDU belongs is not currently active at this entity; the CFDP implementation **does** retain transaction history, and the transaction is thereby known to be one that was formerly active and has been terminated.

11 – Unrecognized: The transaction to which the acknowledged PDU belongs is not currently active at this entity; the CFDP implementation **does** retain transaction history, and the transaction is thereby known to be one that has never been active at this entity.

Table 2-10: ACK PDU Contents

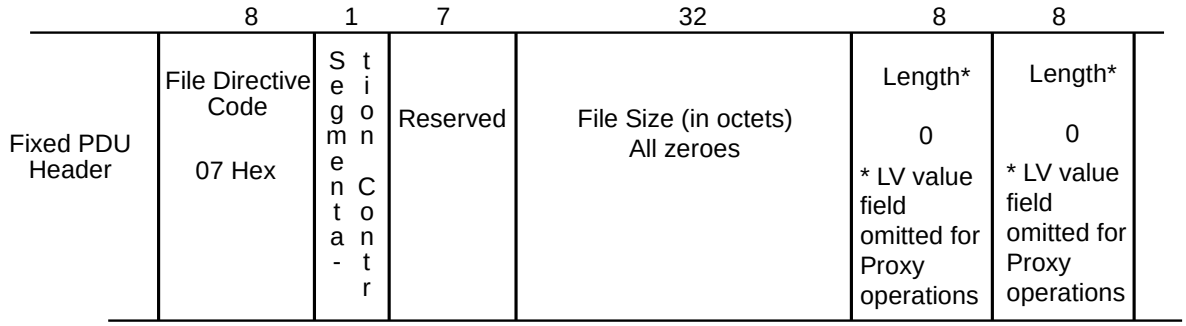
Parameter	Length (bits)	Values	Comments
Directive code	4	See table 2-2. Only EOF and Finished PDUs are acknowledged.	Directive code of the acknowledged PDU.
Directive subtype code	4		Values depend on directive code. For ACK of Finished PDU: binary 0000 if generated by waypoint, binary 0001 if generated by end system. Binary 0000 for ACKs of all other file directives.
Condition code	4	See table 2-3.	Condition code of the acknowledged PDU.
Spare	2		
Transaction status	2		Status of the transaction in the context of the entity that is issuing the acknowledgment.

3 USER OPERATIONS MESSAGE FORMATS

3.1 USER OPERATIONS

3.1.1 METADATA PDU

User Operations Messages are contained in a metadata PDU, as pictured below:



3.1.2 RESERVED CFDP MESSAGE

Each individual User Operations Message in the metadata PDU is preceded by the Reserved Message Header field, pictured below. User Operations Message types are contained in table 3-1.

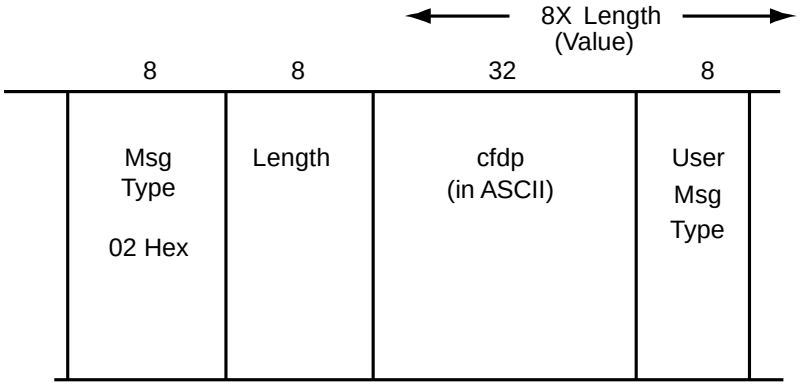


Table 3-1: User Operations Message Types

Msg Type (hex)	Interpretation
00	Proxy Put Request
01	Proxy Message to User
02	Proxy Filestore Request
03	Proxy Fault Handler Override
04	Proxy Transmission Mode
05	Proxy Flow Label
06	Proxy Segmentation Control
07	Proxy Put Response
08	Proxy Filestore Response
09	Proxy Put Cancel
10	Directory Listing Request
11	Directory Listing Response
20	Remote Status Report Request
21	Remote Status Report Response
30	Remote Suspend Request
31	Remote Suspend Response
38	Remote Resume Request
39	Remote Resume Response

3.1.3 ORIGINATING TRANSACTION ID MESSAGE

The Originating Transaction ID message is common to all categories of User Operations messages, and its format, below, is the same when used in any of the categories.

8	1	3	1	3	Variable	Variable
Msg Type 0A Hex	Reserved '0'	entity ID Length	Reserved '0'	Transaction Seq. Num Length	Source entity ID	Transaction sequence number

3.2 PROXY OPERATIONS

3.2.1 PROXY PUT REQUEST

8	8	8X Length	8	8X Length	8	8X Length
Msg Type 00 Hex	Length	(Value) Destination entity ID	Length*	(Value) Source file name	Length*	(Value) Destination file name

* Length is zero if parameter is omitted

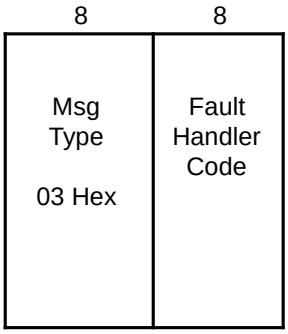
3.2.2 PROXY MESSAGE TO USER

8	8	8X Length
Msg Type 01 Hex	Length	(Value)

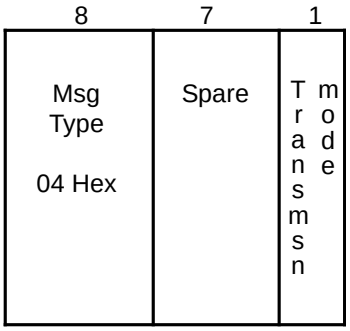
3.2.3 PROXY FILESTORE REQUEST

8	8	8X Length
Msg Type 02 Hex	Length	(Value) (A single CFDP File Store Request)

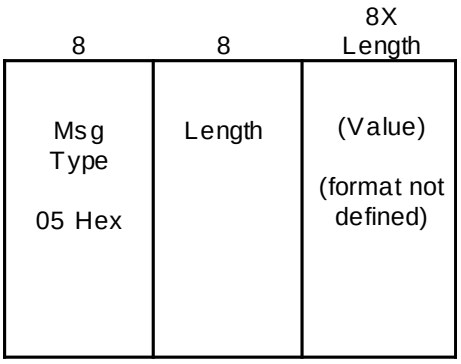
3.2.4 PROXY FAULT HANDLER OVERRIDE



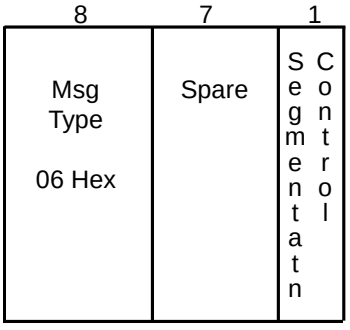
3.2.5 PROXY TRANSMISSION MODE



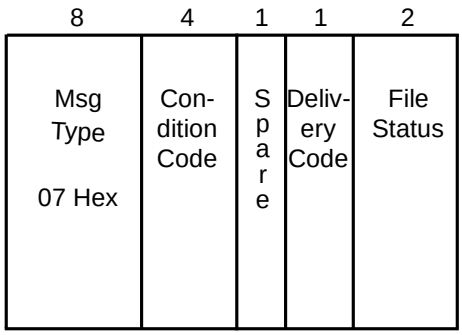
3.2.6 PROXY FLOW LABEL



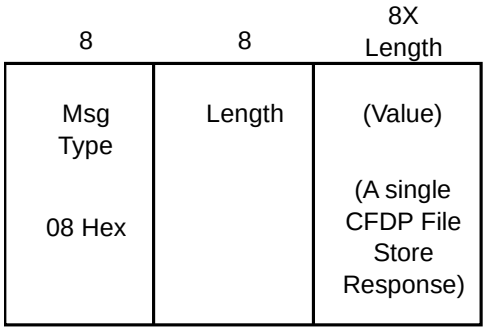
3.2.7 PROXY SEGMENTATION CONTROL



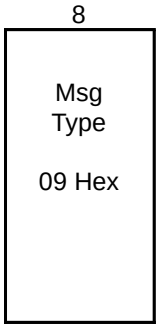
3.2.8 PROXY PUT RESPONSE



3.2.9 PROXY FILESTORE RESPONSE



3.2.10 PROXY PUT CANCEL



3.3 DIRECTORY OPERATIONS

3.3.1 DIRECTORY LISTING REQUEST

8	8	8X Length	8	8X Length
Msg Type 10 Hex	Length	(Value) Directory Name	Length	(Value) Directory File Name*

* The file name and path at the filestore local to the requesting CFDP user in which the responding CFDP user should put the directory listing

3.3.2 DIRECTORY LISTING RESPONSE

8	8	8	8X Length	8	8X Length
Msg Type 11 Hex	Listing Response Code 00-7F-Successful 80-FF-Unsuccess-ful	Length	(Value) Directory Name*	Length	(Value) Directory File Name**

*The name of the directory being listed, taken from the directory listing request

**The file name and path at the filestore local to the requesting CFDP in which the listing has been put, taken from the directory listing request

3.4 REMOTE STATUS REPORT OPERATIONS

3.4.1 REMOTE STATUS REPORT REQUEST

8	1	3	1	3	variable	variable	8	8X Length
Msg Type 20 Hex	Reserved	Entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number	Length	(Value) Report File Name

3.4.2 REMOTE STATUS REPORT RESPONSE

8	2	6	1	3	1	3	variable	variable
Msg Type 21 Hex	Transaction Status	Reserved	Reserved	Entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number

3.5 REMOTE SUSPEND OPERATIONS

3.5.1 REMOTE SUSPEND REQUEST

8	1	3	1	3	variable	variable
Msg Type 30 Hex	Reserved	entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number

3.5.2 REMOTE SUSPEND RESPONSE

8	1	2	5	1	3	1	3	variable	variable
Msg Type 31 Hex	Susp Ind	Transaction Status	Reserved	Reserved	entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number

3.6 REMOTE RESUME OPERATIONS

3.6.1 REMOTE RESUME REQUEST

8	1	3	1	3	variable	variable
Msg Type 38 Hex	Reserved	entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number

3.6.2 REMOTE RESUME RESPONSE

8	1	2	5	1	3	1	3	variable	variable
Msg Type 39 Hex	Susp Ind	Transaction Status	Reserved	Reserved	entity ID length	Reserved	Transact Seq Num length	Source entity ID	Transaction Sequence Number

3.7 STORE-AND-FORWARD OVERLAY (SFO)

3.7.1 SFO REQUEST

8	2	1	1	4	8	variable	variable	variable	variable	variable
Msg Type 40 Hex	Trace Flag Contr.	Trnsmsn Mode	Segmnt Control	Reserved	Prior Waypoints Count	SFO Request Label	Source Entity ID	Destination Entity ID	Source File Name	Destination File Name

3.7.2 SFO MESSAGE TO USER

8	variable	variable
Msg Type 41Hex	Msg to User	More Msgs to User as Needed

3.7.3 SFO FLOW LABEL

8	variable
Msg Type 42Hex	Flow Label

3.7.4 SFO FAULT HANDLER OVERRIDE

8	variable
Msg Type 43 Hex	Fault Hndlr Override Msgl

3.7.5 SFO FILESTORE REQUEST

8	8	8 X Length
Msg Type 44 Hex	Length	Filestore Request Msg

3.7.6 SFO REPORT

8	variable	variable	variable	variable	8	8	4	1	1	2
Msg Type 45 Hex	SFO Request Label	Source Entity ID	Destination Entity ID	Reporting Enitiy ID	Prior Waypoints Count	Report Code	Condition Code	Direction	Delivry Code	File Status

3.7.7 SFO FILESTORE RESPONSE

8	8	8 X Length
Msg Type 46 Hex	Length	Filestore Response

4 PROTOCOL OPTIONS, TIMERS, AND COUNTERS

4.1 OVERVIEW

This section contains implementation options, timers, and counters.

4.2 OPTIONS

Table 4-1: Options

Put Modes	Effect
UnACK	Selects Unreliable mode of operation.
NAK	Selects Reliable mode of operation.

Put NAK Modes	Effect
Immediate	NAKs are sent as soon as missing data is detected.
Deferred	NAK is sent when EOF is received.
Prompted	NAK is sent when a Prompt (NAK) is received.
Asynchronous	NAK is sent upon a local (implementation-specific) trigger at the receiving entity.

PDU CRC	Effect
True	Requires that a CRC be calculated and inserted into each File Data PDU.
False	No CRC is inserted in File Data PDUs.

Put File Types	Effect
Bounded	Sends a normal file, i.e., one in which the file is completely known before transmission.
Unbounded	Sends a file the length of which is not known when transmission is initiated (intended primarily for real-time data).

Segmentation Control (Record Boundaries Respected)	Effect
Yes	Causes each File Data PDU to begin at a record boundary.
No	Ignores record structure when building PDUs.

Table 4-1: Options (continued)

Put Primitives	Effect
EOF-sent.ind	Indicates to User at source entity that the EOF for the identified transaction was sent.
EOF-recv.ind	Indicates to User at destination entity that the EOF for the identified transaction was received (optional).
Transaction-finished.ind	Mandatory at source entity, optional at destination entity.
File-segment-receive.ind	Indicates to the user at destination entity that a File Data PDU has been received.
Transfer-consigned.indication	Indicates to the User at source entity that the identified transaction has been entrusted to the next entity (waypoint) (Extended Procedures only).

Action on Detection of a Fault	Effect
Cancel	Cancels subject transaction.
Suspend	Suspends subject transaction.
Ignore	Ignores error (but sends Fault.indication to local user).
Abandon	Abandons transaction with no further action.

Action on Cancel At Receiving End	Effect
Discard data	Discards all data received in the transaction.
Forward incomplete	Forwards all data received to the local destination.

Put Report Modes (Sending End)	Effect
Prompted Rpt	Returns report on Prompt from local user.
Periodic	Returns report to local user at specified intervals.

Release of Retransmission Buffers	Effect
Incremental and Immediate	Releases local retransmission buffer as soon as sent.
In total when ‘Finished’ Received	Releases local retransmission buffer only when Finished PDU is received.

Waypoint Forwarding Method	Effect
Incremental and Immediate	Sends received PDUs to next entity as soon as received.
In Total Upon Complete Custody Acquisition	Sends FDU to next entity only when entire FDU has been received.

4.3 TIMERS

The following should be considered relative to the use of timers:

a) At the sender, the timer for a given EOF or Finished PDU should not be started until the moment that the PDU is delivered to the link layer for transmission. All outbound queuing delay for the PDU has already been incurred at that point.

b) At the receiver, acknowledgment PDUs should always be inserted at the *front* of the priority First-In-First-Out (FIFO) list to ensure that they are transmitted as soon as possible after reception of the PDUs to which they respond. (Acknowledgment PDUs are small and are sent infrequently, so the effect on the delivery of any emergency traffic is insignificant.)

c) To account for any additional delays introduced by loss of connectivity, the implementer must rely on external link state cues. Whenever loss of connectivity is signaled by a link state queue, the timers for all PDUs destined for the corresponding remote entity should be suspended; reacquiring the link to the entity should cause those timers to be resumed. By using this method, there is no need to try to estimate connectivity loss delays in advance, and there is no need for CFDP itself to be aware of either the ephemerides or the tracking schedules of the local entity or of any remote entity.

Table 4-2: Timers

TIMER NAME	TYPE	TIMER LOCATION	STARTS ON	RESETS ON	TERMINATES ON	ACTIONS ON EXPIRY
NAK Retry Timer	Mandatory for all acknowledged modes	FDU Receiving entity	Issuance of a NAK	Issuance of a NAK	Reception of all requested data	Issue a new NAK for all unreceived data
ACK Retry Timer	Mandatory for all acknowledged modes	Entity issuing PDU to be acknowledged	Issuance of a PDU requiring positive acknowledgment	Re-issuance of the PDU	Reception of expected response	Re-issue the original PDU
Inactivity Timer *(suspended by Suspension Procedures)*	Mandatory except sending entity in unacknowledged mode	Each Source and Destination entity	Reception of any PDU	Reception of any PDU	Implementation-specific	Issue an Inactivity.in-dication

4.4 COUNTERS

Table 4-3: Counters

COUNTER NAME	TYPE	COUNTER LOCATION	COUNTER LIMIT	ACTION ON REACHING LIMIT
NAK Timer Expiration Limit	Mandatory for all acknowledged modes	FDU Receiving entity	Implementation-specific	Invoke Fault procedures
ACK Timer Expiration Limit	Mandatory for all acknowledged modes	Entity issuing PDU to be acknowledged	Implementation-specific	Invoke Fault procedures
Keep Alive Discrepancy Limit	Optional		Implementation-specific	Invoke Fault procedures

5 CFDP STATE TABLES

NOTE – Contributed by Timothy Ray, National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC).

5.1 OVERVIEW

This section provides validated logic for implementing a practical subset of the CFDP standard. This subset includes Class 1 service (i.e., Unacknowledged Mode) and the Deferred-Nak subset of Class 2 service (i.e., Acknowledged Mode). *Deferred-Nak* means that the Receiver waits for the Sender to transmit the entire file once before responding with any Naks requesting retransmission of missing data. Proxy operations (e.g., asking a partner to send a file back) are not covered here.

The core logic is contained in these state tables:

a) Class 1 Sender (S1);

b) Class 1 Receiver (R1);

c) Class 2 Sender (S2);

d) Class 2 Receiver (R2).

For any CFDP transaction that falls within the supported subset, one of the state tables will apply. For each active transaction, a *state machine* exists. For example, if a CFDP-entity has 3 active transactions for which its role is Class 2 Receiver, it will have 3 R2 state machines, each utilizing the R2 state table logic. Each state machine runs independently of any others.

The Class 2 Sender state table logic supports Deferred, Immediate, and Asynchronous Nak-modes—it will work with any Class 2 Receiver. The Class 2 Receiver state table logic supports only the Deferred Nak-mode. The state tables (tables 5-1 through 5-4) are contained in subsection 5.2, and are followed by general notes in subsection 5.3.

There is additional logic for routing each incoming PDU or User Request to the appropriate state machine, creating a new state machine for each new transaction, and maintaining the list of active state machines. This logic is called the *Kernel* logic, and is contained in subsection 5.4. State table logic runs in response to each *event* that occurs. Events are listed in subsection 5.5. For each event, the state tables specify a set of *actions* to be taken. Some actions are described in more detail in subsection 5.6. Variables used within the state tables are described in subsection 5.7.

While these state tables are not replacements for the specifications provided in the CFDP Blue Book, the logic described in this section has been implemented and validated. The implementation was connected to each of the other existing CFDP implementations, and Service Classes 1 and 2 were tested. Validation included a variety of test scenarios where data was purposely dropped, as well as suspend/resume/cancel operations.

5.2 STATE TABLES

Table 5-1: Class 1 Sender

Event:	State	
	S1 **Send Metadata**	**S2** **Send File**

E0
Entered this state
	Initialize	Open source file
		If (Open failure?)
		Fault (Filestore)
		If (Invalid file structure?)
		Fault (File structure)
		Trigger E1

E1
Please send file-data
	N/A	If (Suspended=False and
		Frozen=False)
		If (Comm layer ready?)
		Tx: one File-data
		If (Entire file sent?)
		Tx: EOF (no error)
		Issue *Transaction-Finished*
		Shutdown
		Trigger E1

E2 Abandon this transaction	N/A	Issue *Abandoned* **Shutdown**
E3 Notice of Cancellation	N/A	Tx: EOF (cancel) Issue *Transaction-Finished* **Shutdown**
E4 Notice of Suspension	N/A	If (Suspended=False) Issue *Suspended* Suspended=True

E30
Rx: Put Request
(This is the first event received)
	Issue *Transaction*
	Tx: Metadata
	If (File Transfer?)
	State=S2
	Else
	Tx: EOF (no error)
	Issue *Transaction-Finished*
	Shutdown
		N/A

E31 Rx: Suspend Request	N/A	Trigger E4

E32
Rx: Resume Request
	N/A	If (Suspended=True)
		Issue *Resumed*
		Suspended=False
		If (Frozen=False)
		Trigger E1

E33 Rx: Cancel Request	N/A	Condition='Cancel.request received' Trigger E3
E34 Rx: Report Request	N/A	Issue *report*
E40 Rx: Freeze	N/A	Frozen=True

E41
Rx: Thaw
	N/A	If (Frozen=True)
		Frozen=False
		If (Suspended=False)
		Trigger E1

Table 5-2: Class 1 Receiver

Event:	State	
	S1 **Wait for MD**	**S2** **Wait for EOF**
E0 Entered this state	Initialize	N/A
E2 Abandon this transaction	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**
E3 Notice of Cancellation	Issue *Transaction-Finished* **Shutdown**	Possibly retain temp file Issue *Transaction-Finished* **Shutdown**
E4 Notice of Suspension	N/A	N/A

E10
Rx: Metadata
(This is normally the first event received)
	Issue *Metadata-Recv*
	If (File Transfer?)
	Open temp file
	If (Open failure?)
	Fault (Filestore)
	Process Metadata TLVs
	State=S2
		N/A

E11 Rx: File-Data	N/A	If (File Transfer?) Store file-data Update Received_file_size

E12
Rx: EOF (no error)
	(Optionally, let the User know that the transaction completed without any Metadata being received)
	Issue *Transaction-Finished*
	Shutdown
		If (File Transfer?)
		Close temp file
		If (File size error?)
		Fault (File size)
		If (File checksum failure?)
		Fault (File checksum)
		Delivery=Complete
		Copy temp file to dest file
		If (Copy error?)
		Fault (Filestore)
		If (Filestore Requests?)
		Execute Filestore Requests
		Issue *Transaction-Finished*
		Shutdown

E13 Rx: EOF (cancel)	Update Condition Issue *Transaction-Finished* **Shutdown**	Update Condition Possibly retain temp file Issue *Transaction-Finished* **Shutdown**
E27 Inactivity-timeout	Start Inactivity-timer **Fault (**Inactivity)	Start Inactivity-timer **Fault (**Inactivity)
E33 Rx: Cancel Request	Condition=‘Cancel.request received’ Trigger E3	Condition=‘Cancel. request received’ Trigger E3
E34 Rx: Report Request	Issue *Report*	Issue *Report*

Table 5-3: Class 2 Sender (Immediate/Deferred/Asynchronous Nak-mode)

State: Event:	**S1** **Send *Metadata***	**S2** **Send the File Once**	**S3** **Send *EOF*; Fill Any Gaps**	**S4** **Transaction Cancelled**

E0
Entered this state
	Initialize	Open source file
		If (Open failure?)
		Fault (Filestore)
		If (Invalid file structure)
		Fault (File structure)
		Trigger E1
			Tx: EOF
			Start Ack-timer
			Start Inactivity-timer
				Suspended=False
				Tx: EOF (cancel)
				Start Ack-timer

E1
Please send file-data
	N/A	If (Suspended=False
		and Frozen=False)
		If (Comm layer ready?)
		Tx: one File-data
		If (Entire file sent?)
		State=S3
		Trigger E1
			If (Suspended=False
			and Frozen=False)
			If (File-data queued?)
			If (Comm layer ready?)
			Tx: one File-data
			Trigger E1
				N/A

E2 Abandon transaction	N/A	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**
E3 Notice of Cancellation	N/A	**State=S4**	**State=S4**	N/A

E4
Notice of Suspension
	N/A	If (Suspended=False)
		Issue *Suspended*
		Suspended=True
		If (Frozen=False)
		Trigger E5
			If (Suspended=False)
			Issue *Suspended*
			Suspended=True
			If (Frozen=False)
			Trigger E5
				If (Suspended=False)
				Issue *Suspended*
				Suspended=True
				If (Frozen=False)
				Trigger E5

E5 Suspend timers	N/A	N/A	Suspend Inactivity-timer If (Is Ack-timer running?) Suspend Ack-timer	Suspend Inactivity-timer Suspend Ack-timer
E6 Resume timers	N/A	Trigger E1	Resume Inactivity-timer If (Is Ack-timer suspended?) Resume Ack-timer Trigger E1	Resume Inactivity-timer Resume Ack-timer
E14 Rx: Ack-EOF	N/A	N/A	Cancel Ack-timer	If (Condition_code<>No_Error) Issue *Transaction-Finished* **Shutdown**
E15 Rx: Nak	N/A	If (Suspended=False and Frozen=False) Queue nakked data	If (Suspended=False and Frozen=False) Queue nakked data Trigger E1	N/A
E16 Rx: Finished (no error)	N/A	N/A	Tx: Ack-Finished Issue *Transaction-Finished* **Shutdown**	N/A
E17 Rx: Finished (cancel)	N/A	Update Condition Tx: Ack-Finished Issue *Transaction-Finished* **Shutdown**	Update Condition Tx: Ack-Finished Issue *Transaction-Finished* **Shutdown**	Update Condition Tx: Ack-Finished Issue *Transaction-Finished* **Shutdown**

E25
Ack-timeout
	N/A	N/A	Start Ack-timer
			If (Positive ack limit
			reached?)
			Fault (Ack limit)
			Tx: EOF
				Start Ack-timer
				If (Positive ack limit reached?)
				Trigger E2
				Else
				Tx: EOF

E27 Inactivity Timeout	N/A	N/A	Start Inactivity-timer **Fault** (Inactivity)	Issue *Abandoned* **Shutdown**

E30
Rx: Put Request
(This is the first event received)
	Issue *Transaction*
	Tx: Metadata
	If (File transfer?)
	State=S2
	Else
	State=S3
		N/A	N/A	N/A

State: Event:	**S1** **Send *Metadata***	**S2** **Send the File Once**	**S3** **Send *EOF*; Fill Any Gaps**	**S4** **Transaction Cancelled**
E31 Rx: Suspend Request	N/A	Trigger E4	Trigger E4	Trigger E4

E32
Rx: Resume Request
	N/A	If (Suspended)
		Issue *Resumed*
		Suspended=False
		If (Frozen=False)
		Trigger E6
			If (Suspended)
			Issue *Resumed*
			Suspended=False
			If (Frozen=False)
			Trigger E6
				If (Suspended)
				Issue *Resumed*
				Suspended=False
				If (Frozen=False)
				Trigger E6

E33 Rx: Cancel Request	N/A	Condition='Cancel.request received' Trigger E3	Condition='Cancel.request received' Trigger E3	N/A
E34 Rx: Report Request	N/A	Issue *Report*	Issue *Report*	Issue *Report*

E40
Rx: Freeze
	N/A	If (Frozen=False)
		Frozen=True
		If (Suspended=False)
		Trigger E5
			If (Frozen=False)
			Frozen=True
			If (Suspended=False)
			Trigger E5
				If (Frozen=False)
				Frozen=True
				If (Suspended=False)
				Trigger E5

E41
Rx: Thaw
	N/A	If (Frozen=True)
		Frozen=False
		If (Suspended=False)
		Trigger E6
			If (Frozen=True)
			Frozen=False
			If (Suspended=False)
			Trigger E6
				If (Frozen=True)
				Frozen=False
				If (Suspended=False)
				Trigger E6

Table 5-4: Class 2 Receiver (Deferred Nak-mode)

State: Event:	**S1** **Wait for *EOF***	**S2** **Get Missing Data**	**S3** **Send *Finished* and Confirm Delivery**	**S4** **Transaction Cancelled**

E0
Entered this State
	Initialize	If (Suspended=False
		And Frozen=False)
		Tx: Nak
		Start Nak-timer
			Delivery=Complete
			Cancel Nak-timer
			If (File transfer?)
			Close temp file
			If (File checksum failure?)
			Fault (File checksum)
			Copy temp file to dest file
			If (Copy error?)
			Fault (Filestore)
			If (Filestore Requests?)
			Execute filestore requests
			Tx: Finished (no error)
			Start Ack-timer
				Suspended=False
				If (Previous_state<>S3)
				Possibly retain temp file
				Tx: Finished (cancel)
				Start Ack-timer

E2 Abandon this transaction	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**	Issue *Abandoned* **Shutdown**
E3 Notice of Cancellation	**State=S4**	**State=S4**	**State=S4**	N/A

E4
Notice of Suspension
	If (Suspended=False)
	Issue *Suspended*
	Suspended=True
	If (Frozen=False)
	Trigger E5
		If (Suspended=False)
		Issue *Suspended*
		Suspended=True
		If (Frozen=False)
		Trigger E5
			If (Suspended=False)
			Issue *Suspended*
			Suspended=True
			If (Frozen=False)
			Trigger E5
				If (Suspended=False)
				Issue *Suspended*
				Suspended=True
				If (Frozen=False)
				Trigger E5

E5 Suspend timers	Suspend Inactivity-timer	Suspend Inactivity-timer	Suspend Inactivity-timer Suspend Ack-timer	Suspend Inactivity-timer Suspend Ack-timer
E6 Resume timers	Resume Inactivity-timer	Resume Inactivity-timer	Resume Inactivity-timer Resume Ack-timer	Resume Inactivity-timer Resume Ack-timer

State: Event:	**S1** **Wait for *EOF***	**S2** **Get Missing Data**	**S3** **Send *Finished* and Confirm Delivery**	**S4** **Transaction Cancelled**

E10
Rx: Metadata
(This is normally the first event received)
	Reuse Senders first PDU header
	If (Metadata_Received = False)
	  Metadata_Received=True
	  Issue *Metadata-Recv*
	  If (File Transfer?)
	  If (File_Open=False)
	  Open temp file
	  If (Open failure?)
	  Fault (Filestore)
	  File_Open=True
	  Update Nak-list
	  Process Metadata TLVs
		If (Metadata_Received=False)
		  Metadata_Received=True
		  Issue *Metadata-Recv*
		  If (File Transfer?)
		  If (File_Open=False)
		  Open temp file
		  If (Open failure?)
		  Fault (Filestore)
		  File_Open=True
		  Update Nak-list
		  Process Metadata TLVs
			N/A	N/A

E11
Rx: File-Data
	Reuse Senders first PDU header
	If (File_Open=False)
	  Open temp file
	  If (Open failure?)
	  Fault (Filestore)
	  File_Open=True
	Store file-data
	Update Received_file_size
	Update Nak-list
		If (File_Open=False)
		  Open temp file
		  If (Open failure?)
		  Fault (Filestore)
		  File_Open=True
		Store file-data
		Update Received_file_size
		Update Nak-list
		If (File size error?)
		  Fault (File size error)
			N/A	N/A

E12
Rx: EOF (no error)
	Reuse Senders first PDU header
	Update Nak-list
	Tx: Ack-EOF
	If (File size error?)
	  Fault (File size error)
	If (Is Nak-list empty?)
	  State=S3
	Else
	  State=S2
		Tx: Ack-EOF	Tx: Ack-EOF	N/A

E13 Rx: EOF (cancel)	Reuse Senders first PDU header Update Condition Tx: Ack-EOF Possibly retain temp file Issue *Transaction-Finished* **Shutdown**	Update Condition Tx: Ack-EOF Possibly retain temp file Issue *Transaction-Finished* **Shutdown**	Update Condition Tx: Ack-EOF Issue *Transaction-Finished* **Shutdown**	Update Condition Tx: Ack-EOF Issue *Transaction-Finished* **Shutdown**

E18
Rx: Ack-Finished
	N/A	N/A	Issue *Transaction-Finished*
			Shutdown
				If (Condition_code<>No_Error)
				  Issue *Transaction-Finished*
				  Shutdown

State: Event:	**S1** **Wait for *EOF***	**S2** **Get Missing Data**	**S3** **Send *Finished* and Confirm Delivery**	**S4** **Transaction Cancelled**

E25
Ack-timeout (i.e., Partner has not responded)
	N/A	N/A	Start Ack-timer
			If (Positive ack limit reached?)
			Fault (Ack limit)
			Tx: Finished (no error)
				Start Ack-timer
				If (Positive ack limit reached?)
				Trigger E2
				Else
				Tx: Finished (cancel)

E26
NAK-timeout (i.e., Periodic feedback to partner)
	N/A	Start Nak-timer
		If (Is Nak-list empty?)
		State=S3
		Else if (Suspended=False and Frozen=False)
		If (Nak limit reached?)
		Fault (Nak limit)
		Tx: Nak
			N/A	N/A

E27 Inactivity Timeout	Restart Inactivity-timer **Fault** (Inactivity)	Restart Inactivity-timer **Fault** (Inactivity)	Restart Inactivity-timer **Fault** (Inactivity)	Issue *Abandoned* **Shutdown**
E31 Rx: Suspend Request	Trigger E4	Trigger E4	Trigger E4	Trigger E4

E32
Rx: Resume Request
	If (Suspended=True)
	Issue *Resumed*
	Suspended=False
	If (Frozen=False)
	Trigger E6
		If (Suspended=True)
		Issue *Resumed*
		Suspended=False
		If (Frozen=False)
		Trigger E6
			If (Suspended=True)
			Issue *Resumed*
			Suspended=False
			If (Frozen=False)
			Trigger E6
				If (Suspended=True)
				Issue *Resumed*
				Suspended=False
				If (Frozen=False)
				Trigger E6

E33 Rx: Cancel Request	Condition='Cancel.request received' Trigger E3	Condition='Cancel.request received' Trigger E3	Condition='Cancel.request received' Trigger E3	N/A
E34 Rx: Report Request	Issue *Report*	Issue *Report*	Issue *Report*	Issue *Report*

E40
Rx: Freeze
	If (Frozen=False)
	Frozen=True
	If (Suspended=False)
	Trigger E5
		If (Frozen=False)
		Frozen=True
		If (Suspended=False)
		Trigger E5
			If (Frozen=False)
			Frozen=True
			If (Suspended=False)
			Trigger E5
				If (Frozen=False)
				Frozen=True
				If (Suspended=False)
				Trigger E5

E41
Rx: Thaw
	If (Frozen=True)
	Frozen=False
	If (Suspended=False)
	Trigger E6
		If (Frozen=True)
		Frozen=False
		If (Suspended=False)
		Trigger E6
			If (Frozen=True)
			Frozen=False
			If (Suspended=False)
			Trigger E6
				If (Frozen=True)
				Frozen=False
				If (Suspended=False)
				Trigger E6

5.3 STATE TABLE NOTES

5.3.1 These tables provide the logic for implementing a subset of the CFDP standard. All required behavior for Service Classes 1 and 2 is provided. The Class 2 Sender state table logic supports Deferred, Immediate, and Asynchronous Nak-modes—it will work with any Class 2 Receiver. The Class 2 Receiver state table logic supports only the Deferred Nak-mode.

5.3.2 Generally, these state tables include the minimum required behavior. An implementer is free to add optional behavior as desired. (One example: a Receiver may issue a File-Segm-Recv indication for each File-data PDU received. Another example: the Prompt-Keepalive and Keepalive PDUs may be used.)

5.3.3 The state tables specify which PDU(s) are to be issued in response to each possible event. The details concerning *how* these PDUs are built are left out. For example, if a state table specifies *Tx: EOF*, this means to generate an EOF PDU and then transmit it. Check the protocol specification for formatting details. (In order to build all the outgoing PDUs, it will be necessary to store information from some of the incoming PDUs.)

5.3.4 The *method* used to pass PDUs to the lower communications layer is an implementation issue. The state table logic assumes that File Directive PDUs are output immediately and File Data PDUs are queued (and released one at a time as the lower communications layer is ready—see Event 1). If desired, an implementer can use a different method.

5.3.5 The set of actions taken in response to an event is not to be interrupted. For example, if a User Request arrives while an incoming EOF is being responded to, the response to the EOF must complete before the response to the User Request begins.

5.3.6 Where ‘if’ statements are used in the state tables, indentation is used to indicate which lines are covered by each clause.

5.3.7 The Inactivity Timer must be reset each time a PDU is received. This is not shown in the state tables, but must be performed. If the Inactivity Timer is not currently suspended, it must also be restarted each time a PDU is received (i.e., a fresh countdown begins).

5.3.8 ‘Ack limit reached’ and ‘Nak limit reached’ are implementation-dependent conditions. The state tables show the concept of using these limits, but each implementer must fill in the details.

NOTE – If a particular event is not included in a state table, then no action is required.

5.3.9 Regarding E11+S2 of the Class 2 Receiver state table: ‘File size error’ occurs when the offset of the received File-data extends beyond the File Size specified in the initial EOF (no error) PDU. For example, if the EOF PDU specifies a File Size of 1200, and File-data arrives with an offset of 1000 and length of 500, then a ‘File size error’ Fault will occur.

5.4 KERNEL

5.4.1 The kernel keeps a list of active state machines. For each state machine, the kernel keeps track of which transaction it is assigned to, its role (e.g., Class 2 Sender), and its current state (S1, S2, …, or *Completed*). Each state machine starts in state S1; when finished, it sets its state to *Completed.* The kernel also receives all incoming PDUs and User Requests, and decides what action to take.

5.4.2 Kernel logic for incoming PDUs:

Take note of the Mode (Unack or Acknowledged)--If an entity does not provide both Class 1 and Class 2 service, then the kernel must check for an *Invalid Transmission Mode* fault; see the protocol specification for details.

Take note of the Direction (Toward Sender or Toward Receiver).

Determine which transaction the PDU references.

If (an active state machine is assigned to that transaction AND has the proper role),

If (that state machine's state is not *Completed*),

Deliver the PDU to that state machine.

Else

Remove that state machine from the list of active state machines.

Consult table 5-5.

Else

Consult table 5-5.

Table 5-5: Kernel Actions for Incoming PDUs

PDU type	Direction=Toward_Sender And Mode=Unack	Direction=Toward_Sender And Mode=Acked	Direction=Toward_Receiver
Metadata	Ignore	Ignore	Start new machine*
File-data	Ignore	Ignore	Start new machine*
EOF	Ignore	Ignore	Start new machine*
Ack	Ignore	Ignore	Ignore
Nak	Ignore	Ignore	Ignore
Fin	Ignore	Send an Ack-Fin	Ignore

**Start new machine* means start a new state machine and deliver the PDU to it.*

Kernel logic for incoming User Requests:

If (Request is a Put Request)

Start a new state machine (S1 or S2, as appropriate) and deliver the Put Request to it.

Else

Take note of which transaction the Request references:

If (an active state machine is assigned to that transaction),

If (that state machine's state is not *Completed*),

Deliver the Request to that state machine.

Else

Remove that state machine from the list of active state machines.

Ignore the Request.

Else

Ignore the Request.

5.5 EVENTS

NOTE – A single set of events is defined (i.e., event E31 is the same for all state tables). Most events are delivered to the state tables. *Derived* events are triggered from within a state table.

5.5.1 DERIVED EVENTS

E0 - Entered this state. This event is implicit whenever a state change occurs.

E1 - Please send some file-data. This allows File-data to be metered out one PDU at a time.

E2 - Abandon this transaction.

E3 - Notice of Cancellation.

E4 - Notice of Suspension.

E5 - Suspend timers.

E6 - Resume timers.

5.5.2 RECEIVED A PDU

E10 - Metadata

E11 - File-data

E12 - EOF (no error)

E13 - EOF (cancel)

E14 - Ack-EOF

E15 - Nak

E16 - Finished (no error)

E17 - Finished (cancel)

E18 - Ack-Finished

5.5.3 TIMEOUT

E25 - Ack-timeout

E26 - Nak-timeout

E27 - Inactivity-timeout

5.5.4 RECEIVED A USER REQUEST

E30 - Put

E31 - Suspend

E32 - Resume

E33 - Cancel

E34 - Report

5.5.5 OTHER EVENTS

E40 - Freeze

E41 - Thaw

5.6 ACTIONS

5.6.1 OVERVIEW

For each possible event, the state tables specify a set of actions to be taken. Some actions are described in more detail, as follows:

a) …*temp file…* *(e.g., Open temp file, Copy temp file to dest file, Close temp file):* The temp file is a concept; it is a place to store incoming file-data until the Receiver decides whether or not to accept the file. How this is accomplished is implementation-dependent. Incoming file-data is stored in the temp file until the file is accepted; then the data is stored in the file specified by the Destination-File-Name field in the Metadata PDU.

b) …*Nak-list…* *(e.g., Update Nak-list):* The Nak-list is a concept. All Receivers must keep track of which data has been received and which data is missing. The state tables call that information the Nak-list. The method used is implementation-dependent.

c) *Comm layer ready?:* The state tables assume that File Directive PDUs are output 'immediately', and that File-data PDUs are output one at a time 'when the communication layer is ready'. This mechanism is not required by the protocol; implementers can use a different mechanism if they care to.

d) *Execute Filestore Requests:* Execute any Filestore Requests that were present in the Metadata PDU.

e) *Fault:* A fault was detected; take action as specified by the Fault Handler Table. See subsection 5.6.2 for details. Some fault names are shortened in the state tables, as follows:

1) Positive ACK limit reached → Ack limit;

2) Filestore rejection → Filestore;

3) File checksum failure → File checksum;

4) File size error → File size;

5) Nak limit reached → Nak limit;

6) Inactivity detected → Inactivity;

7) Invalid file structure → File structure.

f) *File size error?:* If the Received File Size (see *Update received_file_size*) is greater than the File Size referenced in the EOF PDU, then there is a *File Size Error*.

g) *Initialize:* Condition=No_Error, Delivery=Incomplete, Frozen=False, Metadata_received=False, Pdu_Received=False, Suspended=False, Initialize the *Nak-list*, Load MIB parameters (e.g., the Fault Handler Table, Ack-timeout, Nak-timeout), Start the Inactivity-timer (Class 1 Receiver and Class 2 Receiver only).

h) *N/A:* No action is required.

i) *Possibly retain temp file:* This action occurs when a transaction is not fully successful. The protocol allows the file-data to be retained, if desired. See the protocol specification for details.

j) *Process Metadata TLVs:* If there are any TLV (type-length-value) items included in the Metadata PDU, then handle them as described in the protocol specification. For example, User Messages are passed to the User immediately; Fault Handler Overrides are used to update the Fault Handler Table immediately, and Filestore Requests are stored for later execution.

k) *Queue nakked data:* Resend any nakked Metadata immediately; queue any nakked File-data for release (how this is done is implementation-dependent).

l) *Reuse Senders first PDU header:* If (Pdu_Received=False), then:

   1) store a copy of the PDU-header from this incoming PDU;

   2) reverse the 'Direction' field;

      NOTE – Use this header as the PDU-header for all outgoing PDUs.

   3) Pdu_Received=True.

m) *Shutdown:* Cancel all timers; close any open files (and/or release all file buffers); set the state to *Completed*. No action is taken in the *Completed* state; therefore, it is not shown in the state tables.

n) *Store file-data:* Store any new incoming file-data in the *temp file* (discard any File-data that has already been received).

o) *Update Condition:* Update the internal variable *Condition.* When an EOF (cancel) or Finished (cancel) is received, the condition is copied from the *Condition Code* field in the incoming PDU.

p) *Update received_file_size:* Keep track of the highest file-offset within all the file-data received during this transaction (i.e., what is the size of the received file?).

5.6.2 FAULTS

5.6.2.1 Overview

The specific faults that can occur are defined in the protocol specification. The response to each fault is contained in the Fault Handler Table. The Fault Handler Table contents are specified in the MIB, and can be overridden by an incoming Metadata PDU. The possible responses to a fault are defined in the protocol specification. They are *ignore, suspend, cancel, or abandon.*

5.6.2.2 Responding to Faults

When a fault occurs, follow this logic:

Look in the Fault Handler Table to see what response is specified.

If (response=*ignore*)

Simply continue.

Else if (response=*suspend*)

Trigger event E4, and do not perform any remaining actions within the current event.

Else if (response=*cancel*)

Set the internal variable *Condition* to whichever fault occurred.

Trigger event E3, and do not perform any remaining actions within the current event.

Else if (response=*abandon*)

Set the internal variable *Condition* to whichever fault occurred.

Trigger event E2, and do not perform any remaining actions within the current event.

5.7 INTERNAL VARIABLES

NOTE – In addition to those variables shown here, there are a variety of MIB parameters (e.g., Ack_timeout, Nak_timeout, etc). Check the protocol specification for details on MIB parameters.

Internal variables are as follows:

a) *Condition:* Equivalent to the *Condition Code* shown in the protocol specification (see the section on PDU formats). Typical values are 'No Error' or 'Cancel.request received'.

b) *Delivery:* Equivalent to the *Delivery Code* shown in the protocol specification (see the Finished PDU format). The value is either 'Complete' (all data received) or 'Incomplete' (some data missing).

c) *File_Open:* Indicates whether or not a temp file has been opened.

d) *Frozen:* Indicates whether or not the transaction is currently frozen.

e) *Metadata_Received:* Indicates whether or not a Metadata PDU has been received.

f) *Pdu_Received:* Indicates whether or not a PDU has been received.

g) *Suspended:* Indicates whether or not the transaction is currently suspended.

6 AN SDL/GRAPHICAL REPRESENTATION OF CFDP STATE DIAGRAMS

NOTE – Contributed by Hiroaki Miyoshi, National Space Development Agency (NASDA)/NEC Toshiba Space Systems.

6.1 PURPOSE AND SCOPE

This section provides state diagrams of the CCSDS CFDP Entities using the International Telecommunication Union (ITU) Specification and Description Language (SDL) graphical representation technique (reference [4]). These representations are not intended as replacements for the natural-language specifications provided in the CFDP Blue Book, but as a pilot boat to navigate specifications for implementers.

These representations describe procedures for Class 1-Source, Class 1-Destination, Class 2-Source and Class 2-Destination entities with the following MIB settings:

a) Immediate NAK mode enabled - *no*;

b) Prompt NAK mode enabled - *no*;

c) Asynchronous NAK mode enabled - *no*;

d) CRCs required on transmission - *false*.

6.2 STATE DIAGRAM TERMINOLOGY

6.2.1 INTERFACES

'UI' - Interface for CFDP SERVICE PRIMITIVES (see reference [1], subsection 3.5).

'UT' - Interface for CFDP PDUs (see reference [1], section 5).

6.2.2 VARIABLES

See subsection 5.7.

6.2.3 TIMER OPERATIONS

'start' – preset and start (or restart) an ACK, Inactivity or NAK timer (see reference [1], subsection 4.1.6.4).

'cancel' – reset and stop an ACK, Inactivity or NAK timer (see reference [1], subsection 4.1.11).

'Suspend' – suspend an ACK or Inactivity timer (see reference [1], subsection 4.1.11).

'Resume' – resume an ACK or Inactivity timer (see reference [1], subsection 4.1.6.7).

6.2.4 DECISIONS

See subsection 5.6.

6.2.5 PROCEDURES

See subsections 5.3, 5.4 and 5.6.

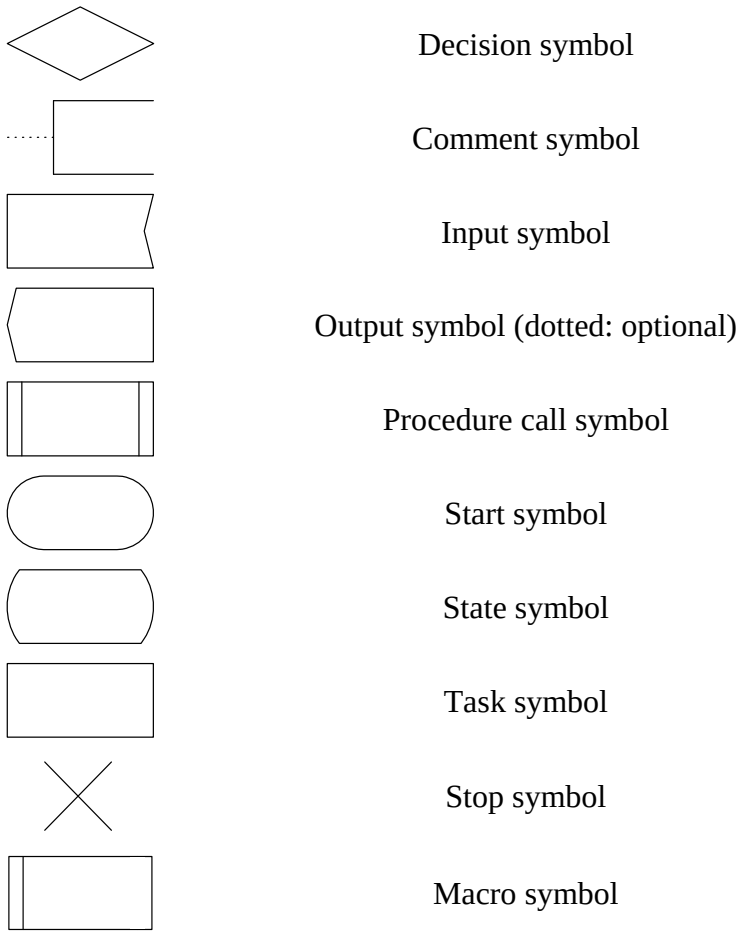
6.2.6 ABBREVIATIONS

'XN' – the abbreviation for a 'Transaction'.

'Filestore reqs?' – the abbreviation for 'Filestore Requests?'.

6.3 GRAPHICAL SYMBOL CONVENTION

This subsection contains a summary of graphical symbols used in the state diagrams depicted in figures 6-1 through 6-4. Detailed information about symbols is described in the SDL recommendations.



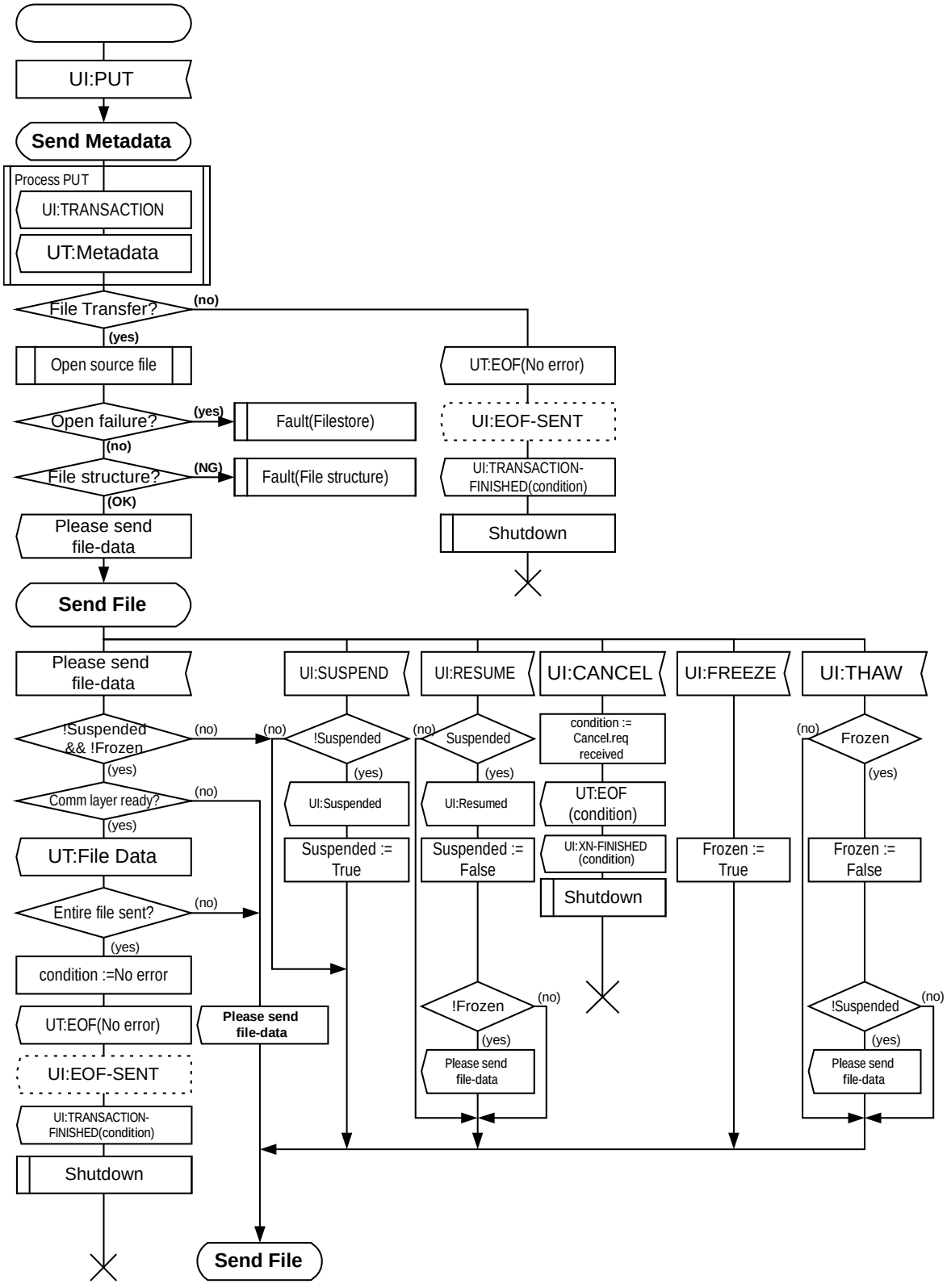


Figure 6-1: Class 1 Source State Diagram

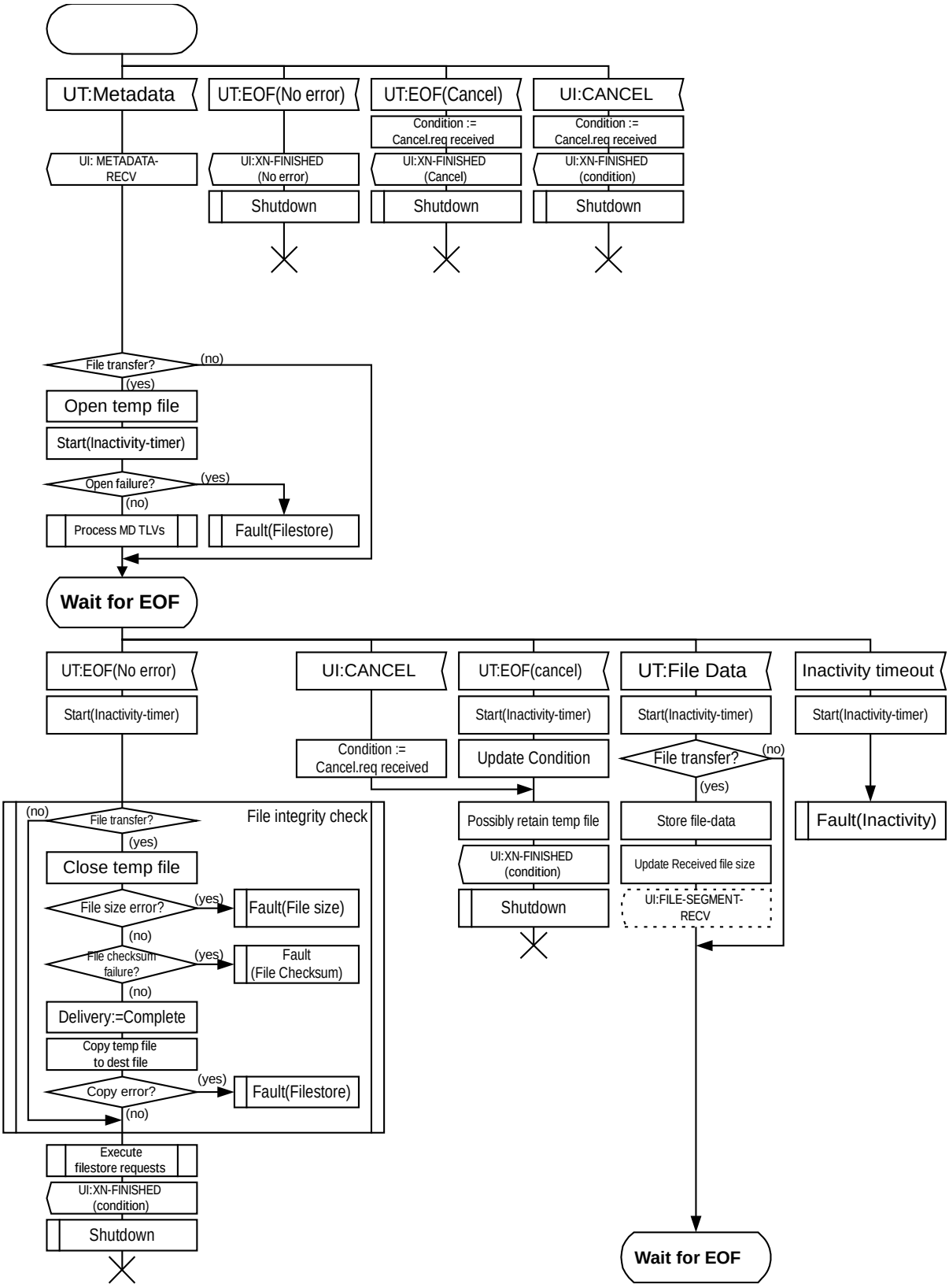


Figure 6-2: Class 1 Destination State Diagram

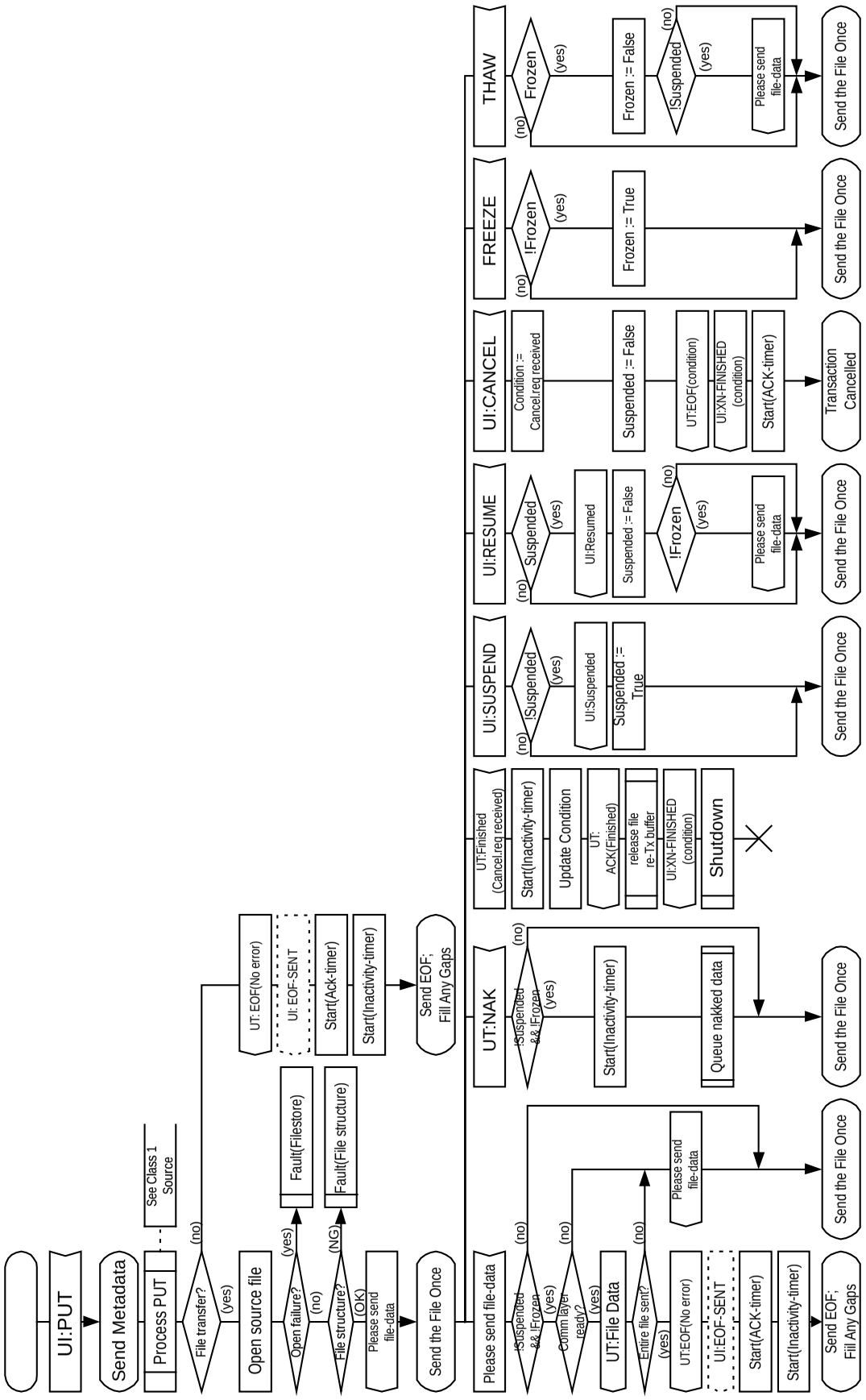


Figure 6-3: Class 2 Source State Diagram

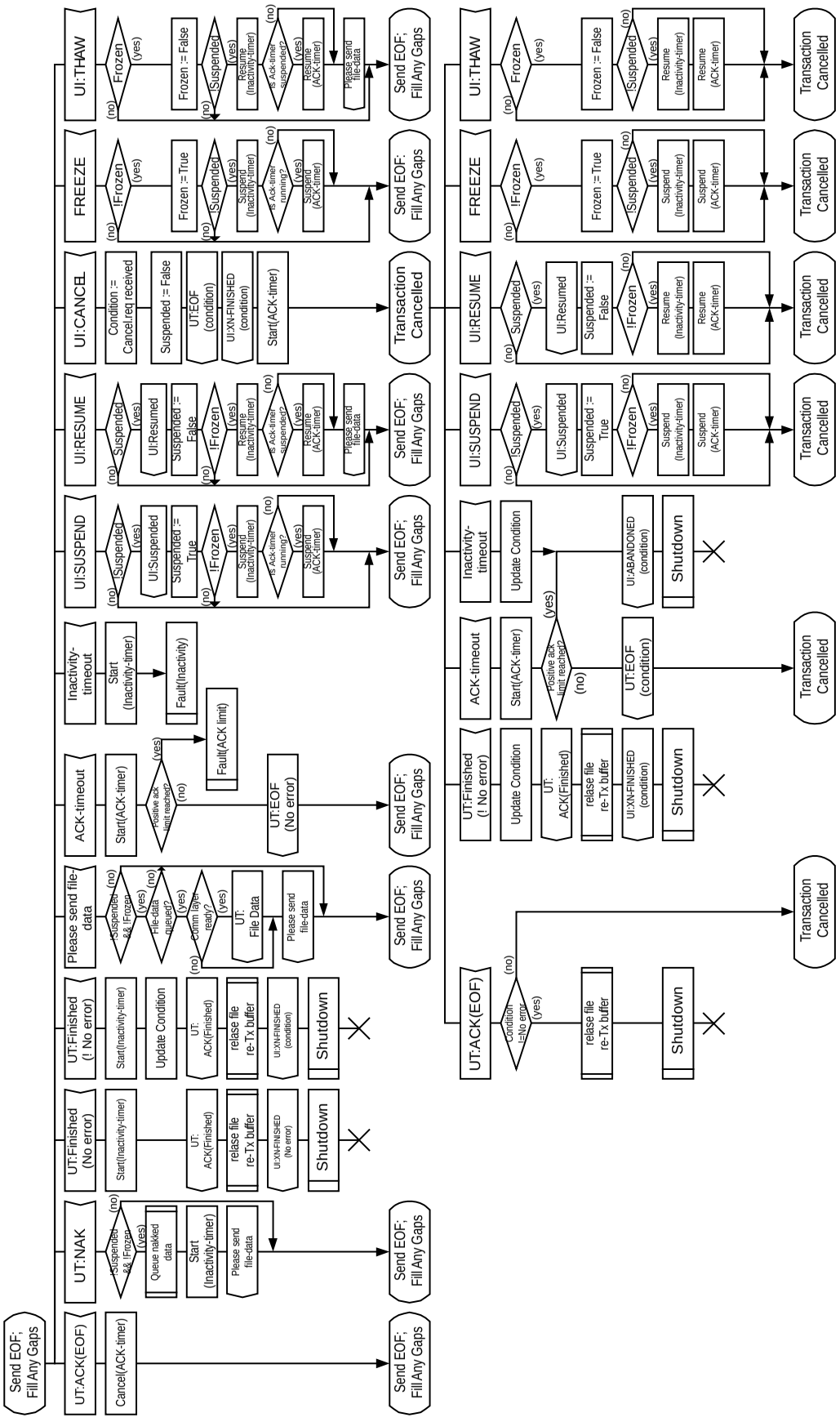


Figure 6-3: Class 2 Source State Diagram (continued)

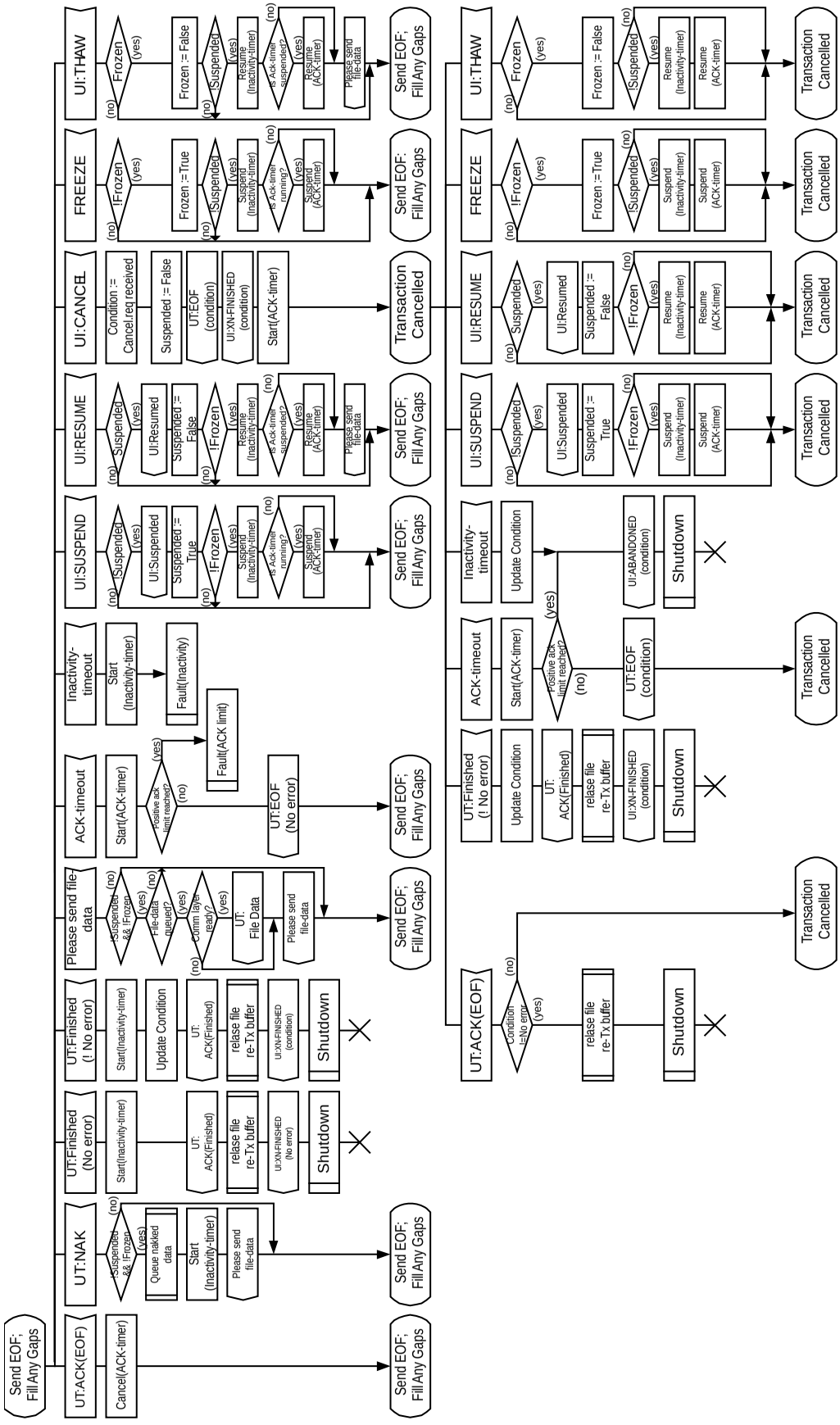


Figure 6-4: Class 2 Destination State Diagram

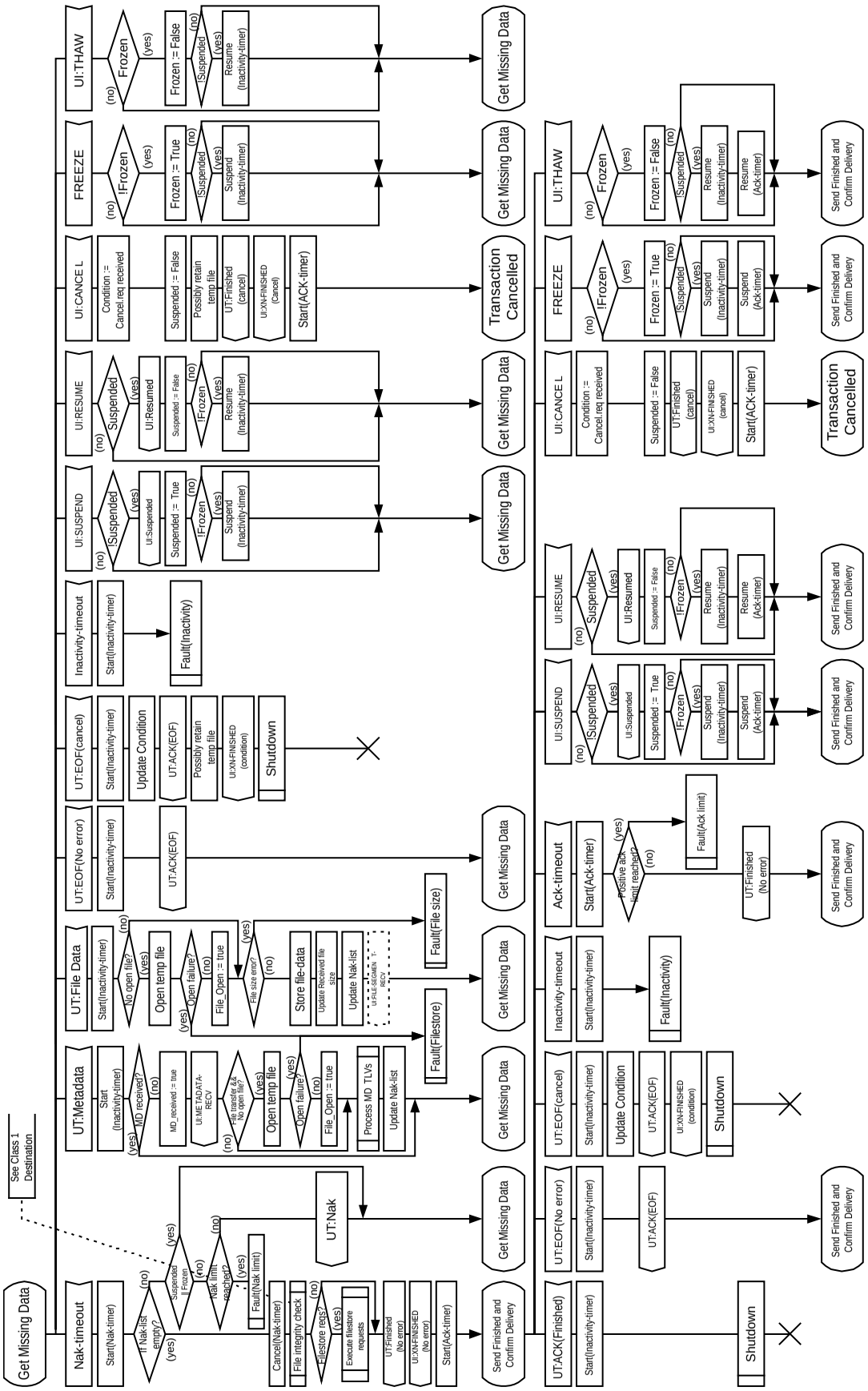


Figure 6-4: Class 2 Destination State Diagram (continued)

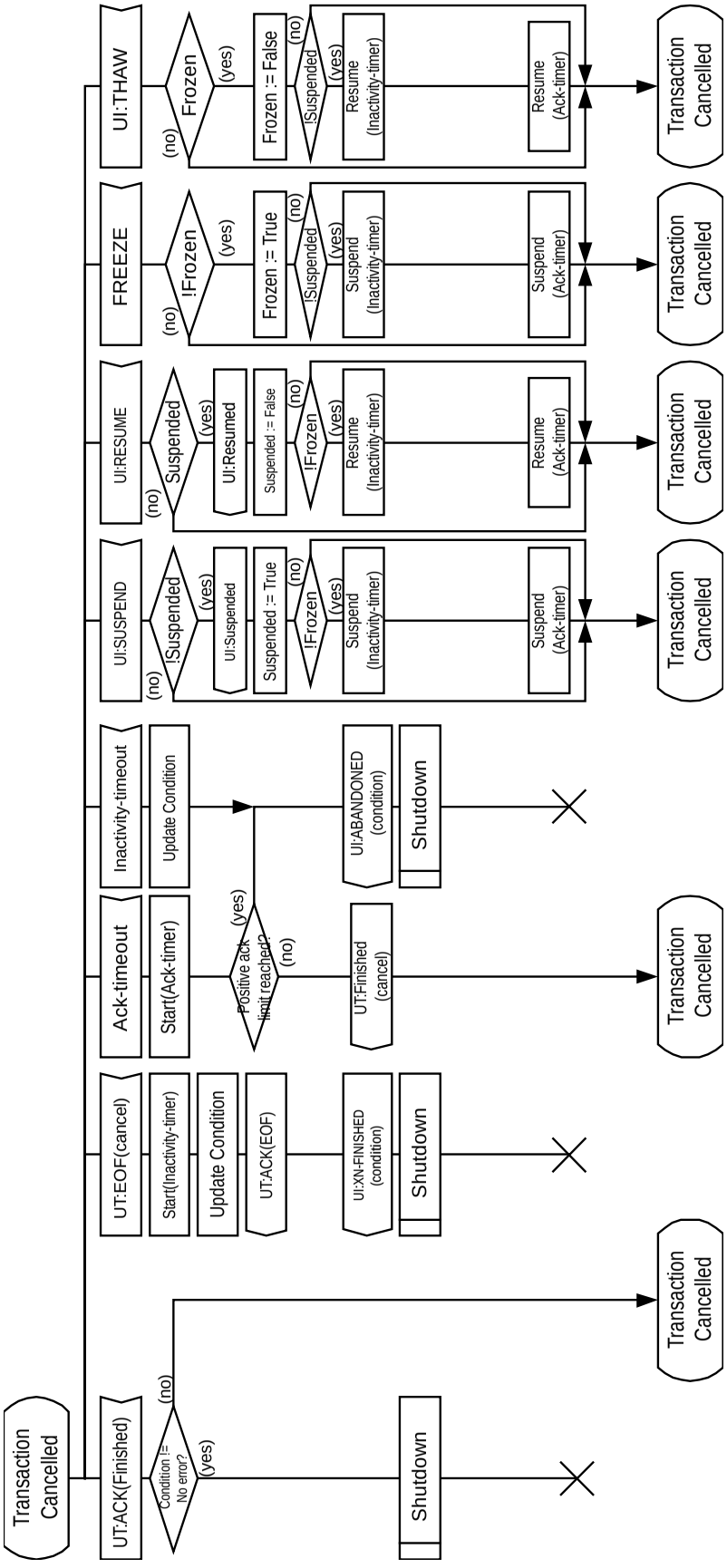


Figure 6-4: Class 2 Destination State Diagram (continued)

7 IMPLEMENTATION CONSIDERATIONS

7.1 OVERVIEW

The CFDP protocol was designed to provide file delivery services in a wide variety of space missions which were derived from a series of representative, but generic, scenarios.

In the context of a specific mission, many considerations can affect the way that CFDP services will be requested and solicited. For example:

a) mission analysis;

b) system requirements (reliable/unreliable transfers, autonomy, and transfer initiative management);

c) spacecraft orbit and visibility (Low Earth Orbit [LEO], Geosynchronous Earth Orbit [GEO], Geosynchronous Transfer Orbit [GTO], and Deep Space);

d) onboard data handling capabilities;

e) ground stations density (disjoint/overlapping passes);

f) ground segment connectivity (bandwidth limitation);

g) ground segment topology and interfaces (functional distribution, reusability of existing components, compatibility issues);

h) operational requirements (pass management, ground station availability);

Such considerations may lead to the selection of specific classes or subsets of the CFDP (e.g., reliable or unreliable modes of data transmission). In order that the protocol may successfully operate in any particular mission environment, it must be complemented by implementation-specific information and enabling mechanisms.

7.2 IMPLEMENTATION NOTES

NOTE – Subsections 7.2.1 through 7.2.3 all refer to reference [1].

7.2.1 The action taken upon detection of a File Checksum Error or of a File Size Error need not necessarily entail discarding the delivered file. The default handler for File Size Error faults may be Ignore, causing the discrepancy to be announced to the user in a **`Fault.indication`** but permitting the completion of the Copy File procedure at the receiving entity. This configuration setting might be especially appropriate for transactions conducted in unacknowledged mode.

7.2.2 In reference to Completion Procedures at the Receiving Entity, it should be noted that whether the incomplete data are retained even if the Metadata PDU has not been received, and therefore the Destination file name is unknown, is implementation-specific.

7.2.3 The 8-bit Listing Response Code in the Directory Listing Response record gives implementers the option of providing detailed and informative response codes that might be specific to particular implementations of filestore functionality, e.g., to identify specific types of directory structure corruption.

7.3 TRANSFERRING SUPPORTING INFORMATION

During the CFDP design phase, considerable effort was deployed to avoid an exponential expansion of the number of optional parameters carried by CFDP PDUs. To reduce complexity, CFDP is intentionally restricted to a minimum set of primitives sufficient to achieve its primary objective of transferring files.

In situations where it is necessary to convey CFDP-related information to a remote system, the information is propagated outside of the CFDP protocol.

Basically, three alternative 'bypass' solutions are suggested:

a) CFDP may be used to transfer a 'message to user' using a metadata PDU for an FDU that does not contain file data. The message will be passed to the CFDP user and from there it may be conveyed to a local application using implementation-specific mechanisms. This 'user to user' pass-through interface can be used to deliver a mission-specific directive or option. For example: 'suspend transaction number X in 6 minutes then auto resume this transaction in 7 hours and 35 minutes' is the kind of macro directive *not* supported by CFDP, but which can be carried by CFDP to an appropriate application via the CFDP 'message to user'.

b) CFDP may be used to transfer a file with an associated message to user. For example, 'here is a file containing pass schedules for next 10 days'.

c) CFDP is not the only way to communicate with the remote system, and any alternative interface (Telecommand [TC] or Telemetry [TM] packet) can be used to carry unsupported CFDP features. For example, 'this packet means that remote CFDP is momentarily off, due to an onboard reconfiguration'.

Bypass and proprietary solutions should only be used when basic CFDP services are not able to provide the required function.

7.4 EXAMPLE FILE CHECKSUM CALCULATION

NOTE – Contributed by Hiroaki Miyoshi, NASDA/NEC.

7.4.1 SPECIFICATIONS

As specified in reference [1], the checksum shall be 32 bits in length and calculated by the following method:

a) it shall initially be set to all 'zeroes';

b) it shall be calculated by modulo 2^{32} addition of all 4-octet words, aligned from the start of the file;

c) each 4-octet word shall be constructed by copying into the first (high-order) octet of the word, some octet of file data whose offset within the file is an integral multiple of 4, and copying the next three octets of file data into the next three octets of the word;

d) the results of the addition shall be carried into each available octet of the checksum unless the addition overflows the checksum length, in which case carry shall be discarded.

7.4.2 EXAMPLE

NOTE – This subsection contains an example of creating the checksum, developed by NASDA.

The checksum is calculated by modulo 2^{32} addition of 4-octet integers. The integers are constructed from 4-octet sets aligned from the start of the file. Each set is converted to an integer by placing its first octet in the leftmost octet of the integer, and so on, up to the fourth octet which is placed in the rightmost octet. The integer is then added to the 4-octet running total, ignoring addition overflow.

Octets may be omitted, either because file segments arrive out of order or because the file size is an inexact multiple of 4, i.e., 32 bits. Missing octets may be substituted with zeroes for the purposes of checksum calculation, as addition is commutative.

Worked example:

a. Consider a 10-byte file:

0x8a	0x1b	0x37	0x44	0x78	0x91	0xab	0x03	0x46	0x12

b. The checksum calculation is:

```
      0x8a1b3744   Bytes 0-3
+     0x7891ab03   Bytes 4-7
    ------------
    0x102ace247
&     0xffffffff   Modulo 2^32, clear carry flag
    ------------
      0x02ace247
+     0x46120000   Bytes 8-9, padded with trailing zeroes
    ------------
      0x48bee247   Final checksum, carry flag not set
```

7.5 JPL NOTES ON CFDP IMPLEMENTATION

NOTE – Contributed by Scott Burleigh, NASA/JPL.

7.5.1 OVERVIEW

The Jet Propulsion Laboratory's (JPL) implementation of CFDP has been aimed at reducing the need for active management of the protocol to the lowest level possible, in the expectation that maximizing protocol agent autonomy will help minimize the cost of operating complex deep space missions (the Mars program, for example). Here is a discussion of several design approaches embodied in that implementation which other implementers might (or might not) find useful.

7.5.2 DEFERRED TRANSMISSION

Deferred transmission can offer a degree of convenience to applications: it simplifies applications by relieving them of the need to know when communication links are active.

Deferred transmission makes CFDP responsible for scheduling file delivery to various other CFDP entities. Two implementation measures support this:

a) First, the function of responding to application requests for file delivery is partitioned from the function of handing data to the link layer for transmission; the former is handled by the *fdpd* (FDP daemon) task, the latter by a separate *fdpo* (FDP output) task (fdpd is always running, but fdpo runs only while the communication link to a specific CFDP entity is active). In response to application requests, fdpd constructs CFDP PDUs and enqueues them in **persistent FIFOs** (linked lists) of data destined for the designated entities; separately, fdpo dequeues PDUs from those FIFOs and passes them on to the underlying communication system for immediate radiation. The FIFOs grow while links are inactive, and shrink while they are active, but this is transparent to applications.

b) Second, the implementation fully supports the 'link state change' procedures by communicating these changes to fdpo. CFDP itself is just a communication protocol, not an operating system; in order for the host of the CFDP entity (spacecraft, ground station, whatever) to be able to use CFDP for communication, the host itself must establish the communication links that CFDP will use. Some mechanism—e.g., scheduled tracking passes, beacon response, or some combination of both—must therefore exist for commanding the host to establish and break those links. This implies that knowledge of link state already exists outside of CFDP, so delivery of that knowledge to CFDP can be used to control fdpo tasks. In the case of a single entity that can communicate with multiple remote entities, those external *link state cues* also tell fdpo which entity is currently 'in view' and, therefore, from which FIFOs to dequeue PDUs.

By relying on link state cues to control the operation of fdpo, we can accommodate occultation and other interruptions in connectivity simply and efficiently: when the link is lost, CFDP simply stops transmission and reception of data between the two endpoints of the link. This implementation of deferred transmission incurs far less overhead than using the Remote Suspend and Resume user operations to control suspension and resumption of communication:

a) Remote Suspend and Resume operations entail protocol activity, requiring a cooperative interchange of data between entities. Deferred transmission is entirely local; no PDUs are issued or received to affect it.

b) Because deferred transmission is an entirely local mechanism, it is unaffected by delay due to the distance between the participating entities. Moreover, there is no chance of incomplete remote suspension/resumption due to loss of a PDU.

c) Remote Suspend and Resume are transaction-specific. This means that suspending all transmission between any pair of entities would require the reliable transmission of PDUs for every transaction currently in progress between them, as would resumption of transmission. In contrast, the deferred transmission mechanism is atomic and comprehensive.

7.5.3 PDU QUEUING WITHIN THE CFDP ENTITY

Under some circumstances, CFDP PDUs should be physically transmitted (radiated) in an order that differs from the order in which they were generated.

Operational considerations or other user constraints may require that access to transmission bandwidth be allocated among multiple ‘flows’ according to a user-visible management algorithm. Typically, it may be necessary to prevent the transmission of a single large but non-critical file from delaying the delivery of small but critical files whose transmission is requested later. The CFDP ‘flow label’ mechanism is intended to address this sort of requirement. The various ‘flows’ are typically implemented as logically distinct transmission channels within CFDP that must be multiplexed on output.

Additionally, though, some PDUs that serve only CFDP internal control purposes may need to be radiated on an urgent basis, possibly ahead of a large number of file data PDUs that are currently queued for transmission. A single CFDP service request or protocol procedure may result in the transmission of multiple PDUs. Since any single transmission medium can only send one value at a time, a CFDP implementation must provide some mechanism for imposing a rational order of transmission on those PDUs. Typically *queues* (or *FIFOs)* are the basis for this mechanism. However, PDU queuing must be done carefully in order to avert various kinds of trouble. In particular, if a single queue is used and new PDUs are always added to the back of this queue, then:

a) The File Data PDUs for an urgently needed file can never be transmitted until the previously queued PDUs of less important files, bound for the same destination entity, have been transmitted.

b) An ACK PDU will never be transmitted until all previously queued PDUs have been transmitted. This makes the arrival time of the ACK heavily dependent on the size of the backlog of PDUs pending transmission at the ACK's sending entity. Since this size is difficult or impossible to estimate accurately, the sender of the PDU to which the ACK is responding cannot accurately anticipate the ACK's arrival time; it therefore cannot know with any accuracy when to presume data delivery failure and retransmit the PDU.

One alternative approach is to use a single queue but manage it intensively, inserting new PDUs not just at the back but at various points throughout the queue, and possibly rearranging items within the queue as necessary.

Another approach, which seems structurally more complex but may be procedurally simpler, is to use multiple queues and merge them at the point of access to the Unitdata Transfer (UT) layer. A possible implementation is discussed in 7.5.5.

A further note on the effect of queuing on ACK arrival time: selection of accurate retransmission timer intervals in CFDP is difficult, but it need not be impossible. Nearly all of the uncertainty in computing these values can be removed if the CFDP implementation observes these principles (refer to 7.5.6 for a fuller discussion):

a) A positive acknowledgment timer should not be started until the affected PDU can be assumed to have been physically radiated. A service indication from the UT layer may be required for this purpose.

b) Positive acknowledgment timers should be temporarily stopped during any time interval in which the responding entity is unable to transmit (i.e., between tracking passes) and restarted when the responding entity's ability to transmit is restored (i.e., the next tracking pass starts). This activity is entailed in the 'link state change' procedures.

c) ACKs should be delivered to the UT layer immediately, as soon as they are created. This might mean inserting them at the front (rather than the back) of the single outbound PDU queue, or alternatively inserting them at the back of a separate, top-priority queue reserved for ACK transmission.

7.5.4 ADDITIONAL COMMUNICATIONS CHANNEL

The separate queue for ACK transmission alluded to in 7.5.3 might also be considered an 'additional communications channel', a mechanism for immediate transmission of urgent protocol control information.

It has been speculated that such a mechanism might be used for transmission of several types of file directive PDUs. ACK PDUs are clearly urgent enough to warrant top-priority transmission: significant delay in transmitting an ACK can result in premature timer expiration and unnecessary retransmission, consuming scarce bandwidth. It is not yet clear that any other file directive PDUs are similarly critical, so no consensus on this topic has been reached within CCSDS Panel 1F.

7.5.5 FLOW LABELS

Flow label processing is identified in reference [1], but is left undefined. The JPL implementation of CFDP incorporates a flow label algorithm that is intended to provide highly flexible bandwidth allocation without requiring active management.

A JPL flow label is an integer in the range 0 through N inclusive, where N is some small value. In testing to date we have used N = 3, with 0 as the default flow label for transactions that omit the flow label TLV from transaction metadata.

For each remote CFDP entity, fdpd enqueues the PDUs of each file destined for that entity onto one of N+1 FIFOs, depending on the flow label associated with the transaction. FIFO 'N' is designated the 'priority' queue for that entity Each of the other queues is assigned a 'service level', a number that indicates that queue's allocation of total transmission bandwidth in the absence of priority traffic.

Fdpo loops endlessly through the following algorithm to obtain from these N+1 FIFOs the PDUs it sends to the remote entity that is currently in view:

a) If there are any PDUs currently in the priority FIFO, remove the first PDU from that FIFO and transmit it.

b) Otherwise, if any of the non-priority FIFOs are non-empty:

   1) Compute 'service provided' for each non-empty non-priority FIFO. For a given FIFO, service provided is calculated as the FIFO's service total (the total number of bytes of data dequeued so far from this FIFO) divided by the service level assigned to the FIFO.

   2) Remove the first PDU from the FIFO for which the least service has been provided, transmit it, and add its length to that FIFO's service total.

c) Otherwise, wait until fdpd signals that PDUs have been placed in one or more of the FIFOs.

The service levels assigned to non-priority FIFOs can be any numeric values, but the service level assignment scheme we have used in testing enables a small optimization. If the service level assigned to FIFO n (where 0 <= n < N) is 2**n, then you can compute service provided for any FIFO by simply shifting its service total n bits to the right. If N = 3, the FIFOs are configured as follows:

FIFO number	Service level
0	2**0 = 1
1	2**1 = 2
2	2**2 = 4
3	(priority FIFO, service level n/a)

Assigning a given file the flow label N causes it to be appended to the priority FIFO, so that it is transmitted after all previously enqueued priority transmissions (if any), but before all non-priority transmissions. Assigning a given file a flow label less than N causes it to be appended to the corresponding FIFO; it will be transmitted after all previously enqueued transmissions with the same flow label, but possibly before previously enqueued transmissions with different flow labels, depending on the lengths of the various FIFOs and the service levels assigned to them. For example, if all non-priority traffic is assigned either flow label 0 (with service level 1) or flow label 2 (with service level 4), and FIFOs 0 and 2 are both kept non-empty at all times, then transmissions assigned to flow 2 will be delivered four times as rapidly as those assigned to flow 0; flow 2 will occupy 80% of the transmission bandwidth, while flow 0 occupies the remaining 20%.

The effect of this scheme is to apportion transmission resources automatically to various classes of traffic, without ever starving any class of traffic altogether, while still enabling an emergency transmission to take temporary precedence over all other traffic when necessary. No management is necessary, aside from the assignment of service levels to flows.

NOTE – When an unused FIFO begins to be used, the algorithm described above may enable it to monopolize the transmission link for some time. (Its service total is initially zero, so its computed service provided may remain less than that of all other flows for a while, even if has a lower service level.) For this reason, an additional computation is performed each time a PDU is dequeued from a non-priority channel: if the difference between lowest and highest calculated values of service provided is greater than some constant K times the current data transmission rate (in bytes per second), then the service totals of all FIFOs are reset to zero to resynchronize the algorithm automatically. K, a management parameter, represents the maximum number of seconds the mission operator is willing to risk letting one flow monopolize the transmission link.

7.5.6 TIMERS

Successful transmission of a PDU can be signified by an acknowledgment, but the only reliable way to detect a possible failure in transmission is to wait for a timeout period to expire prior to acknowledgment. Computation of these timeout periods in CFDP is complicated by the fact that connectivity is discontinuous; reception of an acknowledgment may be arbitrarily delayed, not only by planetary occultation but also by resource scheduling decisions at both ends of the link. The effect of using an inaccurate timeout period to control retransmission can be either unnecessary delay in data delivery (if the timeout period is too long), or unnecessary retransmission traffic (if the timeout period is too short).

The JPL implementation of CFDP uses the following mechanism to detect timeout expiration for EOF and Finished PDUs:

a) The total time consumed in a ‘round trip’ (transmission and reception of the original PDU, followed by transmission and reception of the acknowledgment) has the following components:

1) Protocol processing time at sender and receiver.

2) Inbound queuing: delay at the receiver while the original PDU is in a reception queue, and delay at the sender while the acknowledgment is in a reception queue.

3) Outbound queuing: delay at the sender while the original PDU is in a FIFO waiting for transmission, and delay at the receiver while the acknowledgment is in a FIFO waiting for transmission.

4) Round-trip light time: propagation delay at the speed of light, in both directions.

5) Delay due to loss of connectivity.

b) Processing time at each end is assumed to be negligible.

c) Inbound queuing delay is also assumed to be negligible, because processing speeds are high compared to data transmission rates, even on small spacecraft.

d) Two mechanisms are used to make outbound queuing delay negligible:

1) At the sender, the timer for a given EOF or Finished PDU is not started until the moment that fdpo delivers the PDU to the link layer for transmission. All outbound queuing delay for the PDU has already been incurred at that point.

2) At the receiver, acknowledgment PDUs are always inserted at the *front* of the priority FIFO to ensure that they are transmitted as soon as possible after reception of the PDUs to which they respond. (Acknowledgment PDUs are small and are sent infrequently, so the effect on the delivery of emergency traffic is insignificant.)

e) We assume that one-way light time to the nearest second can always be known (e.g., provided by the MIB). So the initial value for each timer is simply twice the one-way light time plus 1 second of margin to account for processing and queuing delays.

f) This leaves only one unknown, the additional round trip time introduced by loss of connectivity. To account for this, we again rely on external link state cues. Whenever loss of connectivity is signaled by a link state queue, we not only stop fdpo, but also suspend the timers for all PDUs destined for the corresponding remote entity; reacquiring link to the entity causes those timers to be resumed. There is no need to try to estimate connectivity loss delays in advance, nor is there is a need for CFDP itself to be aware of either the ephemerides or the tracking schedules of the local entity, or of any remote entity.

In testing performed to date, this mechanism seems to trigger timeout-driven retransmission without imposing either excessive retransmission traffic or excessive file delivery delay.

7.5.7 IGNORING LATE DATA

Unacknowledged-mode transactions always terminate on receipt of the EOF (No error) PDU. Therefore any Metadata or file data PDU received after the EOF (No error) PDU for the same transaction may be ignored.

7.5.8 TRANSACTION INDICATIONS

The Transaction.indication primitive that is issued to the user application upon initiation of a transaction indicates the ID assigned to the new transaction. However, CFDP is not constrained to block the submission of a Put.request primitive until a Transaction.indication has been issued in response to the prior Put.request; nothing in the standard prevents the submission of multiple Put.requests in quick succession without intervening reception of any resulting Transaction.indications. In order for the user application to be able to associate a transaction ID with the corresponding Put.request (and, implicitly, with the corresponding file), an implementation-specific mechanism must be supplied.

One option is flow control, the single-threading of Put.request activity: after the CFDP implementation receives a Put.request, it refuses to accept another one until it has delivered the resulting Transaction.indication. While the CFDP standard does not require this behavior, neither does it prohibit it.

Another option would be an implementation-specific transaction tag system, such as might be provided in an application programming interface. For example, the function used to submit a Put.request might return a 'request ID' number, which could subsequently be inserted into the data object that is sent to the user application when the resulting Transaction.indication is produced; the user application could link the Transaction.indication to the corresponding Put.request by request ID.

7.6 SIMPLE ANALYSIS OF NAK RETRANSMISSION

NOTE – Contributed by R. J. Smith, British National Space Centre (BNSC)/Qinetiq.

The performance for CFDP can be gauged by making a few simple approximations using the method outlined in this subsection. The most important measure is the probability of a PDU being received.

It has been assumed that the link has a long delay, whereby the data rate is high relative to the link delay; i.e., all data is transmitted and then, at some later time, all data is received. In this case, there is no time overlap between transmission and reception, which is not unreasonable, as data rates will increase and the speed of light will not.

The probability of PDU loss, q_{pl}, is dependent on the number of bytes in the PDU, n_p, and the probability of a bit error, p_{be}.

This confirms that risk of PDU loss increases with PDU length.

In a single transaction, most of the traffic consists of File Data PDUs, which are typically significantly larger than other PDUs involved. The majority of non-File Data PDUs are small, i.e., 20-200 bytes, and so are less prone to corruption. The only exception is the NAK PDU, which may be large if there is a lot of data corruption, and it is the trigger for File Data PDU retransmission.

Hence, the transaction simplifies to:

a) Send all File Data PDUs.

b) Return NAK PDU.

How big should File Data PDUs be? If they are too big, they are easy prey to bit errors, meaning the whole PDU must be resent. If they are too small, then their headers become an unacceptably large overhead. In this example, 1024 bytes has been taken as a reasonable compromise.

How big are NAK PDUs? Their size is dependent on the File Data PDU length, n_{fd}, probability of bit error, p_{be}, and the file size, n_{fl}.

The number of File Data PDUs, N_{fd}, is:

The probability that a File Data PDU is lost, q_{fd}, is:

So an estimate of the number of NAKs, n_n, is:

And the probability of a NAK PDU loss, q_n, is:

How likely is a bit error? This depends on the mission and its environment. For the purposes of this example, a typical link is assumed to lose 1 bit in 10^{10} ($p_{be}=10^{-10}$), and a poorly-designed link will drop 1 bit in 10^6 ($p_{be}=10^{-6}$). These figures are based on current space communications links with and without error correction.

Consider a 1Gb file ($N_{fd} = 10^6$):

p_{be}	10^{-10}	10^{-8}	10^{-6}	10^{-5}
q_{fe}	$8x10^{-7}$	$8x10^{-5}$	$8x10^{-3}$	8x10-2
n_n	~1	80	8000	80000
q_n	$8x10^{-10}$	$6x10^{-6}$	$6x10^{-2}$ (1:16)	0.998 (~1)

Bit error probabilities of 10^{-8} and 10^{-5} have been added for context, as the trends are far from linear, especially around 10^{-6}. As the link quality decreases, the number of NAKs rises sharply, and the probability of NAKs failing becomes almost certain (~1).

The example presented here provides a rough guide to performance for a typical transaction. However, the analysis method has also been outlined to allow users to evaluate CFDP performance with their own mission parameters.

8 IMPLEMENTATION REPORTS

8.1 OVERVIEW

8.1.1 This Section contains reports contributed by several different member Agencies describing the CFDP implementations developed and tested by those Agencies. The first such report was contributed by CNES, and it discussed their very early CFDP implementation effort. That effort and report subsequently initiated, and in many ways defined, the later activities of the other implementers and implementations. The CNES report has become outdated and is therefore not included here, but this pioneering effort is gratefully acknowledged.

8.1.2 A total of five independent implementations have been made as a part of the CFDP development process. The creating organizations are BNSC/Qinetiq, European Space Agency (ESA)/European Space Research and Technology Centre (ESTEC), NASA/GSFC, NASA/JPL and NASDA/NEC. Each of the implementations has been tested thoroughly with the other implementations, and each interoperates correctly with the others. In addition, an operational implementation for the MESSENGER spacecraft has been developed by the Applied Physics Laboratory (APL) of the Johns Hopkins University (JHU). Reports on each of these implementations, except for that of JPL, are contained in this section.

8.1.3 A short description of the test program is contained in annex A of reference [3].

8.2 BNSC/QINETIQ IMPLEMENTATION REPORT

NOTE – Contributed by R. J. Smith, BNSC/Qinetiq.

8.2.1 SCOPE

This report provides a brief outline of the BNSC/Qinetiq implementation of CFDP, highlighting the design approach taken and pitfalls encountered.

8.2.2 INTERFACES

There are three interfaces to CFDP, as follows:

a) primitives;

b) Protocol Data Units (PDU);

c) filing system.

Primitives are defined in the CFDP specification. They constitute the high-level application interface to CFDP and fall into two categories, requests and indications. The former is a command to the CFDP entity, and the latter provides feedback from it to the user application.

PDUs define the format of the packet data transmitted. They form the low-level interface of the protocol and can be layered with other network protocols, e.g., transport, security or packet layer. These are defined in reference [1].

CFDP includes filestore operations (e.g., list directory, delete file). However, the precise implementation of these is architecture-specific, so the code must interact with the local operating system in order to perform these tasks.

8.2.3 ENTITY OVERVIEW

The BNSC implementation is broken into a number of components (figure 8-1), each with its own responsibilities:

a) Individual transaction tasks handle the generation of appropriate responses for a particular CFDP transaction, i.e., servicing primitive requests and incoming PDUs, generation of primitive indications and outgoing PDUs, and tracking of the transaction's status.

b) PDU servers are small service routines responsible for receiving PDUs from their transport layer and passing them to the daemon.

c) The daemon is the administrator for the entity and is responsible for monitoring transactions' status. It acts as a router for individual transaction tasks.

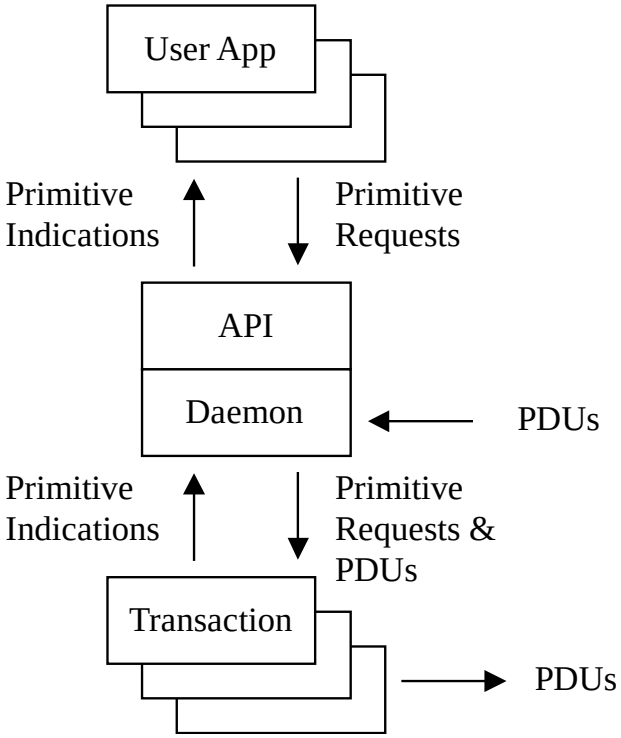


Figure 8-1: Data Flow Between CFDP Entity Components

8.2.4 DESIGN NOTES

8.2.4.1 VxWorks

A flight test opportunity arose on the Sparc Microprocessor Experiment (SMX-2) on Qinetiq's STRV-1d satellite. The satellite is a test bed for new space technologies, and the experiment is designed to allow codes to be run in a space environment, with genuine mission parameters.

SMX-2 employs a Sun Sparc-based chipset called the Embedded Real-Time Computer 32-bit, ERC32. It is more powerful than the current generation of space-based CPUs and has sufficient power to allow significant onboard processing. VxWorks was chosen as the real-time embedded operating system for the experiment, as it is multitasking, scalable and architecture-independent. It also has its own development tools which operate remotely on a separate host machine, allowing the embedded code to be debugged in-situ on its target system with minimal interference. This combination has several significant benefits:

a) Sophisticated software tools reduce development times.

b) In-situ development reduces risk of operational software failure.

c) A scalable operating system minimizes resource overheads.

d) Multitasking allows simple operational management.

e) Architecture-independence means code can be reused, giving greater stability through heritage.

CFDP development has taken advantage of the VxWorks environment in precisely those ways outlined above. The choice of operating system immediately led to certain design decisions:

a) Multiple transactions are handled via the multitasking side of the operating system.

b) Operating system message queues are used to communicate between the daemon and transaction tasks.

c) The C programming language was chosen, as a version of the GNU C cross-compiler for the target system is included with VxWorks.

8.2.4.2 Transaction Task

Each transaction has a unique handling task at each CFDP entity involved (figure 8-2). PDUs are identified by the entity daemon and passed to the relevant transaction task in the order in which they arrive. These PDUs are then parsed, filtered and processed.

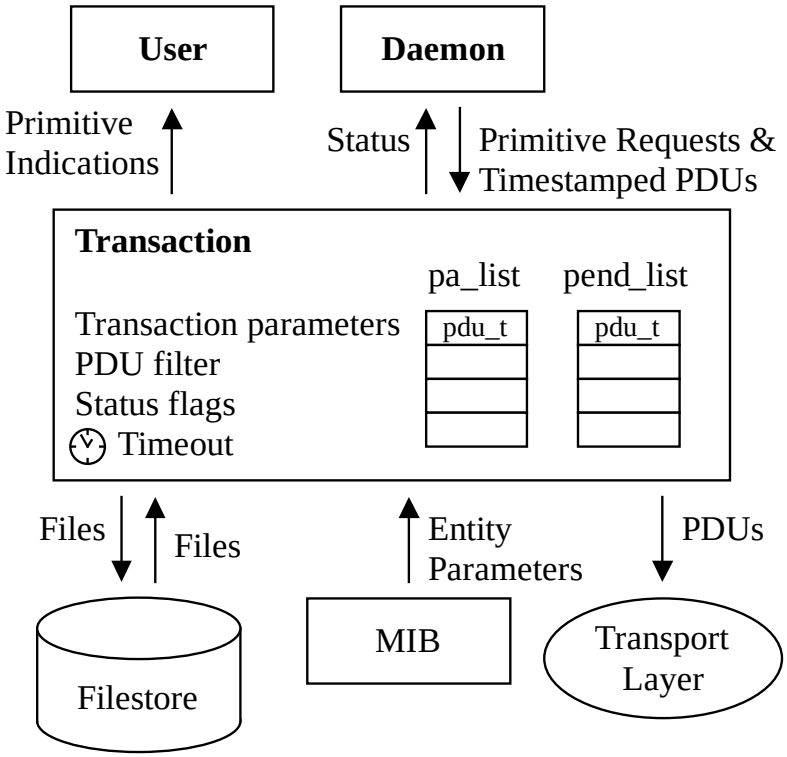


Figure 8-2: Detailed Data Flow for Transaction Task

Incoming PDUs or primitive requests cause the task to:

a) update its status accordingly;

b) store received data;

c) generate an appropriate response, i.e., primitive indications and/or outgoing PDUs.

Outgoing PDUs are assembled in a data space, ready for transmission over the underlying network. The current implementation can use a UDP or TCP transport layer, but the modularity of the code also allows other protocols to be employed. In addition, PDUs can be encapsulated in CCSDS Packets and SMP handshaking protocols for connection to the Qinetiq Ground Segment.

The main loop of the transaction task is shown in a simplified form in figure 8-3. PDUs which require positive acknowledgement are added to a list with an expiry time. This list is maintained in time order and redundant entries are removed when a suitable acknowledgement is received.

If no suitable response has been received prior to a positive acknowledgement entry expiring, the original PDU is resent and a retry is counted. Once a prescribed number of retries have occurred, fault handling procedures are engaged. An inactivity timeout is generated if no PDUs have been received by a transaction for a prescribed period.

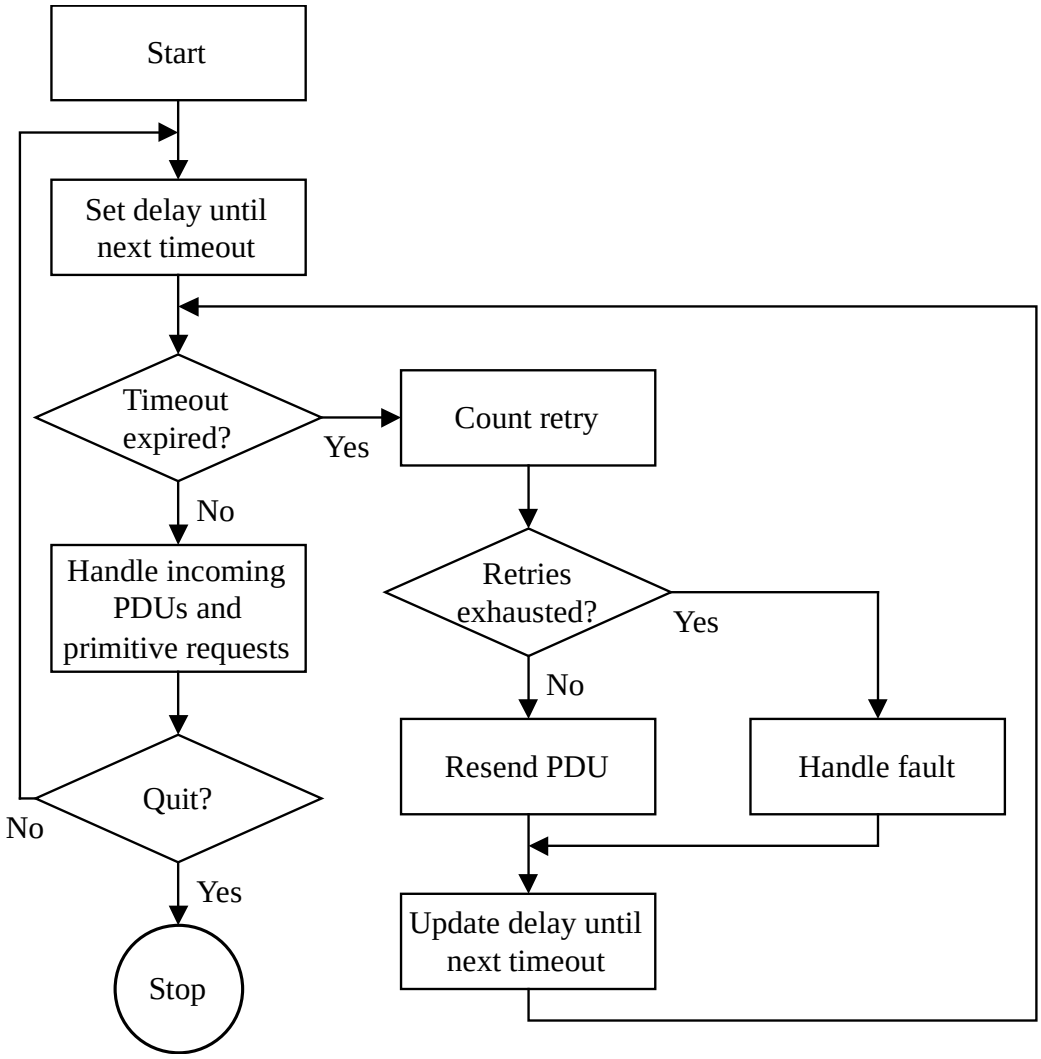


Figure 8-3: Simplified Transaction Task Algorithm

If an error is detected in the protocol, the fault handler determines the subsequent course of action. This is determined from the entity's MIB values, unless fault handler overrides have been specified as part of the transaction.

During suspension, an incoming PDU is processed at a basic level to ensure that an appropriate acknowledgement response is sent and to allow cancellation to be initiated if required. Barring cancellation, the PDU is then simply stored in a list until resumption, when it is parsed and processed by unsuspended entity.

The task must communicate with the daemon to inform it of significant changes in status, which it does through the daemon's message queue. It must also notify the user application of status changes, via the primitive indications.

There are currently two types of transaction task implemented: sending (client) and receiving (server). Each has a core routine with a common design framework to send or receive a file respectively. The simplicity of this design means that the numerous state changes can be accommodated with comparative ease, regardless of their order.

Local entity parameters are accessed via the MIB. This currently includes details of:

a) local entity identity;

b) receiving ports, i.e., PDU servers;

c) connections to other entities;

d) default characteristic parameters for transactions.

File access is performed directly by the transaction task on the filestore, either reading a file for transmission or writing a received file.

Transaction tasks linger for a single activity timeout period beyond their transaction's termination in a zombie state. This ensures retransmission of acknowledged PDUs and easy identification of residual transaction traffic, which may occur due to the nature of the underlying space network.

8.2.4.3 Daemon

The daemon task handles the administration of CFDP transactions and scheduling of events, as depicted in figure 8-4. It is responsible for spawning PDU server tasks, according to the information contained in the MIB, and performing a syntax check on the MIB at start-up.

PDU servers extract PDUs from their transport layer, timestamp them and pass them to the daemon. The PDUs are classified by the daemon according to whether they relate to a client or server transaction, from their direction bit. They are then checked against the daemon's internal lists of known transaction identifiers.

Transactions in progress are listed as 'active', and are transferred to the 'dead' list when they terminate. The dead list is a ring buffer which holds a given number of identifiers, so old transactions are only remembered until their identifier is overwritten.

PDUs with unlisted identifiers are considered to be new transactions and cause the daemon to spawn a new transaction task. During the inevitable delay while a new transaction task initiates, all related PDUs are stored on a pending list within the daemon. These pending PDUs are routed to the transaction task when it is ready to accept them.

Primitive requests are passed to the daemon and automatically routed to the appropriate transaction task in a similar way to PDUs. All primitive indications are returned directly to the user.

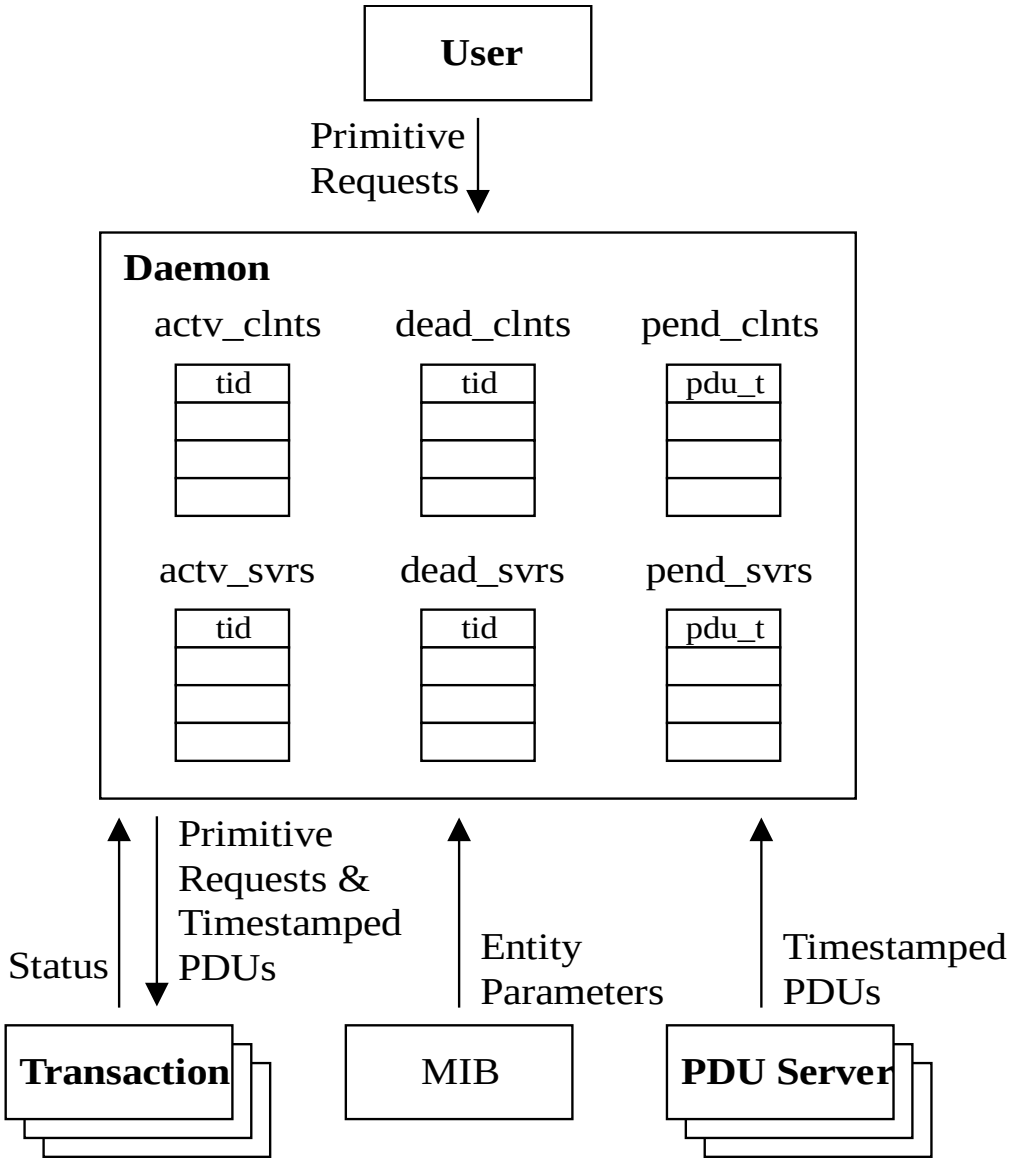


Figure 8-4: Detailed Data Flow and Interfaces for Daemon

8.2.5 CAPABILITIES MATRIX

CFDP Implementation Survey

Agency	Name	Submitted by
BNSC	Qinetiq	R Smith

General Implementation Information

Platform	OS	Language
Force Sparc 3CE	VxWorks 5.4	C

Max. File Size	Max. Segment Size	Mechanism Used for Persistent Storage	Other Persistent Storage Options
0xffffffff	0xffffffff	RAM-based DOS FS	None on development system

Underlying Communications Systems

CCSDS AOS VCDU	CCSDS TM_TC	CCSDS Prox_1	SCPS_NP	UDP_IP	Other
				X	TCP_IP, Encapsulated CCSDS TM_TC in IP Packets, SMP (Qinetiq Ground Segment)

1. CFDP Procedures

CRC Proced.	Put Proced.	Transaction Start Proced.	PDU Forwarding Proced.	Copy File Proced.	Positive Ack. Proced.	Faults Proced.	Filestore Proced.
X	X	X	X	X	X	X	x

2. CFDP Protocol Classes

Unreliable Transfer	Reliable Transfer	Reliable Transfer by Proxy
X	X	X

3. CFDP Protocol Options

End Type

Sender	Receiver
X	X

Put Modes

UnACK	NAK
X	X

Put NAK Modes

Immediate	Deferred	Prompted	Asynchronous
X	X	X	X

Put File Types

Bounded	Unbounded
X	X

Segmentation Control (Record Boundaries Respected)

Yes	No
	X

Put Primitives (Receiving End)

File_segment_receive.ind
X

Put Error Responses (Sending End)

Ignore	Abandon	Cancel	Suspend
X	X	X	X

Put Error Responses (Receiving End)

Ignore	Abandon	Cancel	Suspend
X	X	X	X

Put Actions

Cancel_PutAction_	Suspend_PutAction_
X	X

Cancel Put Action (Receiving End)

Discard data	Forward incomplete
X	X

Put Report Modes (Sending End)

Prompted_Rpt_	Periodic
X	On termination

File Store Options

Create File	Delete File	Rename File	Append File	Replace File	Deny File	Create Dir	Remove Dir	Deny Dir
X	X	X	X	X		X	X	

Directory Operations

Directory Listing Request	Directory Listing Response
X	X

Release of Retransmission Buffers

Incremental and Immediate	In total When 'Finished' Received
	X

4. Timers and Counters

Timers

NAK Retry Timer	ACK Retry Timer	Prompt _NAK_ Timer	Async NAK Timer	Keep Alive Timer	Prompt _Keep Alive_ Timer	Inactivity Timer
X	X			X		X

Counters

NAK Retry Counter	ACK Retry Counter
X	X

8.3 ESA/ESTEC IMPLEMENTATION REPORT

NOTE – Contributed by Massimiliano Ciccone, ESA/ESTEC.

8.3.1 INTRODUCTION

The goal of this implementation report is to describe the CFDP software implementation within ESTEC through a detailed architectural design of the protocol's kernel and an overview of its interaction with the supporting software components, as well as a brief description of the development environment.

8.3.2 IMPLEMENTATION STATUS

8.3.2.1 Overview

The ESTEC CFDP software coverage so far entails the implementation of the entire *Core* file delivery capability, in both Reliable and Unreliable transfer mode, and the implementation of the Extended and SFO procedures providing Store-and-Forward capabilities. This means that the CFDP software has the capability to perform a single *Point-to-Point* file copy operation between two CFDP entities; a *Proxy* file copy operation involving three CFDP entities (two if used as a Get Request); and a file transfer via a single or multiple *Serial Waypoint(s).*

As described in reference [1], the implemented classes are as follows:

a) Class 1: Unreliable Single *Point-to-Point* File Transfer;

b) Class 2: Reliable Single *Point-to-Point* File Transfer;

c) Class 3: Unreliable File Transfer via one or multiple Waypoint(s) in series;

d) Class 4: Reliable File Transfer via one or multiple Waypoint(s) in series.

8.3.2.2 CFDP Implementation Survey

Agency	Name	Submitted by
European Space Agency(ESA)	ESTEC	Massimilliano Ciccone

General Implementation Information

Platform	OS	Language
PC	Windows NT/9X	Object Pascal on Delphi

Max. File Size	Max. Segment Size	Mechanism Used for Persistent Storage	Other Persistent Storage Options
FFFFFFFF	1024 bytes	DOS File System	

Underlying Communications Systems

CCSDS AOS VCDU	CCSDS TM_TC	CCSDS Prox_1	SCPS_NP	UDP_IP	Other
				X	

1. CFDP Procedures

Core Procedures

CRC Proced.	Put Proced.	Transaction Start Proced.	PDU Forwarding Proced.	Copy File Proced.	Positive Ack. Proced.	Faults Proced.	Filestore Proced.
X	X	X	X	X	X	X	x

Extended Procedures

General Req.s	File Data Relay Proced.	Consignment Notification Proced.	PDU Relay Proced.	Suspend/Resume Propagation Proced.	Deferred Transmission Proced/
X	X	X	X		X

2. CFDP Protocol Classes

Unreliable Transfer	Reliable Transfer	Reliable Transfer by Proxy	Unreliable via One Waypoint	Reliable via One Waypoint
X	X	X	X	X

3. CFDP Protocol Options

End Type

Sender	Receiver
X	X

Put Modes

UnACK	NAK
X	X

Put NAK Modes

Immediate	Deferred	Prompted	Asynchronous
X	X	X	X

Put File Types

Bounded	Unbounded
X	X

Segmentation Control (Record Boundaries Respected)

Yes	No
	X

Put Primitives (Receiving End)

File_segment_receive.ind
X

Put Error Responses (Sending End)

Ignore	Abandon	Cancel	Suspend
X	X	X	X

Put Error Responses (Receiving End)

Ignore	Abandon	Cancel	Suspend
X	X	X	X

Put Actions

Cancel_PutAction_	Suspend_PutAction_
X	X

Cancel Put Action (Receiving End)

Discard data	Forward incomplete
X	

Put Report Modes (Sending End)

Prompted_Rpt_	Periodic
X	X

File Store Options

Create File	Delete File	Rename File	Append File	Replace File	Deny File	Create Dir	Remove Dir	Deny Dir
X	X	X	X	X	X	X	X	X

Release of Retransmission Buffers

Incremental and Immediate	In Total When 'Finished' Received
	X

Put Report Modes

Prompted	Periodic
X	X

Extended Options

Entity Role

Original Sender	Waypoint	Final Receiver
X	X	X

Number of Hops

Number of Hops One	More Than One
X	X

Forwarding Method

Incremental and Immediate	In Total On Complete Custody Acquisition
X	X

Timers and Counters

Timers

NAK Retry Timer	ACK Retry Timer	Prompt _NAK_ Timer	Async NAK Timer	Keep Alive Timer	Prompt _Keep Alive_ Timer	Inactivity Timer
X	X			X		X

Counters

NAK Retry Counter	ACK Retry Counter
X	X

8.3.3 CFDP VS. OSI MODEL LAYERING CONTEXT

CFDP can be seen as a high-level protocol service designed to take advantage of the lower-level protocols, relying on a minimal underlying data communication service , such as CCSDS TC/TM, User Datagram Protocol (UDP)/Internet Protocol (IP), etc.

NOTE – Although the protocol can operate over a wide range of underlying communication services, this recommendation assumes the use of CCSDS packet delivery services (reference [1]), including:

a) CCSDS conventional packet telecommand;

b) CCSDS conventional packet telemetry;

c) CCSDS Advanced Orbiting Systems (AOS) Path service.

Even though CFDP has not been designed to comply strictly with any correspondent OSI model layer, a rough 'logical' comparison can still be made.

CFDP's correspondent group of layers can be found in the *Application Service*. It includes portions of the OSI *Session Layer*, *Presentation Layer* and the *Application Layer*, and it also extends into the space above the OSI stack itself, traditionally considered to be system application space (i.e., for CFDP Extended procedures).

The ESTEC version so far works only over the connectionless UDP/IP underlying protocol (see figure 8-5). The UDP was designed to provide a low network overhead mechanism for transmitting data over the lower layers. Although it still provides packet handling and sequencing services, UDP lacks a number of TCP's more powerful connection-oriented services, such as acknowledgement, flow control and packet reordering (nevertheless provided by CFDP).

The main services offered by UDP can be summarized as follows:

a) segmenting of Data Streams (CFDP PDUs) into packets;

b) reconstruction of Data Streams from packets;

c) socket services (Winsock 2.0 creation and manipulation) for providing multiple connections to ports on remote hosts.

The UDP *Host-to-Host communication layer* (figure 8-5) handles the services needed to provide reliable communications functionality between network hosts.

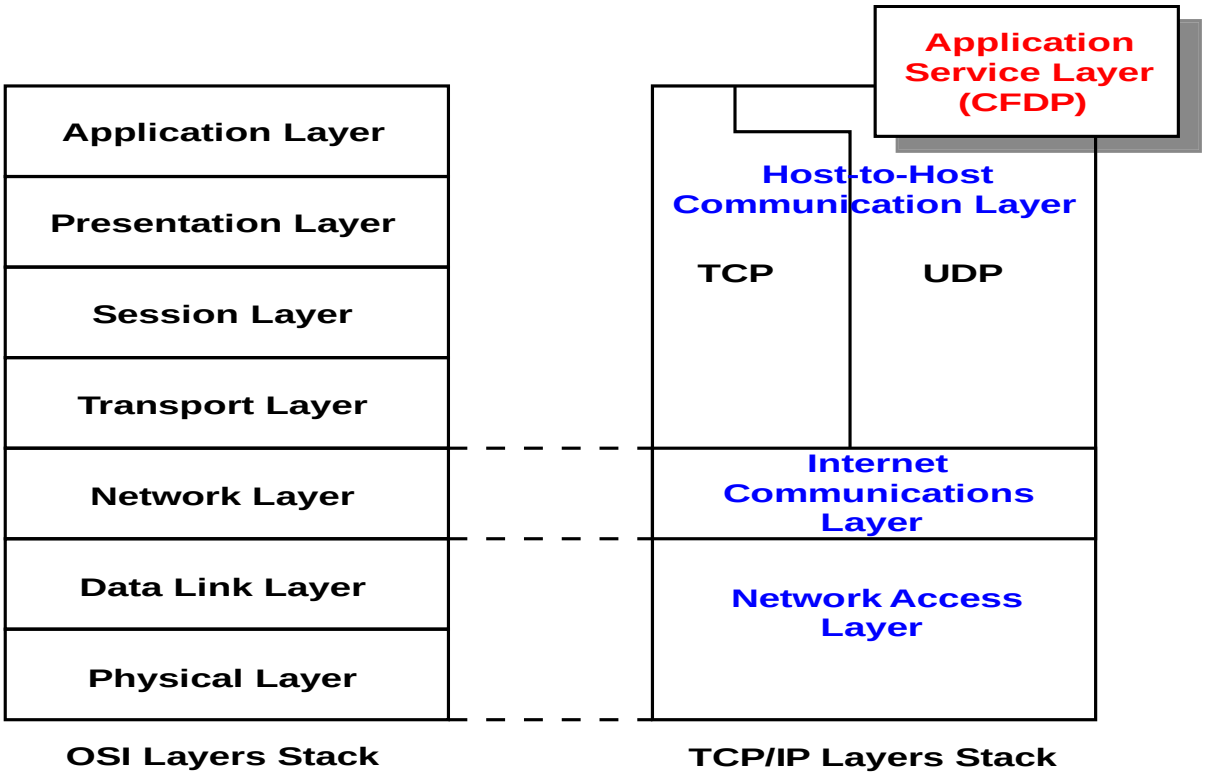


Figure 8-5: Correspondence Between CFDP and OSI Layers

8.3.4 IMPLEMENTATION ENVIRONMENT AND CODING

8.3.4.1 Overview

ESTEC's CFDP implementation utilized a multi-threading Windows application developed under the *Delphi 6* Borland Integrated Development Environment (IDE) using the *Object Pascal* programming language.

All computers used were IBM PC-compatibles running Windows 9X/NT OS.

8.3.4.2 Introduction to Object-Oriented Programming (OOP)

OOP is a programming paradigm that uses discrete objects (an instance of a Class), containing both data and code, as application building blocks. Although the OOP paradigm does not necessarily lend itself to easier-to-write code, the result of using OOP traditionally has been easy-to-maintain code. Having an object's data and code together simplifies the process of searching for bugs, fixing them with minimal effect on other objects, and improving the program one part at time. Traditionally, an OOP language contains implementations of at least three OOP concepts, as follows:

a) *Encapsulation*: Deals with combining related data fields and hiding the implementation details. The advantages of the encapsulation include modularity and isolation of code from other code.

b) *Inheritance*: The capability to create new objects that maintain the properties and behavior of ancestor objects. This concept enables the creation of object hierarchies, such as a Visual Component Library (VCL), first creating generic objects and then creating more specific descendants of those objects that have more narrow functionality. The advantage of inheritance is the sharing of common code.

c) *Polymorphism*: Literally, polymorphism means 'many shapes'. Calls to methods of an object variable will call code appropriate to whatever instance is actually in the variable.

An *Object* is comprised of:

a) *Fields*: Data variables contained within objects.

b) *Methods*: The name of procedures and functions belonging to an object. Methods are those things that give an object behavior rather than just data.

c) *Properties*: A property is an entity that acts as an access interface to the data and code contained within an object. Properties insulate the end user from the implementation details of an object.

8.3.5 CFDP SOFTWARE FUNCTIONAL CONTEXT

Before starting to describe the implemented *CFDP Component* Software, it is worthwhile to give an overview of the complete CFDP Software Package developed within ESTEC, which encompasses three software modules:

a) the CFDP User Software;

b) the CFDP Component, representing the CFDP protocol behavior;

c) the CFDP Packet Service Component (CPSC), representing the interface to the UT Layer.

As shown in figures 8-6 and 8-7, the *CFDP Component* interacts with both the CFDP User Software and the CPSC, allowing a user to fully configure and operate the protocol over a 'selectable' set of conceptual underlying communication systems (UDP, CCSDS, etc.).

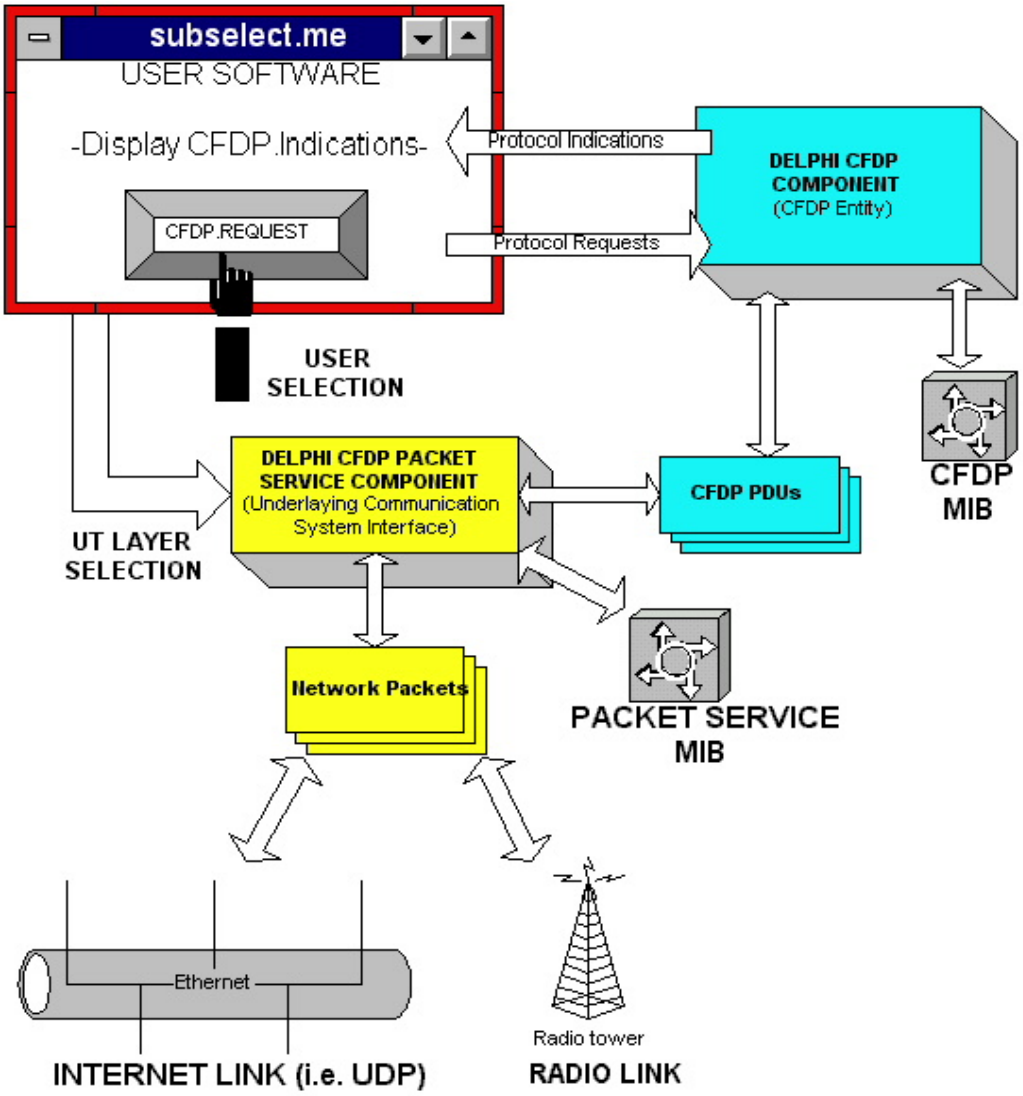


Figure 8-6: CFDP Software Functional Diagram

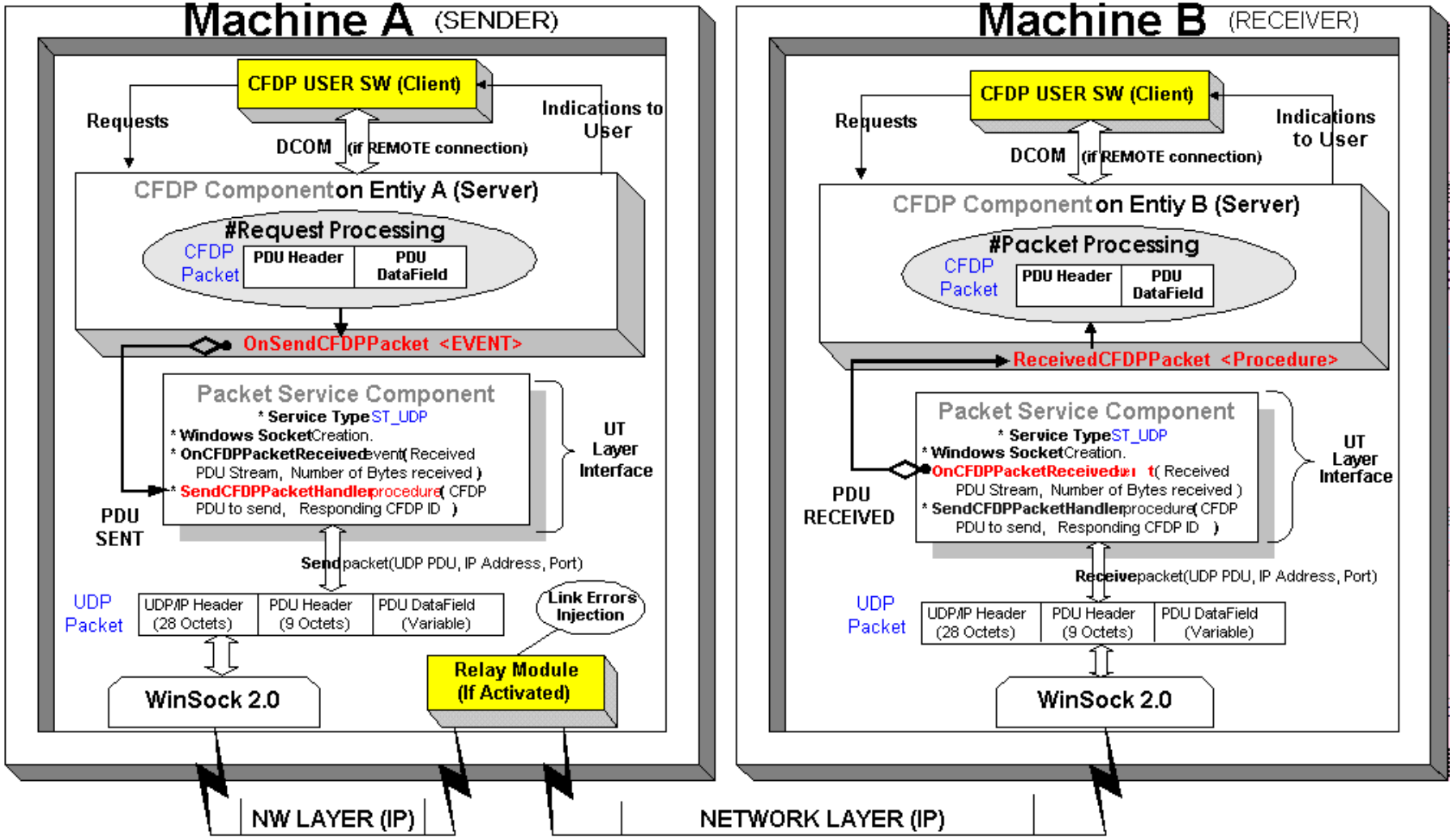


Figure 8-7: CFDP Software Elements (Components) Diagram and Packet Flow

8.3.6 CFDP/UT PACKET ROUTING

It is important to describe the different routing stages occurring in an *end-to-end* CFDP communication. First, the CFDP software performs packet routing 'internally' (at CFDP level) by resolving the CFDP address mapping from PDU's *Destination CFDP ID* to *Next hop CFDP ID* by means of the loaded CFDP MIB file. See figure 8-8.

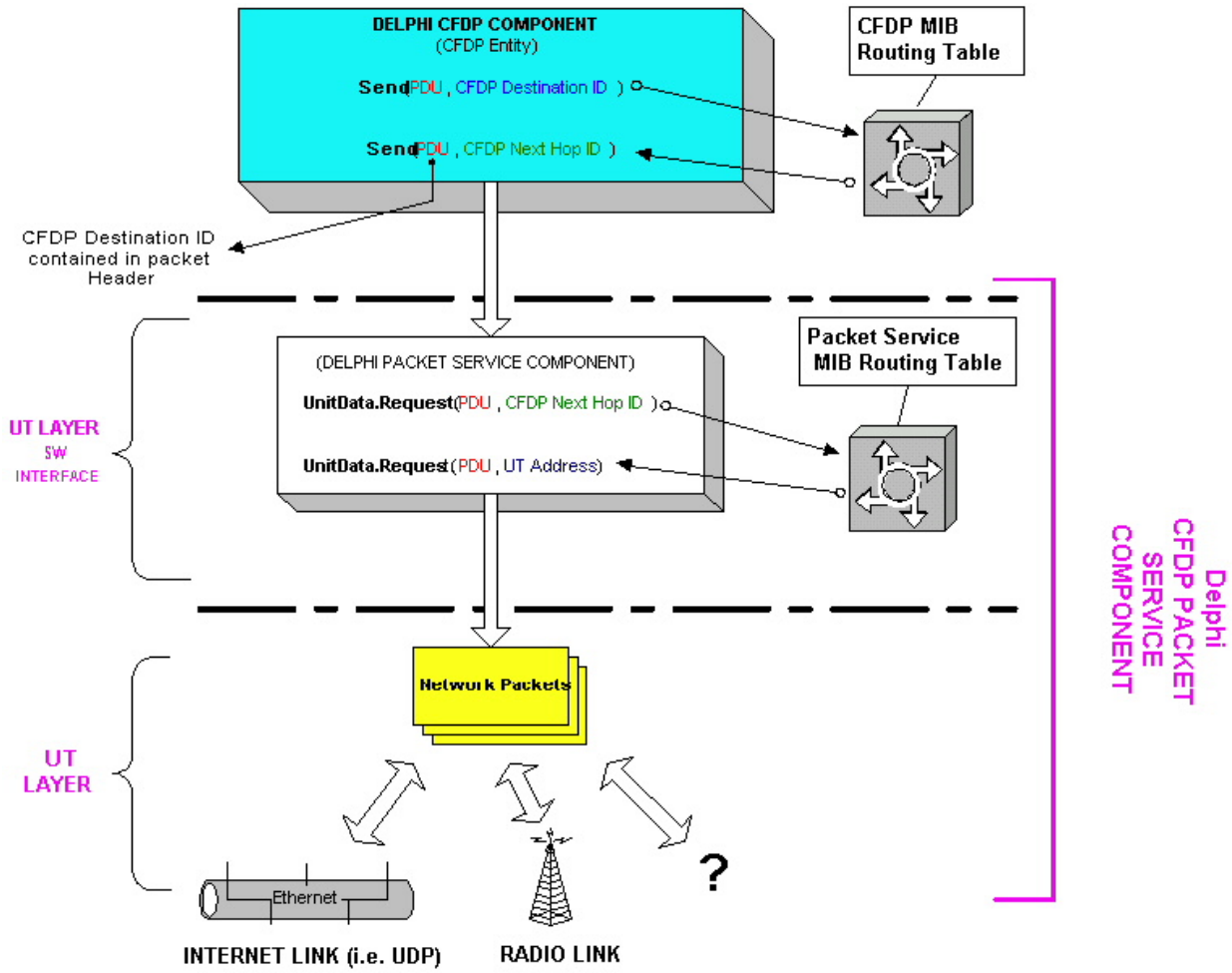


Figure 8-8: CFDP/UT Packet Routing

An example of the CFDP MIB Address table is as follows:

cfdp03: 05 #CFDP Server (Waypoint)

This sample line shows a typical CFDP routing. It means that, in case the PDU final destination is the CFDP entity 03, the packet will be sent to CFDP entity 05 (next hop), since there is not a direct link to entity 03.

At this stage, the CFDP PDU is ready to be released over the underlying communication layer. Therefore, the information provided by CFDP to the ***Unit Data Transfer (UT) Layer Interface*** is:

> CFDPSend (CFDP_PDU (final CFDP Destination ID in the Header), *Next_Hop_CFDP_ID*)

The UT software interface (CPSC) itself will then perform the translation between the passed CFDP ID (*Next_Hop_CFDP_ID*) and the corresponding UT Layer address (i.e., physical network address) by using its own MIB look-up capability.

The resulting UT Layer address may be an Internet address, radio device buffer, APID, virtual channel number, or other implementation-specific mechanism.

An example of a UT layer MIB Address table could be:

> # cfdp05: 128.244.47.100/6769 #APL SRS-Protolab2 PC

Therefore, the UT Service Access Point for CFDP will be:

> UNITDATA.Request (UT_SDU(CFDP PDU), Destination UT Address (i.e., *IP Address*))

The CPSC now has enough information for relaying the CFDP packet to the underlying UT layer.

8.3.7 THE CFDP PACKET SERVICE SOFTWARE COMPONENT

The Delphi *CFDP Packet Service Component* (CPSC) is a software module specially developed to support packet delivery, interacting with both the CFDP User Software and the CFDP Entity (another Delphi component), in a way that can be fully configured by the user software to operate over a set of various underlying communication systems (UDP, CCSDS, etc.). (See figure 8-6 for functional context.) In other words, it is in charge of handling all the procedures related to CFDP PDUs sending and receiving over the *underlying protocol layer*. The only underlying communication protocol interface implemented for the CFDP *Packet Service* component so far is UDP/IP (see figure 8-9).

The UDP was designed to provide a low-network overhead mechanism for transmitting data over the lower layers. Although it still provides packet handling and sequencing services, UDP lacks a number of TCP's more powerful connection-oriented services, such as acknowledgement, flow control and packet reordering (nevertheless provided by CFDP). The main services offered by UDP can be summarized as follows**:**

a) segmenting of Data Streams (CFDP PDUs) into packets;

b) reconstruction of Data Streams from packets.

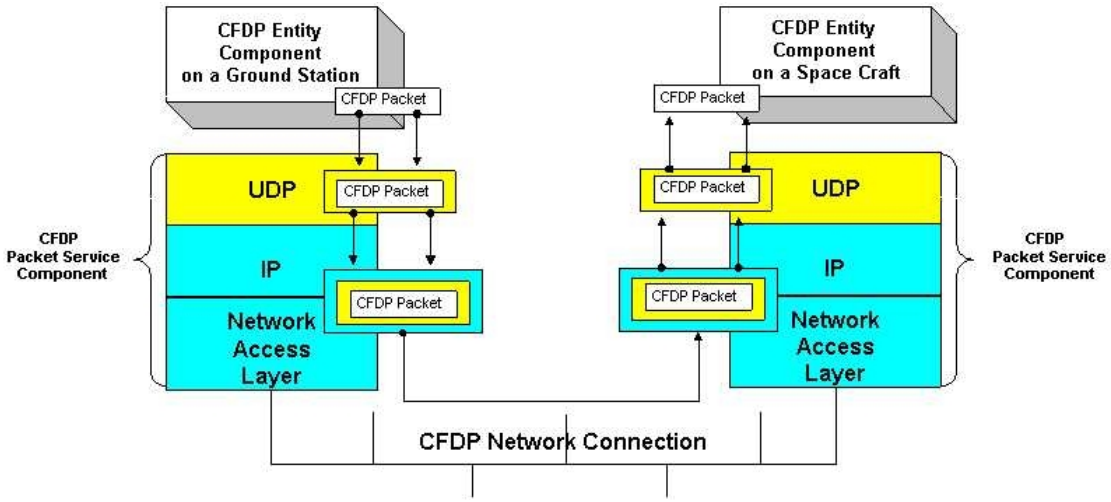


Figure 8-9: CFDP Packet Encapsulation

Socket services (Winsock 2.0 creation and manipulation) are used to provide multiple connections to ports on remote hosts.

The CFDP ***UT Layer Interface*** can be associated with the CFDP *Packet Service component* software linked to the CFDP Entity Component.

The *error handling method* for such an interface is the one related to the selected underlying layer (i.e., UDP delivery and duplicate protection are not guaranteed).

So far, no 'local' *Flow Congestion Control* is performed within the UT interface, but it will be implemented as soon as possible as a *'bit rate control mechanism'* inside the CFDP Packet Service component's sending module. This mechanism will limit the maximum packet flow over the underlying network, and will be useful to measure the CFDP bandwidth efficiency through the total transaction's bandwidth and transfer time.

8.3.8 SOFTWARE ARCHITECTURAL DESIGN (SAD)

8.3.8.1 Overview

ESTEC software implementation of the protocol can be subdivided into three different 'functional modules':

a) *Core* procedures;

b) *Extended* procedures;

c) *SFO* procedures.

The software does not respond to any onboard requirement and is strictly ground-oriented.

The ESTEC prototype implements the CFDP Entity as a *Delphi Component*.

As a result of the object-oriented programming technique, classes defined and organized in software units comprise the CFDP Software. The basic class defining the CFDP Software is called *TCFDPCore*. Furthermore, because the CFDP Extended procedures supplement and rely on the capabilities provided by the CFDP Core procedures, a second class called TCFDPExtended has been implemented by deriving it directly from the *TCFDPCore* class. It is worth noting that the OOP technique for re-using code perfectly matched the need for a functional software module providing all CFDP Core services (*TCFDPCore* class) to the new CFDP Extended component to be implemented (*TCFDPExtended* class).

This SAD is focused on the description of CFDP Component software for Core procedures, and explains what hides inside the CFDP Component from an *implementer* point of view. Moreover, since the entire CFDP code is running in a *Multithread* context, additional diagrams representing threads' function and interaction, as well as CFDP packets In/Out flow, are included. This should ease future maintenance of the code and assist with determining where to add new features to the component's capability.

8.3.8.2 The Delphi CFDP Component

8.3.8.2.1 Component Description

The CFDP component can be defined as a reusable stand-alone software module representing the CFDP behavior (Core/Extended Procedures), with a well-defined interface to the outside world defined by the public or published methods and properties of TCFDPCore and TCFDPExtended *Classes*.

Within the Delphi developing environment, a CFDP *Delphi Component* can be easily dragged and dropped into a form acting as the *CFDP User Software*.

In order to work, the CFDP component needs to be linked to an application (i.e., the CFDP User Software).

Once it is linked to the linked CFDP component, the CFDP User software will be able to submit protocol *Requests* and receive protocol *Indications* back. In other words, the CFDP component receives *stimulus* from the User Software and reacts accordingly, raising *events* when a certain state is reached. From the CFDP point of view, received stimulus can be associated with all the *CFDP Request Service Primitives* and raised events can be associated with all of the *CFDP Indication Service Primitives* (see figure 8-6).

When the CFDP Component raises an event, the connected CFDP User Software shall be able to handle it and to undertake all appropriate actions (i.e., display an info box to the user or update a log window). This can be done by linking an *event handler procedure* implemented by the User Software to each Component's event.

It is clear that a *Delphi component* can be used only from inside a *Delphi IDE*. For this reason, an *ACTIVE X* version of this component is under development. The Active X standard is a technology, built on top of Common Object Model (COM) technology (COM-Based) that allows the component to be 'Language Independent'. That is, an Active X component can be potentially linked to user software developed under any development environment able to import Active X objects (e.g., Visual C++, Visual Basic, etc.).

The COM technology defines an API and a binary standard for communication between objects (i.e., a CFDP Entity) that is independent of any particular programming language or (in theory) platform.

A COM object consists of one or more 'interfaces' which are essentially tables of functions associated with that object. In this way, a COM mechanism handles all the intricacies of calling functions across process and even machine boundaries, which makes it possible to access an object (CFDP Entity) located on machine A from an application (User Software) running on machine B.

This *inter-machine* communication method, also called *Distributed COM* (DCOM) technology, is only mentioned to point out the possibility of having remote user software interacting with the CFDP component across machine boundaries.

8.3.8.2.2 Instructions for Use of the Component

Once the CFDP User Software 'owns' a CFDP Component object (i.e., either a CFDP Component has been dragged and dropped on the *User Interface form* at *design-time* or an *instance* of the TCFDP Class has been created at *run-time)*, the *creation* method will perform all the initialization procedures of a CFDP Entity. The next step is to link all of the CFDP component's *Events* to appropriate *event handlers* defined by the User Software. From now on the CFDP Component is ready to interact with the User Software (i.e., being set, receiving stimulus and raising events).

In OOP, *Public* and *Published* are used to specify visibility of a certain method or property belonging to a class. The *Public/Published* methods and properties of the *TCFDPExtended* and *TCFDPCore* Classes define the interface of the related component to the outside world (CFDP User Software and CPSC).

The *Public* methods and properties can be accessed only at run-time, while the *Published* properties (not methods) are also accessible from inside the IDE at design-time. This makes it possible to set values for the component's *Published* properties before running the application (i.e., User Interface software) that makes use of the component.

8.3.8.2.3 The CFDP Log Window

During the initialization phase, the *CFDP Component* automatically creates a *Log Window* to display its status and the run-time information for all of the file delivery Transactions handled by the current CFDP Entity (types of PDUs sent and received, error log, etc.). See figure 8-10.

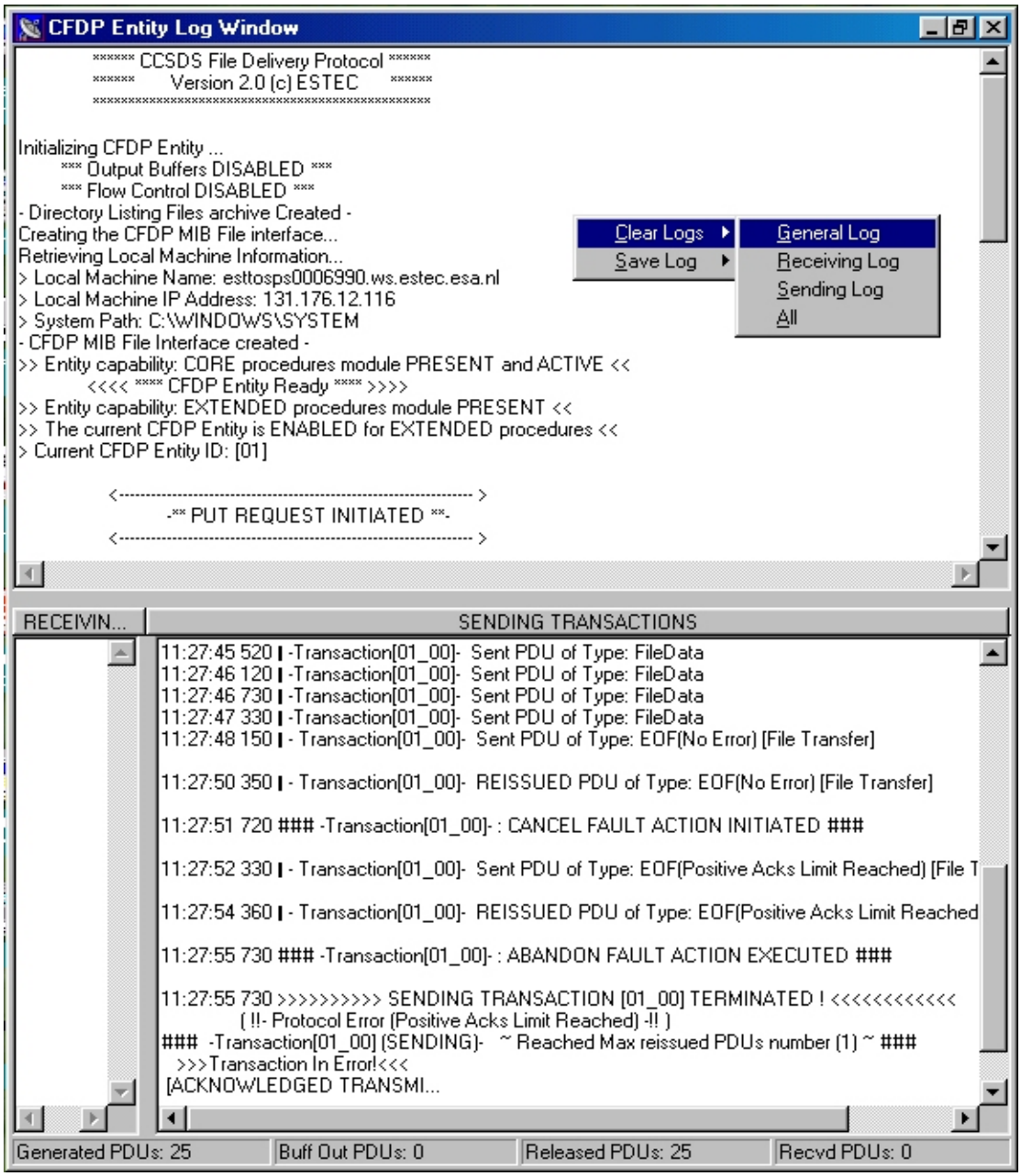


Figure 8-10: CFDP Component's Log Window

The CFDP Log window is divided into three parts:

a) *General* Log (upper part);

b) *Receiving* Transactions Log (left side);

c) *Sending* Transaction Log (right side).

The *General* Log is used for displaying all of the entity's generic messages such as entity status, capabilities and settings values, as well as messages on the transaction's start and end, timers, etc.

The two remaining parts are more Transaction-oriented, displaying details on each CFDP transmission (e.g., received or sent packet, timeouts and error messages) for both *Sending* and file *Receiving* transactions. (Note that a transaction is here defined as *Sending* when it is initiated by the 'local' CFDP entity.) On the contrary, a transaction is defined as *Receiving* when it is initiated by a 'remote' CFDP entity and it involves the local CFDP entity either as an intermediate waypoint or a final destination.

The bottom part of the Log window displays:

a) the total number of PDUs *Generated* by the Local CFDP Entity (but not yet released to the UT layer);

b) the total number of PDUs *Buffered* in the Output Buffers and waiting to be released to the UT layer (a conceptual underlying communication system) at the first transmission opportunity;

c) the total number of PDUs actually *Released* by the Local CFDP Entity to the UT Layer;

d) the total number of PDUs *Received* at the Local CFDP Entity.

By right clicking on the Log window, a pop-up menu will appear. It allows the user to *clear* or *save* a selected log section. Note that each Log section is cleaned every 350 lines, and a file containing the section dump is automatically saved with the name format <*yy_mm_dd hh_mm_ss.txt*> (current date and time) in the directory '…\CFDP_Temp\Log\' of the current Drive. Furthermore, when the CFDP application is closed, it also saves a file containing the dump of all log sections. See figure 8-11 for the log files name format.

The CFDP packets input flow and threads interaction is diagrammed in figure 8-12. The CFDP packets output flow is diagrammed in figure 8-13.

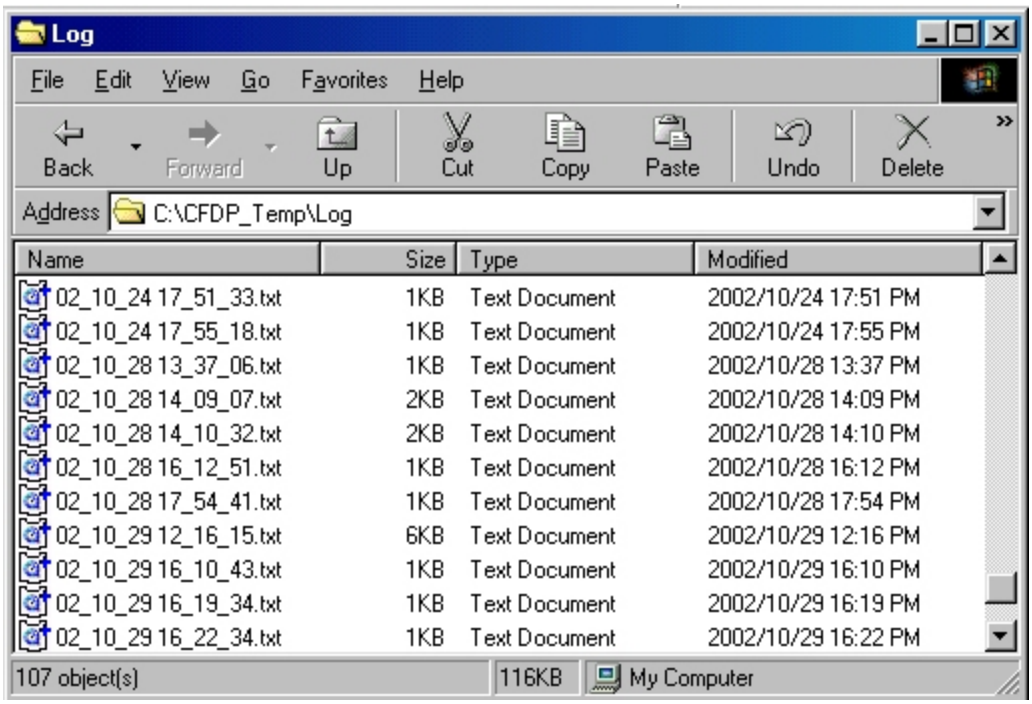


Figure 8-11: Log Files Name Format

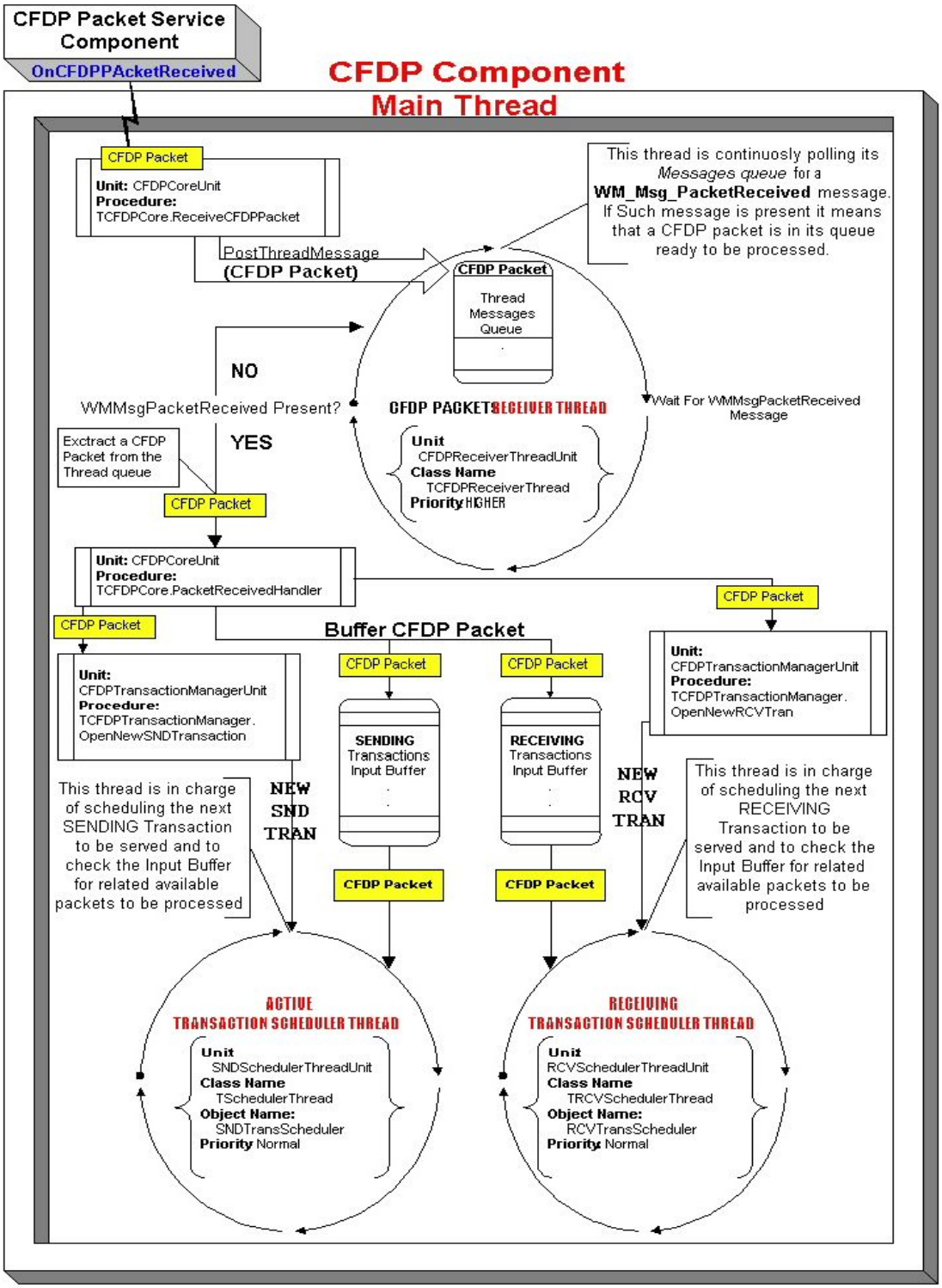


Figure 8-12: CFDP Packets Input Flow Diagram and Threads Interaction

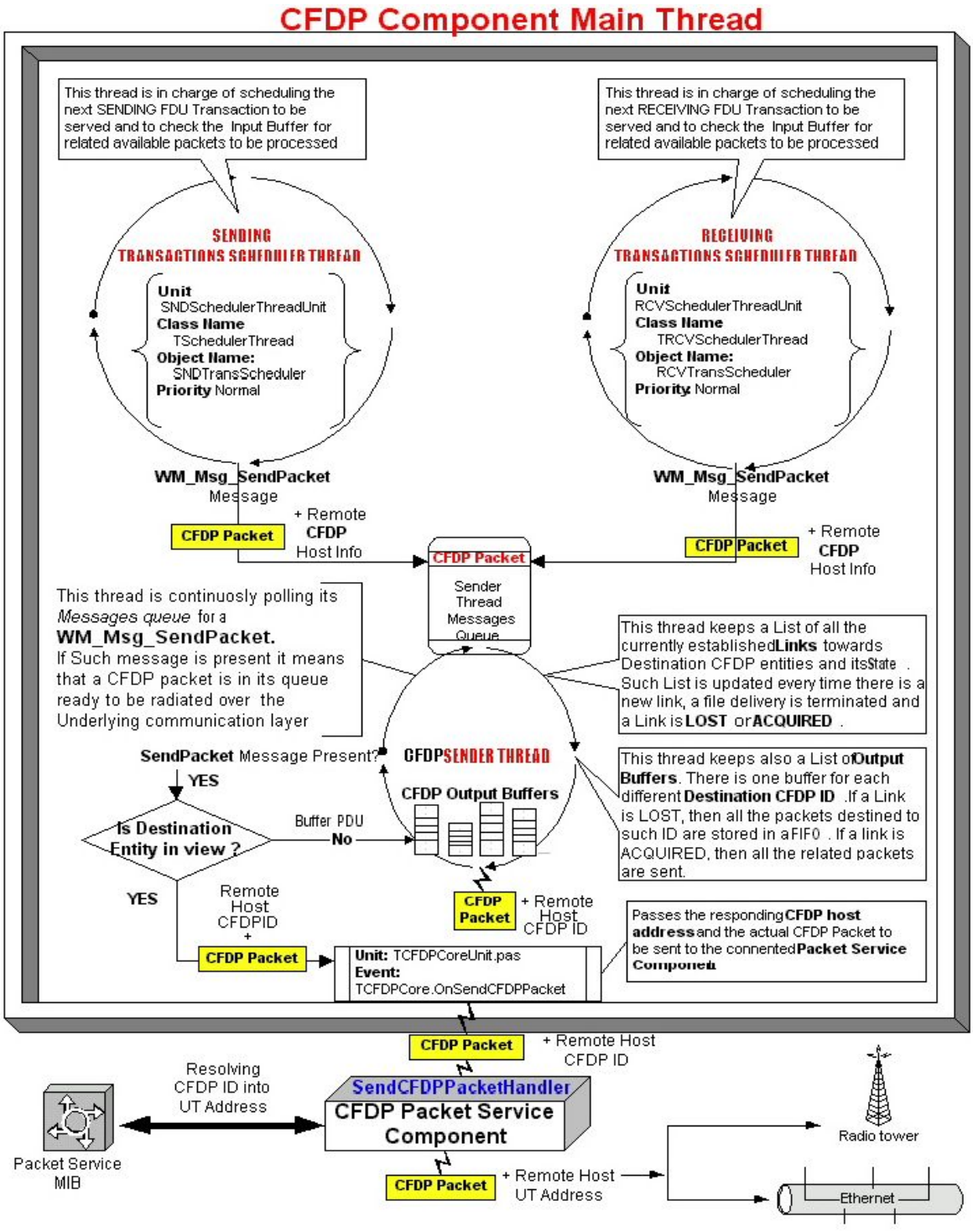


Figure 8-13: CFDP Packets Output Flow Diagram

8.3.8.2.4 The CFDP Component's Messages Window

To fully understand the output flow of CFDP packets inside the CFDP software component, a brief description on how such a component handles all the internal Windows *messages* is necessary.

In order to perform proper actions upon the occurrence of a certain event, due to the multi-threading nature of the CFDP component, user-defined Windows messages are sent from all the secondary threads to a *component's internal window*. In so doing, the messages can be processed within the component's main thread, avoiding violation of shared resources. Hence, the CFDP internal window can be seen as a non-visible 'housekeeping' window performing all of the component's message-handling procedures.

The message types handled by this form are as follows:

a) CFDP_ExtendedMESSAGE;

b) CFDP_ErrorMESSAGE;

c) CFDP_TimerMESSAGE.

8.3.8.2.5 An Outgoing PDU Through CFDP

Every time a scheduled CFDP transaction needs to send a PDU, the Generate_PDU method is called at transaction level from both the scheduler threads (*Sending* and *Receiving* transactions). This method fulfils the following tasks:

a) retrieves the receiving host's CFDP network address (Next Hop ID);

b) extracts the outgoing CFDP packet from the Transaction object;

c) retrieves the action to perform on local timers (enable/disable) upon 'releasing' over the underlying communication layer;

d) performs the CRC calculation on the outgoing PDU (if necessary);

e) stores the PDU and all the related information in a memory structure (Transaction ID, Destination ID, Packet number, etc.);

f) posts a *WM_MSG_SendPacket* message containing the address of the newly allocated memory structure to the *Sender thread*'s messages queue and returns.

All of the messages posted to a thread are buffered in the *thread's message queue* before the thread itself processes them. The Sender thread main code (Execute method) is a loop which is continuously polling for *WM_Msg_SendPacket* messages. When it is found and processed, if the Destination CFDP entity is currently 'In view', the packet is finally *released* on the underlying communication layer via the UT layer interface, and the next pending message is processed (unless the *Flow Control* mechanism is enabled). The CFDP sender thread issues an *OnRadiatePDU* event which in turn issues an *OnSendCFDPPacket* from the CFDP component.

If the Destination CFDP entity is 'not in view', then the packet is buffered in output queues (internal to CFDP) made by persistent FIFO linked lists, and it will wait for the next link acquisition towards that particular destination CFDP ID. The FIFO lists grow while links are inactive and shrink while they are active, but this is transparent to applications using the *CFDP Component*.

An ***Output Buffer*** is created for each 'new' CFDP destination entity involved in a file delivery transaction. In this case, destination entity means the final destination of a FDU if no waypoints are present along the communication path; otherwise it means the next hop CFDP ID towards the final destination.

Thus, the ***Output Buffer*** contains outgoing packets belonging to both *Sending* and *Receiving* transactions handled by the local CFDP entity. The CFDP component is also responsible for keeping track of all the new established, lost, acquired or dismissed links towards different destinations. This task is carried on by the *CFDP component* itself in conjunction with the *CFDP User Software* (which issues the *Link Lost/Acquired* signals during transactions lifetime), as follows.

a) **LinkLost** (Remote_CFDP_ID);

b) **LinkAcquired** (Remote_CFDP_ID).

Obviously, this implies that such knowledge already exists outside of CFDP (at the CFDP User Software level). Both *Core* and *Extended* procedures will benefit from such a *Deferred Transmission mechanism*, giving the CFDP a way to 'drive' the starting and stopping of PDU transmission and schedule the file delivery transactions according to an arbitrary priority scheme.

The use of output buffers can be disabled from the CFDP User Software (both during design and run time) in case the CFDP component is running on a storage-constrained entity. Unfortunately, this implies the loss of all the packets currently stored in the outgoing FIFO linked lists (see figure 8-14). The maximum number of PDUs that a single list can buffer can also be set. Therefore, a full list will discard any further PDU that needs to be buffered.

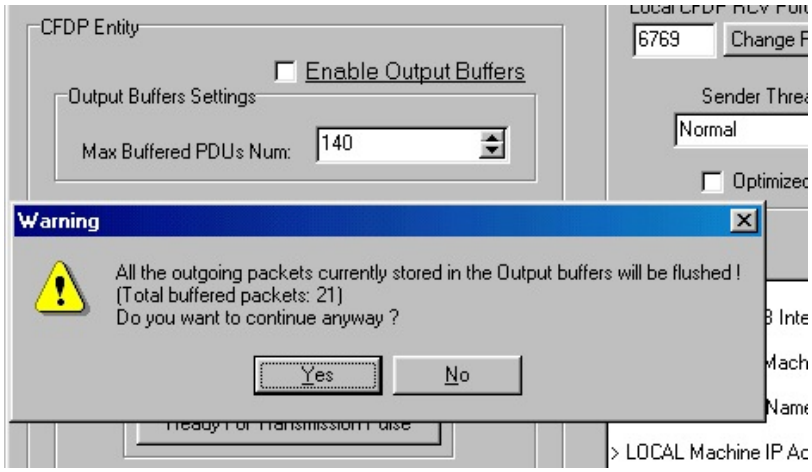


Figure 8-14: Enabling Output Buffers with User Software

8.3.8.2.6 Advantages of Deferred Transmission

By relying on *link state output queues* to control the operation of File Delivery Protocol output, we can accommodate occultation and other interruptions in connectivity simply and efficiently: when the link is lost, the CFDP User Software simply commands CFDP to stop 'all' transmission of data towards the opposite end point of the link' by internally buffering all of the outgoing PDUs for that specific destination. Furthermore, the joined use of *Transmission/Reception Opportunity* CFDP Procedures provides greater flexibility and efficiency by allowing 'freezing' of all transactions involved in the link loss.

The freezing of transmission/reception for a transaction has the same effects as suspension of that transaction by the sending/receiving entity, except that no *Suspended. Indication* is issued and the transaction is not considered suspended.

Given that the same action is performed at the other end of the lost link, this implementation of deferred transmission incurs far less protocol overhead than using the 'remote' *Suspend* and *Resume* operations to control suspension and resumption of communication. *Remote Suspend* and *Resume* procedures are protocol elements, requiring a co-operative interchange of data between entities.

Deferred transmission is entirely local; no PDUs are issued or received to affect it. Because *deferred transmission* is an entirely local mechanism, it is unaffected by delay due to the distance between the participating entities. Moreover, there is no chance of incomplete suspension/resumption due to loss of PDUs. *Remote Suspend* and *Resume* procedures are transaction-specific. This means that a link cut between any pair of CFDP entities would require the reliable transmission of Suspend messages for every transaction currently in progress between them, and resumption of transmission would require the reverse. In contrast, the deferred transmission mechanism is atomic and comprehensive.

8.3.8.2.7 Flow Control integration in CFDP

In addition to the need for handling loss of link visibility, a mechanism for throttling CFDP into sending no faster than the receiver can handle the traffic was also found to be necessary. Thus, an efficient *Flow Control* mechanism in which the receiver (UT Layer interface) provides feedback to the sender (i.e., CFDP component) was implemented.

Assuming that the communication channel is error free, and in case the used link has an uneven data throughput (i.e., Packet Telecommand link), a good solution has been found in the *Stop-and-wait* flow control protocol. It is a mechanism whereby the sender sends one frame and then waits for an acknowledgment before proceeding.

This behavior could also have been accomplished by using the *LinkLost()* and *LinkAcquired()* procedures available at the CFDP component, but a dedicated entry point has been implemented in order to drive flow control procedures in a more easy and efficient way.

For this purpose, the CFDP component has been provided with an *UT_ReadyForTransmission()* method that can be used by a software module located at the UT service layer interface (i.e., *the CFDP Packet Service Component*) in order to stimulate the CFDP component to 'release' PDUs whenever the receiving side is ready.

When this method is invoked, the status of an internal auto-reset CFDP event object is signaled, allowing the CFDP *Sender thread* to resume its execution, release a CFDP PDU, and suspend execution again until the next stimulus.

This capability would spare the CFDP component the use of an 'embedded' Flow Control algorithm. In other words, the network packets received by the CFDP component will still be pure CFDP PDUs, since the flow control header has been read, interpreted and filtered by an 'external' interface software module in charge of driving the CFDP packets flow via the available functions (i.e., *UT_ReadyForTransmission method*). In this way the *Flow Control* mechanism result is transparent to the CFDP component itself.

8.3.8.2.8 Transaction Priority Considerations

The PDUs pending in the sender thread's message queue are already stored in a PRIORITY order, since they have been generated by the Receiving or Sending Transactions SCHEDULERS according to their defined SCHEDULING ALGORITHM.

If such PDUs are extracted and buffered because of a link visibility cut, then the previously assigned priority is lost. This happens because the ordering key for buffered PDUs is no longer their Transaction ID but, instead, their Destination ID. Therefore, all the PDUs stored in an Output Buffer belong to transactions of different nature and IDs, but are all destined for the same remote host.

In other words, a new kind of priority is established between *buffered* outgoing PDUs:

The Output Buffer CREATION ORDER.

In practice, a PDU destined to a Destination ID that was *out-of-view* in a moment X will be released from the from the output buffer queue (as soon as the link is acquired again) before the PDUs destined to a Destination ID that was *out-of-view* in a moment X+Y.

NOTE – CFDP does not allow assigning a priority to a single PDU. On the other hand, a priority can be assigned to an FDU (functional concatenation of data and related metadata) by means of the Flow Label TLV message of the protocol.

8.3.8.2.9 Transaction Scheduling Algorithm

Currently, the scheduling algorithms adopted by the *Receiving Transactions Scheduler* and the *Sending Transaction Scheduler* are quite similar, with only a few minor differences.

Every transaction (Sending or Receiving) is initially assigned a Priority Level (0-255) on creation, but the priority may also be modified at run time with the User Software.

The scheduling algorithm takes into consideration the critical situation of the transaction entering the sending or receiving 'close loop'.

In case this loop is entered, it will be interrupted for scheduling as follows:

a) A *Sending*/*Receiving* transaction that enters an ERROR state.

b) A *Sending* transaction that needs to REISSUE an EOF PDU or a *Receiving* transaction that needs to REISSUE a:

   1) Finished PDU;

   2) Keep Alive PDU;

   3) NAck PDU.

c) A NEWLY CREATED *Sending* transaction.

If there is a group of two or more *Sending*/*Receiving* transactions claiming to be scheduled for the same above-mentioned events, or if the *Sending*/*Receiving* transaction is not in a sending/receiving close loop, then the NEXT *Sending*/*Receiving* transaction will be scheduled from the considered group according to the transaction's *Priority Level* and *Creation Time*. That is if two or more transactions have the same *Priority Level*, then the earliest created is scheduled.

8.4 JHU/APL IMPLEMENTATION REPORT

NOTE – Contributed by Christopher J. Krupiarz, The Johns Hopkins University (JHU) Applied Physics Laboratory (APL).

8.4.1 INTRODUCTION

This report describes the flight software implementation of CFDP on the MErcury, Surface, Space ENvironment, GEochemistry, and Ranging (MESSENGER) spacecraft. MESSENGER is a NASA Discovery mission to study the planet Mercury. CFDP will be used on MESSENGER to downlink science data, images, and telemetry packets.

8.4.2 SOFTWARE ENVIRONMENT

The software is written in the C programming language and runs on the VxWorks Real-Time Operating System. The flight hardware consists of a RAD6000 processor running at 25MHz, 8 Megabytes of RAM, and 4 Megabytes of storage for the complete code image. The processor runs both Command and Data Handling (C&DH), Guidance and Control (G&C), as well as various other processes including image compression. Due to the high processor load and memory constraints, it was determined that a custom implementation was to be developed for the flight system in order to have greater control over system resources. On the ground system, an implementation provided by NASA/JPL was integrated into the current JHU/APL architecture, thus reducing the need for further core CFDP development within the ground software.

8.4.3 SOFTWARE OVERVIEW

8.4.3.1 CFDP Selections

The software running on the flight system implements a subset of the CFDP protocol. Acknowledged Mode was selected to ensure receipt of the files, and Deferred NAKs are used due to the large one-way light time between Mercury and Earth. MESSENGER's use of CFDP was also simplified in several ways to reduce the impact on margins for CPU and memory usage, as follows:

a) *File transfer*: Files are transferred only one way, from the spacecraft to the ground. Previous proven software developed by JHU/APL is used to upload data directly into memory in place of the uploading of files.

b) *File system operations*: File system operations, such as move and delete, are performed via native flight software commands as opposed to within CFDP.

c) *Initiation of file transfers*: File transfers on the flight side are initiated automatically or through commanding outside of the CFDP framework. Although CFDP provides the capability to initiate these transactions from the ground through a proxy service, MESSENGER file transfers from the ground are started via native flight software commands that instruct the CFDP flight software to downlink files.

The UT layer for MESSENGER consists of CCSDS telecommands and CCSDS transfer frames. Incoming PDUs are packaged and treated as a command by the flight system and are therefore processed through the command execute thread. Outgoing PDUs are sent on a separate Virtual Channel to the ground software.

8.4.3.2 Software Architecture

8.4.3.2.1 Overview

The MESSENGER flight software implementation consists of a primary store of information called the transaction table, lists maintained for timers and files, and two processing components, a 1-Hz CFDP task and CFDP methods. The 1-Hz task operates independently, while the methods are called by two other tasks within the MESSENGER flight software architecture.

8.4.3.2.2 MESSENGER Transaction Table

The transaction table stores information on all of the current transactions. See table 8-1.

Table 8-1: MESSENGER Transaction Table

Field	Size (in bits)	Purpose
id	32	CFDP Transaction Sequence Number
transaction_timer	32	Timer for the entire transaction
eof_timer	32	Timer for the EOF ACK
file_size	32	Size of this file
current_location	32	The location within the file to start the next PDU
nak_start	32	Start of next NAK block
nak_end	32	End of next NAK block
checksum	32	Checksum for the file (created as PDUs are being generated)
next_timer	32	Next timer in transaction timer list
next_eof	32	Next timer in EOF timer list
nak_links	64	Links to the previous and next NAKs
fdu_links	64	Links to previous and next FDUs
eof_count	8	Count of EOFs sent

Table 8-1. MESSENGER Transaction Table (continued)

Field	Size (in bits)	Purpose
filename	480	Name of the file to downlink (MESSENGER flight and ground file systems mirror each other, thus the filename for the ground and flight will be the same)
post_action	2	Action to perform after transaction (move to a trash directory or delete)
state	1	Current PDU state of transaction (indicates whether to create an FDU or a NAK)
priority	1	High or low priority
metadata_sent	1	Indicates whether metadata has been sent
timer_active	1	Indicates whether the EOF timer is active and therefore decremented
naking	1	Indicates whether the entry is in the NAK state
eof_sent	1	Indicates whether an EOF has been sent

Given the distance to Mercury and the average file size, it is estimated that at any one time, six hundred transactions may be open. These open transactions are defined as those currently being transferred or those awaiting information from the ground in the form of NAKs, EOF ACKs, or Finished PDUs.

8.4.3.2.3 MESSENGER Lists

The flight software maintains four lists. Two are for timers, the other two are for the processing of PDUs. The lists are as follows:

a) *Transaction list*: This is a timer list for the overall transaction time. When a file is added to the transaction table, an entry is made in this list.

b) *EOF list*: This is a timer list used to time the wait for an EOF ACK from the ground. When an EOF is sent, an entry is made in this list.

c) *FDU list*: This is a prioritized list from which to choose the next transaction from which to send a PDU.

d) *NAK list*: This is a prioritized list from which to choose the next transaction from which to send a PDU derived from a NAK request.

8.4.3.2.4 MESSENGER Tasks

8.4.3.2.4.1 Overview

The MESSENGER flight software consists of numerous VxWorks tasks. The tasks responsible for CFDP are the Command Executive, Playback, and CFDP.

8.4.3.2.4.2 Command Executive Task

Uplinked PDUs are processed within the standard command processing architecture. When a telecommand packet is received an opcode, which determines where the command is routed, is extracted from the data. In the case of CFDP, the opcode indicates a method should be executed which will process the given PDU. Since the flight software is only a CFDP sender, the PDUs received will be limited to NAKs, EOF ACKs, and Finished indicators. They are processed as follows:

a) *NAK*: Because a single NAK can contain multiple retransmission requests, the NAK is divided into potentially several entries in the appropriate internal NAK priority queue. The information contained in each entry consists of the Transaction ID and the start and stop offsets for the NAK. These are processed later by the Playback Task through a method call.

b) *EOF ACK:* The timer is stopped in the transaction table for the Transaction ID in the ACK.

c) *Finished:* A Finished ACK is placed on the outbound PDU queue for this transaction and the transaction is removed.

The Command Executive task may also receive commands from the ground to start a new file downlinking, to cancel a current transaction, and to update the timeout values. Commands exist to process file system operations, although these are handled outside of the CFDP protocol for MESSENGER. File system ops include a directory listing, delete, create directory, move, and copy. Commands which may take an extended amount of time to complete are executed outside the command execute thread by the CFDP 1-Hz task.

8.4.3.2.4.3 Playback Task

The Playback Task is the primary user of CFDP. This task autonomously downlinks files based upon their location in a priority based directory structure. It checks the state of CFDP to determine whether a file is currently downlinking, and, if one is not, it initiates a new file transfer. The CFDP method adds the requested file to the transaction table at this point. As space becomes available in the downlink buffers, the Playback Task requests PDUs from CFDP, which are selected from one of several sources. EOF and Finished ACKs are given priority. If none is available, the NAK queue is checked for retransmission requests. If there are no NAKs to process, CFDP will provide the next file segment from the file currently being processed.

8.4.3.2.4.4 CFDP Task

The 1-Hz CFDP task is responsible for file system operations and for decrementing the timers for EOF PDUs. The CFDP task checks whether any file system commands are waiting to be executed and executes them. Using a resolution of one minute, it also decrements the EOF ACK timers and transaction timers if the CFDP is in the active state. When it determines that the wait for an EOF ACK has timed out, it builds another EOF PDU and places it on the outbound PDU queue for future processing. If a transaction time out occurs, it cancels the transaction.

8.4.4 COMMENTS AND LESSONS LEARNED

8.4.4.1 Suspend and Resume

Originally the flight code was designed to allow for the suspension and resumption of transactions. However, after further study it was concluded that the feasibility of managing transactions with a large round trip light time was not practical.

8.4.4.2 Downlinking

MESSENGER data will be downlinked during 8-hour passes. In lieu of suspend and resume, the timers on the ground and flight side are turned off at the end of a pass and turned back on at the beginning of a pass. Since EOF ACKs and Finished PDUs only result in changes to the transaction table, they are processed continuously. However, PDUs destined for the ground are only created when the Playback task requests the information.

8.4.4.3 Timer Lists

Timers are maintained using a delta list as described in reference [5].

8.4.4.4 Test Harness

Included in the NASA/JPL CFDP implementation is a test harness that allows for the operation of CFDP with UDP as the UT. Through this method, it was possible to test the flight system with the JPL code prior to completion of the actual UT software to be used during the mission. This reduced the degree of integration of issues and ensured that the flight software was properly following the protocol.

8.4.4.5 System Integration and Portability

This implementation of CFDP is not standalone software as it is integrated into the two tasks within the MESSENGER flight software. As JHU/APL uses similar architectures on its missions, it is readily adaptable to future inter-lab missions, although its use outside of JHU/APL would require a larger degree of modification. As stated previously, this was a trade-off for reducing system usage. However, as seen with the treatment of the NASA/JPL software on the ground as a 'black box', a standalone version would promote more reuse on an interagency level.

8.4.4.6 Applied Physics Laboratory/Johns Hopkins University MESSENGER Spacecraft Lessons Learned (21 October2004)

MESSENGER is a Discovery class mission that launched on August 3rd of this year and is destined for a one year orbit around the planet Mercury in 2011. This will be preceded by two flybys of Venus and three flybys of Mercury as well as a flyby Earth one year after launch. The spacecraft's primary computer consists of a RAD6000 running at 25MHz, with 8 Megabytes of RAM, and 4 Megabytes of storage for duplicate code images. Both Command and Data Handling (C&DH) and Guidance and Control (G&C) run on the same computer. The Solid State Recorder (SSR) consists of 8 Gigabits of SRAM. The VxWorks operating system is used including the file system capabilities of the OS. Files on MESSENGER consist of both science data as well as housekeeping data and CFDP is used one-way for downlinking these files from the spacecraft. All files are sent using Acknowledged Mode with Deferred NAKs. Due to the computing constraints on the system, the flight version of CFDP was developed in-house at JHU/APL in order to have full control over its operation. The ground CFDP element as created by JPL and integrated into JHU/APL's ground software. Excluding directory listings, files are generally downlinked autonomously based upon a priority directory/naming scheme.

The spacecraft has been operating nominally since launch and has downlinked a couple thousand files for a total of a 2-3 gigabytes of data. The primary problems encountered post launch were due to the setting of the timers on both the spacecraft and the ground. This caused a handful of transactions to time out prematurely thus resulting in the resending of data. These circumstances result in the following lessons learned:

1) Allow Room in Default Timer Settings--all of the initial problems encountered with CFDP could have been avoided through the proper setting of the various timers within the CFDP system. The default settings for these timers were insufficient to accommodate the various hiccups that may occur during checkout as well as the speed at which round trip light time would influence these variables. Since extra bandwidth was available during this time frame, a better method would have been to have given quite a bit of room for these timers to ensure that the system was working in general and only when the bandwidth constraints would soon require it, proceed to more a tightly constrained environment.

2) Lead CFDP Engineer--these initial problems could have been alleviated by having treated CFDP as a subsystem and having had one individual responsible for testing and understanding the system as a whole. This benefit is derived from CFDP being a closed loop system where operations on one entity can impact the functionality on another. Although the flight developers knew their system and the ground developers knew theirs, there was not one person who really knew both in a detailed way. When it came time for operations personnel to try to analyze why there were initial file failures, there was no one person to whom to turn.

3) Better Prepping of Operations Personnel--the flight and ground developers worked with operations personnel throughout the development cycle, but it was evident that the intricacies of timer settings was not properly conveyed. Understandably, the operations personnel worked under the assumption that the delivered CFDP would work out of the box. In general it did, provided there was little data loss or the files were kept to a manageable size. However, in anomalous situations of excessive PDU loss or in the case of larger than expected files, the timers were set too tight and the files were not received correctly. Due to an insufficent amount of information being provided to Operations personnel prior to launch, they were not positioned to respond to these anomalies independently.

8.5 NASA/GSFC IMPLEMENTATION REPORT

NOTE – Contributed by Tim Ray, NASA/GSFC.

8.5.1 OVERVIEW

This implementation is based upon the CFDP State Tables and Kernel logic (see section 5). It meets the minimum requirements for Service Classes 1 and 2. That is, it can send and receive files in either Unacknowledged Mode or Acknowledged Mode.

NOTE – This implementation is written in the C programming language. It is the implementer's opinion that the conceptual design of this implementation would be similar (although the mechanics would be different) if the implementation had been written in object-oriented language.

8.5.2 STATE TABLE AND KERNEL CONCEPTS

There are three concepts essential to understanding this implementation, as follows:

a) There is one *state table* for each possible role (e.g., Class 2 Sender). Each state table summarizes 'what to do' in response to all possible events.

b) There is one *state machine* for each active transaction. If there are three active transactions for which my entity's role is Class 2 Sender, then there will be three Class 2 Sender state machines running. All three state machines will execute the same *state table* logic, but each state machine will run independently of the others.

c) The *Kernel* is always active. It receives each incoming PDU or User Request, and takes one of three actions: ignore the input, pass it on to the appropriate existing *state machine*, or create a new *state machine* and then pass on the input.

8.5.3 DESIGN SUMMARY

There are four levels of modules, as follows:

a) CFDP Data Representation (i.e., defining CFDP Protocol Data Units as C data structures);

b) CFDP Core (one module for each state table, plus a kernel module);

c) CFDP Support (e.g., a module to keep track of file gaps);

d) Utility (e.g., stopwatch timer).

One design decision that I have been very happy with was to define C data structures to represent each PDU. Each incoming PDU is immediately converted to a C data structure. All 'internal' manipulations use the C data structures. Outgoing PDUs are converted from C data structures just prior to release. I defined one structure for each PDU, plus a generic structure that can hold any PDU. In object-oriented terms, I think this would be a base class ('Pdu') and derived classes (e.g., 'EofPdu').

Another design decision that I have been very happy with was to develop CFDP Support (or 'infrastructure') modules. The idea is that the core CFDP logic changes, but the infrastructure does not. While I was developing the state table logic, the tables changed radically from one week to the next. While the tables are much more stable now, they will probably be updated slightly over the coming years. On the other hand, actions like 'Transmit Metadata' or 'Update Nak-list' did not change. The result is that my CFDP Core modules (e.g., the Class 2 Sender module) are almost a exact copies of the published state tables. Every action called out by the state tables is a subroutine within a CFDP Support module.

8.5.4 MODULE TREE

Unless otherwise specified, each module consists of an '.h' file that defines its specification and client interface, and a '.c' file that contains its implementation.

Here is the high-level module tree:

Cfdp (the main routine is here)

- User (the user interface)
- Comm (the lower-layer communication interface)
  - Udp (an interface to the User Datagram Protocol)
  - Link_sim (simulates the physical link, e.g., dropping of data)
- Kernel
  - Event (determines 'event number' from a given PDU or User Request)
  - Machine_list (manages a list of state machines; add/delete/match, etc.)
  - S1 (Class 1 Sender state table)
  - R1 (Class 1 Receiver state table)
  - S2 (Class 2 Sender state table)
  - R2 (Class 2 Receiver state table)

Nak (keeps track of file gaps; i.e., which data needs to be resent)

Lower-level module shared by S1, R1, S2, and R2:

Aaa (the state table ‘action’ routines)

Lower-level modules shared by various modules:

Id (variable-length Ids; converts between ‘dotted string’ and ‘number’ Ids)

Mib (access to Management Information Base settings)

Pdu (assembles/disassembles PDUs from/to C structures)

Utility modules:

Data (dynamic allocation of strings and binary data)

File (generic filesystem interface, i.e., opening/reading/writing files)

Timer (stopwatch timer)

Usleep (for sleeping for some number of milliseconds)

Data-structure definitions (these are ‘.h’ files; no accompanying ‘.c’ file):

Machine (a large structure containing all state machine variables)

Pdu_data (defines Pdu fields, e.g., all possible values for ‘Condition Code’)

Struct (defines a structure for each type of Pdu, plus a generic Pdu structure)

8.5.5 ADVICE

If you decide to base your implementation on the published state tables and Kernel, make your state table modules (e.g., ‘class_2_sender.c’ or whatever) essentially a copy of the published state tables. Make each *action* called out in the state tables a subroutine call. For example, if the state table action is ‘Update Nak-list’, make a subroutine call to ‘update_nak_list’ or ‘NakList.update’ or something similar. If you do this, it will be much easier to verify that your modules match the published state tables. Also, it will be much easier to update your modules when the state tables are (inevitably) updated.

8.6 NASDA CFDP IMPLEMENTATION REPORT

NOTE – Contributed by Hiroaki Miyoshi, NASDA/NEC.

8.6.1 INTRODUCTION

NASDA has participated in CFDP development activities from the beginning, and has contributed to the validation of CFDP through a series of software implementations, as well as through the review of CFDP documentation (see figure 8-15).

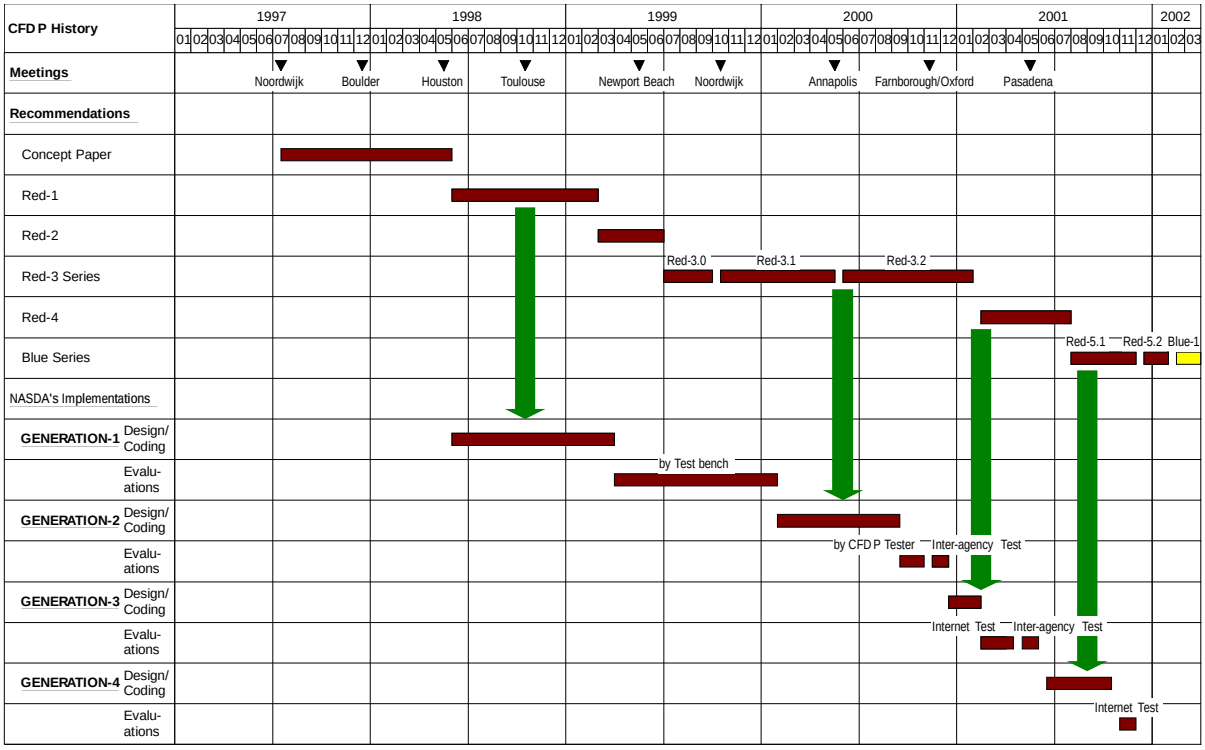


Figure 8-15: NASDA CFDP Implementation History

This implementation report provides detailed descriptions corresponding to the GENERATION-4 implementation, based on final draft Recommendations and Reports.

Subsection 8.6.2 contains an overview of an implementation including policies, environments, scope and architecture.

Subsection 8.6.3 contains detailed information concerning each software component.

Subsection 8.6.4 contains test results and a future plan.

8.6.2 HIGH-LEVEL DESIGN

8.6.2.1 Implementation Policies

The *interoperability* of the implementation was the first priority for inter-agency test.

Portability and *expandability* of the implementation were also emphasized in order to re-use it for the next generation space and ground data systems of NASDA.

8.6.2.2 Implementation Environment

The NASDA CFDP implementation was developed under the Microsoft Visual Studio IDE using the C++ programming language.

All computers used were IBM PC compatibles running Windows NT/2000 OS. A laptop computer was frequently used because its mobility was a great advantage when a face-to-face inter-agency test workshop was held overseas.

The UDP/IP Internet protocol stack over the Ethernet was selected for the subnetwork interface of CFDP in order to enable an inter-agency test over the Internet. CFDP provides a capability for reliable file transfer, which guarantees that all data will be delivered without error, so TCP was not selected for a transport protocol of the subnetwork.

8.6.2.3 Implementation Scope

Table 8-2 describes the scope of the NASDA implementation.

Table 8-2: Scope of NASDA Implementation

CFDP Recommended Standard			Implementation(Yes/No)
CFDP Procedures			-
	Core Procedures		-
		CRC Procedures	No
		Checksum Procedures	Yes
		Put Procedures	Yes
		Transaction Start Notification Procedures	Yes
		PDU Forwarding Procedures	Yes
		Copy File Procedures	Yes
		Positive Acknowledgment Procedures	Yes
		Fault Handling Procedures	Yes
		Filestore Procedures	Yes
		Internal Procedures	Yes
		Inactivity Monitor Procedures	Yes
		Link State Change Procedures	No
	Extended Procedures		No
CFDP Protocol Classes			-

CFDP Recommended Standard			Implementation(Yes/No)
	Class 1: Unreliable Transfer		Yes
	Class 2: Reliable Transfer		Yes
	Class 3: Reliable Transfer by Proxy		Yes
CFDP Protocol Options			-
	End Type		-
		Sender	Yes
		Receiver	Yes
	Put Mode		-
		UnACK	Yes
		ACK/NACK	Yes
	Put NAK Mode		-
		Immediate	Yes
		Deferred	Yes
		Prompted	No
		Asynchronous	No
	Put File Type		-
		Bounded	Yes
		Unbounded	No
	Segmentation Control		-
		Yes or No	No
	Put Primitives		-
		EOF-sent.indication	No
		File-Segment-Recv.indication	Yes
	Fault handler		-
		Ignore	No
		Abandon	Yes
		Cancel	Yes
		Suspend	No
	Cancel Put Action		-
		Discard	Yes
		Retain	Yes
	Filestore Options		-
		Create File	Yes
		Delete File	Yes
		Rename File	Yes
		Append File	No
		Replace File	Yes
		Create Directory	Yes
	Filestore Options (cont'd)		
		Remove Directory	Yes
		Deny File	No
		Deny Directory	No
	Timers		-
		NAK Timer	Yes
		ACK Timer	Yes

CFDP Recommended Standard			Implementation(Yes/No)
		Prompt NAK Timer	No
		Asynchronous NAK Timer	No
		Keep Alive Timer	Yes
		Prompt Keep Alive Timer	No
		Inactivity Timer	Yes
	Counters		-
		NAK Retry Counter	Yes
		ACK Retry Counter	Yes
	Proxy Operations		Yes
	Directory Operations		No
	Remote Status Report Operations		No
	Remote Suspend Operations		No
	Remote Resume Operations		No

8.6.2.4 Architecture

Figure 8-16 shows the architecture of the NASDA CFDP implementation.

The implementation consists of two Windows processes. One is a *User Application Process* and the other is a *CFDP Process*. A *User Application Process*, which is invoked from Windows' GUI, automatically creates and initializes a *CFDP Process*.

User Application Process handles operations and displays from/to an operator. It also intercepts reserved *message to users* for proxy operations and acts as a responding or originating entity of the proxy.

CFDP Process consists of three parts, as follows:

a) *CFDP Core* handles CFDP core procedures that are described in the Recommended Standard.

b) *CFDP Core API* is a sort of 'glue logic' between a user application and the CFDP core logic. It standardizes the representation of a CFDP service interface (*requests* and *indications*) among various user applications in order to maintain portability and expandability of user applications.

c) *Subnetwork API* is also another 'glue logic' between *Subnetwork* and *CFDP core*. It standardizes the representation of a subnetwork service interface, *UNITDATA.request* and *UNITDATA.indication*, among various subnetwork interfaces in order to maintain portability and expandability of *CFDP Core*.

The Winsock is adopted for *Subnetwork* implementation. In order to maintain portability, only the Berkley-compatible socket interfaces are used.

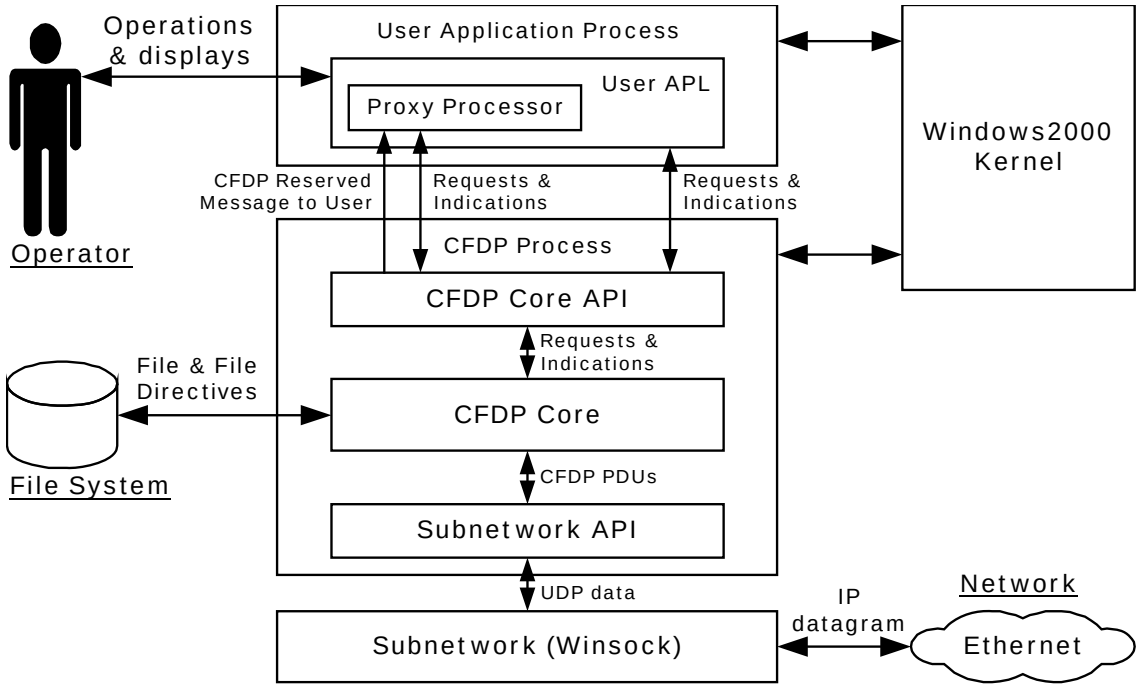


Figure 8-16: The Architecture of NASDA CFDP Implementation

8.6.3 DETAILED DESIGN

8.6.3.1 CFDP Process

NOTE – Figure 8-17 shows the detailed internal structure of the *CFDP Process*.

8.6.3.1.1 CFDP Core API

CFDP Core API provides CFDP service interfaces between a User Application Process via unnamed pipes, which are interprocess communication mechanisms available on Windows NT/2000.

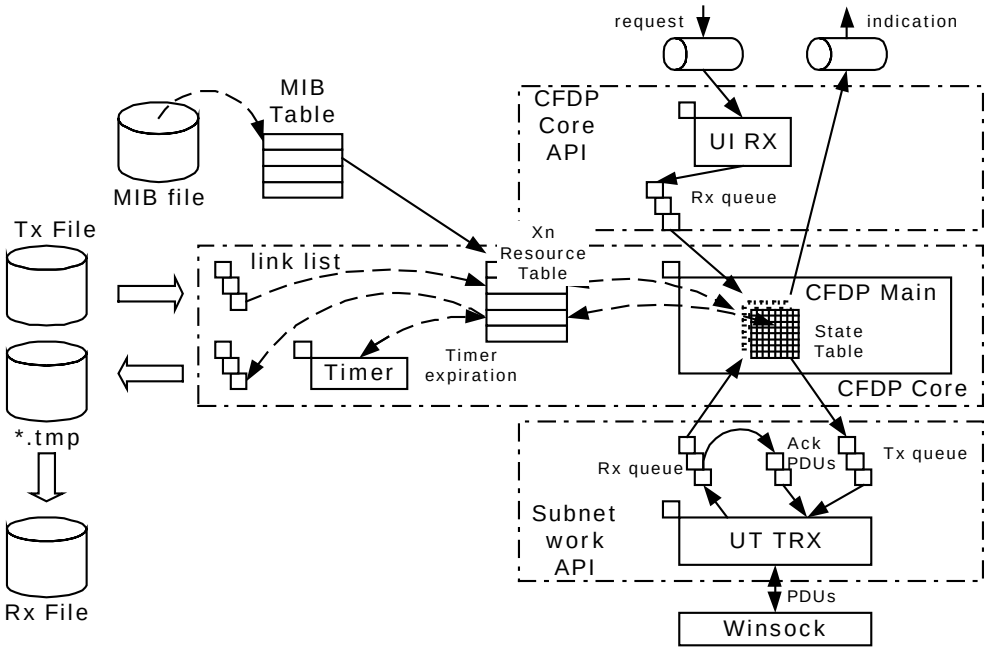


Figure 8-17: CFDP Process

CFDP Core API receives request primitives by means of a '*UI RX thread*', which supervises arrival of messages from an unnamed pipe, and forwards them to *CFDP Core* through a request queue (*RX queue*).

On the other hand, *CFDP Core API* receives an indication primitives by means of a 'service function', which provides some access method for an unnamed pipe.

Since unnamed pipes are also available on other platforms such as UNIX, the NASDA CFDP implementation is easy to port to other operating systems. Also, because CFDP service primitive messages exchanged on pipes are standardized according to the format shown in figure 8-18, it is easy to extend service primitives for implementation-specific purposes in the future.

Msg Block Length	Msg Code	Msg Code Extension	Msg Values (TLVs)
2octets	2octets	2octets	Msg Block Length - 6

Figure 8-18: CFDP Service Primitives Message Format

8.6.3.1.2 CFDP Core

CFDP Core is the main part of the CFDP entity that drives a state table based on the current state and the event that is received from outside, shifts to the next state, and outputs signals.

If a transaction is started by reception of a Request or a PDU, the empty workspace called '*transaction resource table*' will be allocated and the transaction status will be stored there for every transaction ID. The transaction will be setup based on protocol options stored in the MIB Table. The '*Transaction ID*' that is associated with the event identifies which transaction status should be restored to a '*CFDP main thread*' and processed.

Since this architecture is not premised on use of the service that only the specific operating system can provide, it can obtain a suitable processing speed in Windows with a slow process switching speed, and raises platform transplant nature.

The '*timer thread*' holds various timer information which CFDP needs. The event that the *timer thread* detected fault is notified to a *CFDP main thread* through a *transaction resource table* and causes an appropriate state change.

The files that CFDP transmits and receives are managed with a 'file-segment link list (*link list*).' A file divided into file-segments according to the *maximum file segment length parameter* in the MIB and these file-segments are associated with an appropriate link list. *CFDP main thread* uses this list and transmits file-segments in appropriate order.

The received file-segment PDUs are associated with an appropriate link list in order of reception and saved to a temporary file. Missing file-segments are detected from comparison of the file offset that was received last time and this time, and this comparison is used as the source of retransmission requests by NAK. Received re-transmitted file-segments are inserted into the appropriate position of the link list according to the file offset. The temporary file is copied to a target file after a whole file is deemed to be received.

8.6.3.1.3 Subnetwork API

Subnetwork API provides UDP/IP communication services through the Winsock.

The transmitting part of API has two transmitting queues. One is a high priority queue only for ACK PDUs, and the other is a low priority object for other PDUs. By taking this composition, the delay from the PDU reception requiring an ACK to ACK transmission is minimized. This is necessary to make the ACK timer interval setting sensible. *CFDP Core* can transmit PDUs except for ACK PDUs to the other low priority queue by means of a service function.

The receiving part of API supervises a socket via the thread (*UT TRX*). When it detects receipt of a PDU, it stores the received PDU to a receiving queue (*RX Queue*). A read function of the *Rx queue* is provided for *CFDP Core*.

8.6.3.2 User Application Process

A User Application Process mainly takes charge of operations and displays the operator interface. Standard GUI for Windows is adopted as this interface.

Moreover, it contains the processing part that takes charge of proxy operations (*Proxy Processor*). By mounting a *Proxy Processor* apart from *CFDP Core*, a standardization of the interface between *CFDP Core* and an upper layer can be attained, and it provides flexibility to develop future protocol functions, such as Extended Procedures.

The relation between two or more transactions for a proxy operation is managed on the *proxy operation management table* contained in the *Proxy Processor*.

8.6.4 CONCLUSIONS

8.6.4.1 Concluding Remarks

The inter-agency testing over the Internet was carried out at the end of October 2001, and the NASDA CFDP implementation described above talked successfully with NASA/JPL, BNSC/DERA, ESA/ESTEC, and NASA/GSFC.

8.6.4.2 Future Plans

NASDA will deploy this trial implementation to real projects, such as science data management for earth observation satellites. NASDA also plans to evaluate Extended Procedures.

9 REQUIREMENTS

9.1 GENERAL

This section contains the requirements for the CFDP. The development of the requirements was driven by a reference set of five scenarios. These scenarios are included herein. The requirements proper are divided into two subsections: the first lists the requirements for the protocol itself, and the second lists the requirements for the implementation of the protocol.

9.2 CONFIGURATION SCENARIOS

9.2.1 BASIS

Five operational configuration scenarios were used as the basis for the requirements for CFDP. The scenarios are described as both space-to-ground file transfer operations and as ground-to-space file transfer operations. The primary difference for ground-to-space transfers is that most spacecraft are capable of receiving transmissions from only one ground station at a time. Therefore, those configurations implying multiple simultaneous transmissions to a spacecraft in fact have serial non-overlapping access for uplink transmissions.

9.2.2 SPACECRAFT/NETWORK CONTROL CENTER (NCC) WITH NO INTERMEDIATE FILE TRANSFER ENTITY

9.2.2.1 Scenario 1

Scenario 1 consists of End-to-End service using no intermediate File Transfer (FT) entities, as shown in figure 9-1.

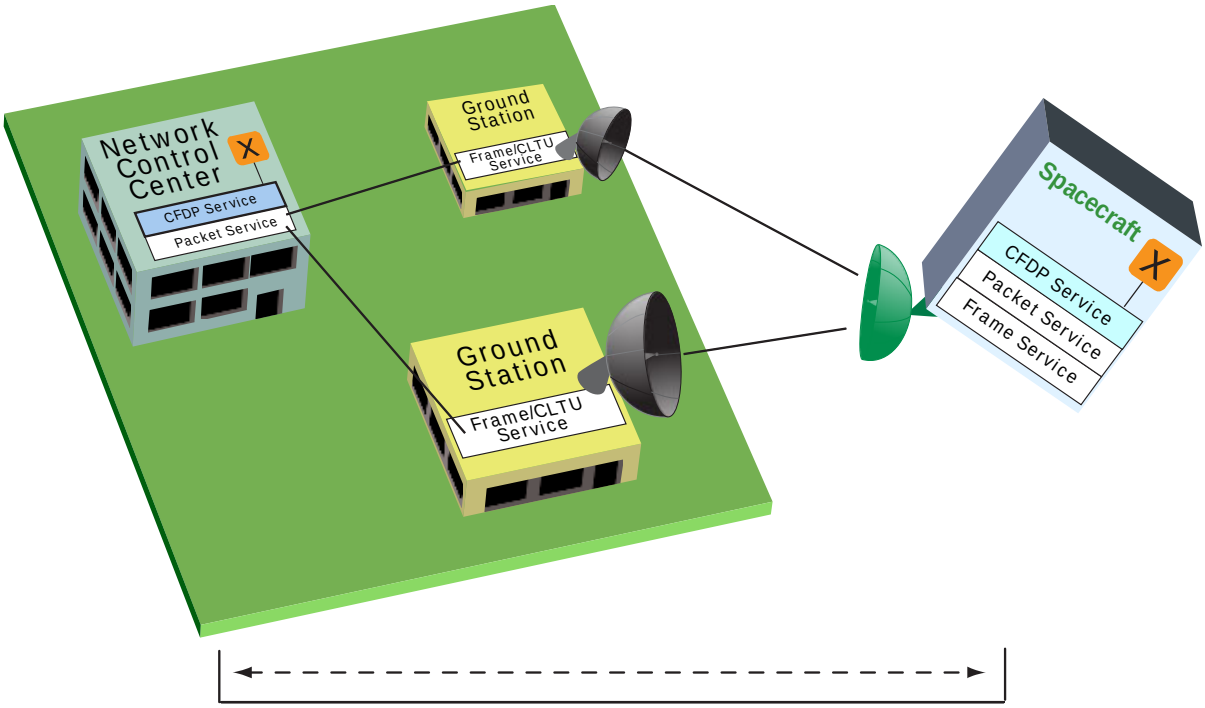


Figure 9-1: Scenario 1

9.2.2.2 Scenario 1: Space-to-Ground

In Scenario 1, the file transfer takes place from a spacecraft to its associated NCC. Multiple ground stations receive frames from the spacecraft and route them to the NCC, with or without extracting packets (i.e., the ground stations may extract the packets using the SLE packet service and forward the packets, or may instead forward the frames, in which case the packets are extracted at the NCC). The ground stations' frame acquisition may overlap one another in time or be entirely disjoint. At the NCC, the packets are passed to the FT entity for assembly and report generation. The reports are routed to the spacecraft's FT entity via the in-view ground station.

NOTE – The NCC's FT entity discards duplicate blocks received during overlapping contacts. The management of frame data at the ground station is not addressed by the protocol.

The NCC's FT entity detects loss and/or corruption of data blocks and requests that they be retransmitted; it also tells the spacecraft's FT entity which blocks it has successfully received. The spacecraft's FT entity retransmits blocks in response to requests from the NCC's FT entity, or in response to determination that an acknowledgment from the NCC's FT entity is overdue (either because the acknowledgment itself was lost, or because the blocks to be acknowledged were not received). The source FT entity (on the spacecraft) continues retransmission until the destination entity (in the NCC) has taken custody of the entire FTU.

Upon notification of complete reception, or upon transaction cancellation (initiated by either of the two FT entities), the spacecraft's FT entity need no longer retain its copy of the FTU in a retransmission buffer. If the data path is simplex (i.e., the NCC can never send data to the spacecraft), then the spacecraft's FT entity assumes that FTU reception is complete as soon as it has finished transmitting the FTU; it may optionally send some or all data blocks multiple times (i.e., 'proactive retransmission') in an attempt to improve the likelihood of successful initial FTU reception.

NOTES

1 The protocol is used to transfer files between space and ground file systems.

2 The protocol can cause file system management commands to be executed with respect to the remote file system (ground or space). FT entities issue those commands in response to file system management command PDUs.

3 The spacecraft can be anywhere in space, from near-Earth orbit to the furthest reaches of the solar system and beyond.

4 Multiple transfers may be in flight concurrently.

5 The protocol may operate over TM/TC packets.

6 Transfers can span link passes (contacts).

7 The protocol delivers a file completion map along with the file (which may be incomplete).

8 A file is defined to be an array of octets (not bits).

9 The 'ground' (the NCC) is a single protocol endpoint, a single FT entity; individual receiving stations are not FT entities in this scenario.

10 The protocol discards duplicate data.

11 The protocol is defined in levels to facilitate a range of implementation complexities from simple to complex. Metadata can command the destination FT entity to:

a) get and put;

b) plus delete, rename, etc.;

c) plus mkdir, rmdir, etc.;

d) perform other functions yet to be defined (e.g., append, rename, patch, read).

12 Support for time-outs: each FT entity involved in one link of a communication path is aware of the one-way light time between the two, and the presumed operative state of the other.

13 Optional features:

a) send and forget (simplex transmission);

b) incremental NAK: the receiving FT entity additionally reports on its reception state (sends a NAK) immediately whenever it detects any missing data block (again, the NAK is automatic, but provides for manual intervention in case of anomaly).

9.2.2.3 Scenario 1: Ground-to-Space

Scenario 1 is also valid for ground-to-space file transfer. In that case, the file transfer takes place, for example, from an NCC to a spacecraft. Multiple ground stations receive packets or frames from the NCC (i.e., the ground stations may insert the packets into frames, or this may be done at the NCC, in which case the ground stations receive frames) and route them to the spacecraft. Because spacecraft usually (with the possible exception of large manned spacecraft) can support only one uplink at a time, the frames are sent to the spacecraft from one ground station at a time, in separate contacts. At the spacecraft the packets are passed to the FT entity for assembly and report generation. The reports are routed to the NCC's FT entity via the in view ground station.

NOTE – The spacecraft's FT entity discards any duplicate blocks which might have been caused by ground station-to-ground station switchovers.

The spacecraft's FT entity detects loss and/or corruption of data blocks and requests that they be retransmitted; it also tells the NCC's FT entity which blocks it has successfully received. The NCC's FT entity retransmits blocks in response to requests from the spacecraft's FT entity, or in response to determination that an acknowledgment from the spacecraft's FT entity is overdue (either because the acknowledgment itself was lost or because the blocks to be acknowledged were not received). The source FT entity (in the NCC) continues retransmission until the destination entity (in the spacecraft) has taken custody of the entire FTU.

Upon notification of complete reception, or upon transaction cancellation (initiated by either of the two FT entities), the NCC's FT entity need no longer retain its copy of the FTU in a retransmission buffer. If, perhaps because of a spacecraft anomaly, the data path is simplex (i.e., the spacecraft cannot send data to the NCC), then the NCC's FT entity assumes that FTU reception is complete as soon as it has finished transmitting the FTU; it may optionally send some or all data blocks multiple times ('proactive retransmission') in an attempt to improve the likelihood of successful initial FTU reception.

9.2.3 SPACECRAFT/USER VIA A SINGLE RELAY ENTITY

9.2.3.1 Scenario 2

Scenario 2 consists of a Hop-by-Hop service using an intermediate store-and-forward process, as shown in figure 9-2.

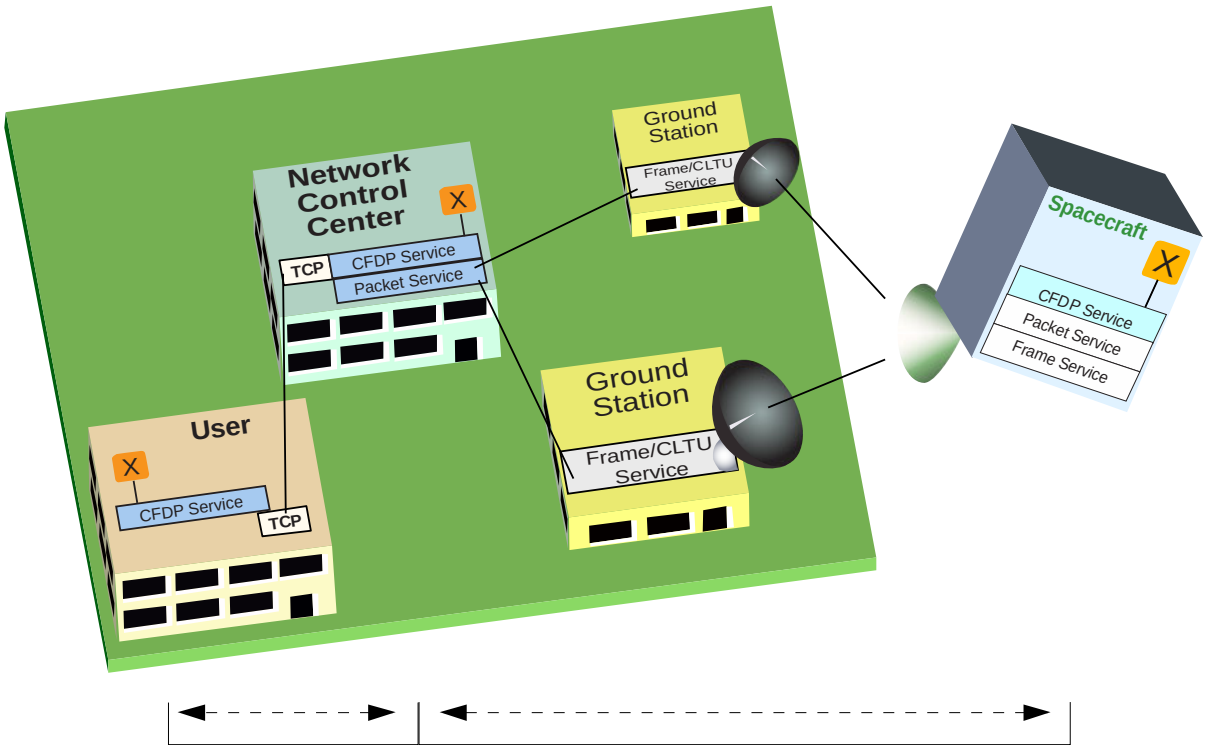


Figure 9-2: Scenario 2

9.2.3.2 Scenario 2: Space-to-Ground

The first Scenario 2 example is a file transfer from a spacecraft to a User via one intermediate entity, the NCC. The User may not always be online, or connection rate limitations might require the NCC to provide store-and-forward delivery. The file transfer from the spacecraft is performed by the NCC's FT entity. The NCC's FT entity serves as a reliable forwarding entity, allowing the spacecraft's FT entity to delete its copy of the file if necessary. File transfer to the User Application is accomplished by the NCC.

NOTE – The NCC's operations with the ground stations and spacecraft are as described in Scenario 1. The protocol can delete the file from the NCC when transfer to the User is accomplished. A protocol status report is sent from the User to the spacecraft.

The source FT entity (on the spacecraft) continues retransmission until the intermediate receiving entity (in the NCC) has taken custody of the entire FTU. The intermediate receiving entity (in the NCC) begins transmission of the FTU to the destination receiving

entity (the User process) as soon as the applicable interim-acquisition rule has been satisfied; this rule might be declared in transaction metadata, or a default rule might be in effect. The intermediate receiving entity continues retransmission until the destination receiving entity has taken custody of the entire FTU, at which time the destination receiving entity notifies the User application.

NOTES

1 The file has proximate as well as final destinations; thus, the protocol has data block relay functionality.

2 There are also final and proximate sources; thus, the protocol has status report-relay functionality.

3 Each intermediate entity has store-and-forward capability; a ground station might or might not be configured as an intermediate entity.

4 The protocol has interim-acquisition rules in effect at each receiving FT entity, for example:

a) forward when N% of the file is received;

b) forward when the link from the sender is lost;

c) forward when the link to the receiver is available.

9.2.3.3 Scenario 2: Ground-to-Space

Scenario 2 is also valid for ground-to-space file transfer. An example is a file transfer from a User to a spacecraft. As in the space-to-ground case, the transfer is via one intermediate entity, the NCC. The spacecraft may not always be online, or connection rate limitations might require the NCC to provide store-and-forward delivery. The file transfer from the User is performed by the NCC's FT entity. The NCC's FT entity serves as a reliable forwarding entity, allowing the User's FT entity to delete its copy of the file if necessary. File transfer to the spacecraft is accomplished by the NCC. As in Scenario 1, because spacecraft usually can support only one uplink at a time, the frames are sent to the spacecraft from one ground station at a time, in separate contacts.

9.2.4 ROVER/NCC VIA MULTIPLE RELAY ENTITIES IN SERIES

9.2.4.1 Scenario 3

Scenario 3 consists of a service from a source through multiple relaying entities in series to a final destination, as shown in figure 9-3.

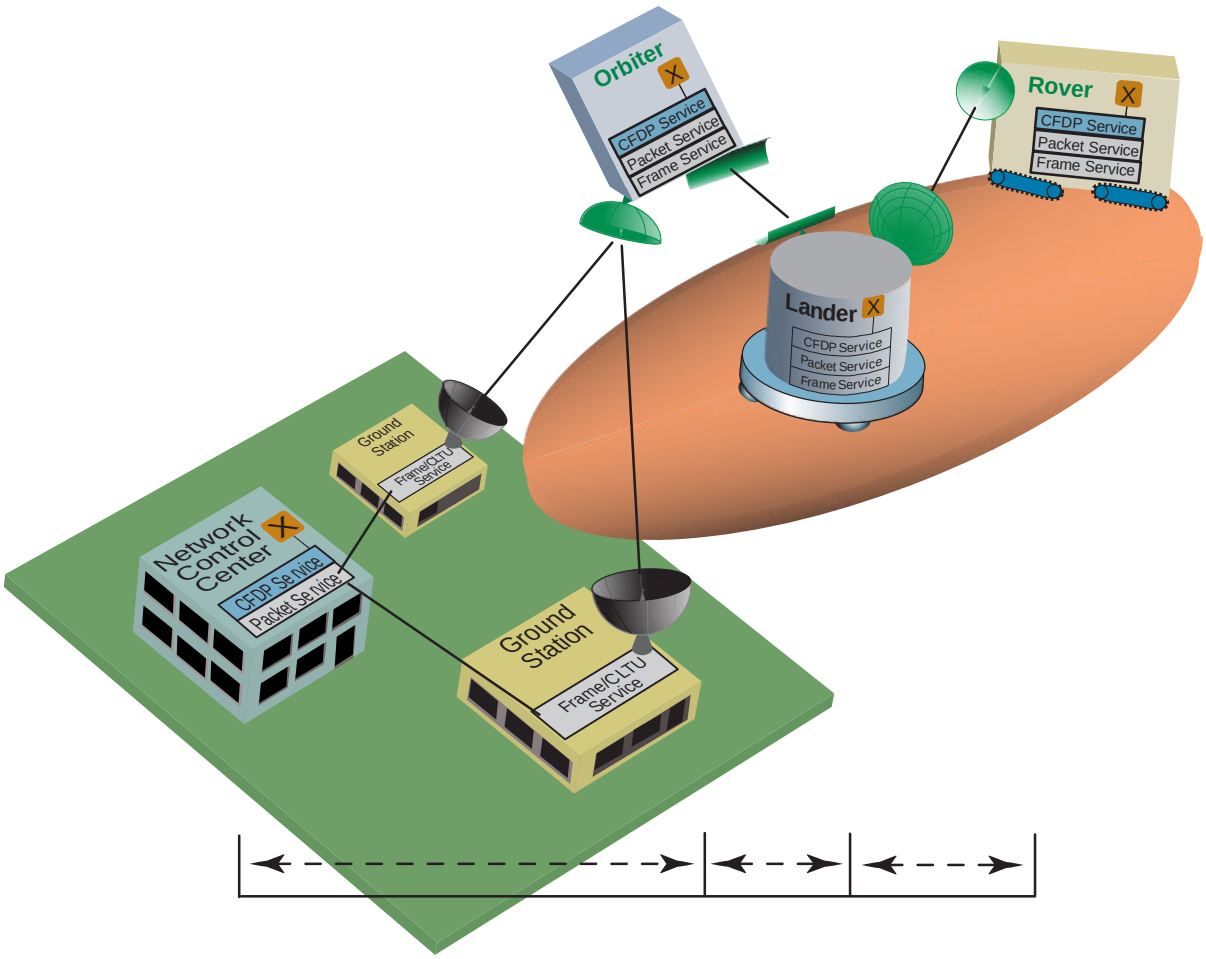


Figure 9-3: Scenario 3

9.2.4.2 Scenario 3: Space-to-Ground

The space-to-ground example is a file transfer from a planetary Rover to an NCC, via a planetary Lander, a planetary Orbiter, and ground stations on Earth. In the example, the Lander and the Orbiter are reliable entities. The files on the Rover and subsequently on the Lander and Orbiter are deleted after acknowledged transfer to the next 'reliable forwarding entity' is completed.

Each intermediate FT entity begins transmission as soon as the applicable interim-acquisition rule has been satisfied (and it has contact with the next FT entity), and continues retransmission until the corresponding receiving entity has taken custody of the entire FTU.

A minor variation of this scenario is to combine it with Scenario 2; i.e., make the NCC another in the series of intermediate entities and add a User application at the destination FT entity for the transaction.

9.2.4.3 Scenario 3: Ground-to-Space

The ground-to-space example of Scenario 3 is a file transfer from an NCC to a planetary Rover, via ground stations on Earth, a planetary Orbiter, and a planetary Lander. In the example, the Orbiter and the Lander are reliable entities. The files in the NCC, and subsequently on the Orbiter and Lander, are deleted after acknowledged transfer to the next 'reliable forwarding entity' is completed.

Each intermediate FT entity begins transmission as soon as the applicable interim-acquisition rule has been satisfied (and it has contact with the next FT entity), and continues retransmission until the corresponding receiving entity has taken custody of the entire FTU.

As in Scenario 1, because spacecraft usually can support only one uplink at a time, the frames are sent to the Orbiter from one ground station at a time, in separate contacts.

9.3 PROTOCOL REQUIREMENTS

9.3.1 GENERAL

This subsection contains the File Delivery Protocol Functional Requirements. For ease of review, they are divided into five groups. These groups are:

a) Requirements Related to Communications.

b) Requirements Related to Underlying Layers.

c) Requirements Related to Structure.

d) Requirements Related to Capabilities.

e) Requirements Related to Records, Files, and File Management.

9.3.2 REQUIREMENTS RELATED TO COMMUNICATIONS

Many of the requirements for the protocol are set by the environment in which it must operate. These include the physical characteristics of the communications links, as well as the availability of those links. The physical characteristics of the communications links include their quality (noisiness), bandwidth, propagation delay, operating mode (Simplex, Half-Duplex, Full-Duplex), and availability. Refer to table 9-1.

Table 9-1: Requirements Related to Communications

Group Num.	Requirement	Req. Ref. Num.	Source
comm 01	The protocol shall be appropriate for both deep space and near earth missions.	01	E11, G1, I1, J15
comm 02	The protocol shall provide effective and efficient service over communications links with propagation delays spanning milliseconds to tens of hours.	02	C4, G3
comm 03	Round trip communications time shall be provided to the protocol from an external source.	66	J37
comm 04	The protocol shall provide effective and efficient service over communications links which are typically bandwidth-restricted.	03	C3
comm 05	The protocol shall provide effective and efficient service over communications links which may be significantly unbalanced in bandwidth.	04	C3, G2
comm 06	The protocol shall provide effective and efficient service when allocation of the available bandwidth is not under the control of the protocol.	05	C1
comm 07	The protocol shall provide effective and efficient service over communications links which have frequent outages.	06	J30
comm 08	The protocol shall provide effective and efficient service over communications links which have long outages.	07	G4, G12, J31
comm 09	The protocol must be capable of providing effective and efficient service over a simplex link.	19	C5, J16, J19
comm 10	The protocol must be capable of providing effective and efficient service over a half-duplex link.	20	C5, E15, J16
comm 11	The protocol must be capable of providing effective and efficient service over full-duplex links.	21	J16
comm 12	Where the underlying protocols can provide the appropriate level of responsiveness, the protocol shall operate when the underlying protocols in both directions provide Reliable service.	22	C1
comm 13	Where the underlying protocol can provide the appropriate level of responsiveness, the protocol shall operate when the underlying protocol in only one direction provides Reliable service.	23	C1, G5, J14
comm 14	The protocol shall operate when the underlying protocols in both directions provide Unreliable service.	24	C1, G5, J14

9.3.3 REQUIREMENTS RELATED TO UNDERLYING LAYERS

The protocol must be able to operate over a wide range of underlying services. Where the underlying services are CCSDS, it must operate over the CCSDS Path Service in Grades of Service 2 and 3. In addition, it must operate over conventional commercial protocols in order to provide required store-and-forward services. See table 9-2.

Table 9-2: Requirements Related to Underlying Layers

Group Num.	Requirement	Req. Ref. Num.	Source
undr 01	The protocol shall provide the capability to operate over current CCSDS Packet Telemetry, Advanced Orbiting Systems, and Telecommand protocols and shall not inhibit the normal operation of these protocols.	11	C8, E2, E3, E4, E5, G9, G10, I12, J2, J26, J27
undr 02	The protocol shall provide the capability to operate over TCP/UDP.	50	E27, J2
undr 03	The protocol shall provide full capabilities over the services provided by existing packet recommendations.	75	E03
undr 04	Full advantage shall be taken of the characteristics of the Packet TM/TC service, i.e., normally 'perfect' data in sequence with possible omissions.	76	E05

9.3.4 REQUIREMENTS RELATED TO STRUCTURE

Two requirements relate to the user-visible structure of the protocol, as described in table 9-3.

Table 9-3: Requirements Related to Protocol Structure

Group Num.	Requirement	Req. Ref. Num.	Source
struct 01	The protocol shall operate between automated, essentially symmetrical peer entities.	09	I3
struct 02	A single service interface will be presented to the client.	10	E10
struct 03	The protocol shall be scaleable so that it may be used on relatively simple, current technology spacecraft, as well as on sophisticated, advanced design spacecraft.	60	G6, G7, G8, J1

9.3.5 REQUIREMENTS RELATED TO PROTOCOL CAPABILITIES

The largest group of requirements relate to the capabilities and operating characteristics which the protocol must possess. Refer to table 9-4.

Table 9-4: Requirements Related to Protocol Capabilities

Group Num.	Requirement	Req. Ref. Num.	Source
cap 01	A protocol Peer shall be capable of both receiving and transmitting files simultaneously.	25	E23, G13, J1, J5
cap 02	A protocol Peer shall be capable of concurrently supporting multiple file transfer instances.	26	E23, G14, J4
cap 03	The protocol shall provide the capability to transfer both files (arrays of octets, which may or may not be further structured as arrays of CCSDS packets) and metadata (which may or may not pertain to those files).	39	I02
cap 04	A file is defined to be an array of octets (not bits).	65	J35
cap 05	The protocol shall handle variable record sizes.	40	E19
cap 06	The protocol shall allow file transfer up to (2^32)-1 octets.	42	E13
cap 07	The protocol shall allow requests for a file transfer to specify the file by name.	43	I8
cap 08	The protocol shall provide immediate access to the received data as it is received, i.e., without waiting for the file to be completed	37	E25, G17, J3
cap 09	The protocol shall provide the capability to operate in a ‘Single Transmission’ mode, in which the data are sent once and only once.	28	C10
cap 10	The protocol shall provide the capability to operate in a ‘Selective Retransmission’ mode, in which missing or corrupted sub-data units are identified by the receiving Peer to the sending Peer, and the sending Peer then retransmits those and only those sub-data units.	30	C12, G15, I10
cap 11	The protocol shall be automatic, but shall provide for manual intervention in case of anomaly.	78	E09
cap 12	The protocol shall support suspend and resume operations.	53	I13
cap 13	The receiving protocol Peer shall remove any duplicate data received.	61	G16, J36
cap 14	The protocol shall provide the capability of initiating a file transfer without transfer initiation handshaking between the Peers.	31	I7, J19
cap 15	The protocol receiving Peer shall provide the capability to,	35	C9, I6,

Group Num.	Requirement	Req. Ref. Num.	Source
	during the file transfer process, make available to the using Application the status of the available received data, including reporting that: a) data are still being received (and the available data do or do not contain errors), and b) data have been completely received (and retransmission requests are or are not pending) (and the available data do or do not contain errors).		J7, J8, J9
cap 16	The protocol receiving Peer shall provide the capability to periodically report comprehensive status back to the sending Peer.	32	J7, J8, J9
cap 17	The protocol receiving Peer shall not require acknowledgment of the comprehensive status reports to proceed if the file integrity is detected to be correct.	33	J19
cap 18	The protocol receiving Peer shall provide the capability to, upon receiving a complete and correct file, provide a final acknowledgment to the sending Peer.	36	I6, J24
cap 19	The protocol shall be capable of completion of a file transfer without transfer completion handshaking between the Peers.	38	I7, J3
cap 20	The protocol shall provide the capability to allow file transfers to span protocol Sender/protocol Receiver contacts.	62	E24, J33
cap 28	The protocol shall inform the recipient application that the file is available for use. If the file is incomplete, the temporary name being used by the protocol process shall be provided along with a completeness map.	64	J34
cap 29	The scope of the data being transferred may be multiple extents (not just a single length starting at zero), which may change over time.	72	J43
cap 30	The protocol shall provide proxy file service.	81	I15
cap 31	For operation over unreliable lower layers, a checksum for each file segment shall be optionally provided.	82	E28
cap 32	For bounded files, a checksum for the entire file shall be provided.	83	E29

9.3.6 MANAGEMENT

The requirements which delineate the record handling, file handling, file management, and directory management which the protocol must possess are listed in table 9-5.

Table 9-5: Requirements Related to Records, Files, and File Management

Group Num.	Requirement	Req. Ref. Num.	Source
rfm 01	The protocol shall assume the following set of file access primitives from the local file system: 'Open', 'Read', 'Write', 'Seek', 'Remove', and 'Close'.	44	E18, J28
rfm 02	The protocol shall provide File transfer capabilities of 'Get' (request file transfer from remote Peer to local Peer), and 'Put' (request file transfer from local Peer to remote Peer).	45	E20, G11, J32
rfm 03	The protocol shall provide the following file handling services: Load a New File, Send a File, Modify a File, and Replace an Existing File.	46	G11, J10, J32
rfm 04	The protocol shall provide the following file management services: Request a File, Rename a File, Delete a File, and Report a File Status.	47	E21, G11, J11, J32
rfm 05	The protocol shall provide the following file directory management services: Create directory, List directory, Rename directory, Delete directory, Change to directory, and Report current directory.	48	E22, G11, J29, J32
rfm 06	The protocol file transfer services shall be independent of local filing systems.	63	E26

9.4 IMPLEMENTATION REQUIREMENTS

The requirements on the implementation of the File Delivery Protocol are shown in table 9-6.

Table 9-6: Implementation Requirements

Group Num.	Requirement	Req. Ref. Num.	Source
imp 01	The protocol shall minimize the load on onboard computing resources.	58	C6, G8
imp 02	The protocol shall minimize the use of onboard memory resources.	59	C7, E1, G8
imp 03	The protocol specification shall be fully validated and tested.	56	J13
imp 04	The protocol sending Peer shall have the option of responding to the final acknowledgment of receipt by deleting the file that is known to have been correctly transmitted.	51	J25

ANNEX A

ACRONYMS AND ABBREVIATIONS

ACK	Positive Acknowledgment
APL	Applied Physics Laboratory (at Johns Hopkins University)
BEOP	Burst Error Occurrence Probability
BNSC	British National Space Centre
CCSDS	Consultative Committee for Space Data Systems
C&DH	Command and Data Handling
CFDP	CCSDS File Delivery Protocol
CNES	Centre National d'Etudes Spatiales
CPSC	CDFP Packet Service Component
DERA	Defence Evaluation and Research Agency
EOF	End of File
ESOC	European Space Operations Centre
ESTEC	European Space Research and Technology Centre
FD(n)	File Data Segment
FIN	Finished (receiver to sender)
FDU	File Delivery Unit
FIFO	First-In-First-Out
FT	File Transfer
G&C	Guidance and Control
GEO	Geosynchronous Earth Orbit
GTO	Geosynchronous Transfer Orbit
GUI	Graphical User Interface
IDE	Integrated Development Environment

ITU	International Telecommunication Union
JHU	Johns Hopkins University
LEO	Low Earth Orbit
M	Metadata
MCC	Mission Control Center
MIB	Management Information Base
MSB	Most Significant Bit
NAK	Negative Acknowledgment
NCC	Network Control Center
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PRMPT	Prompt
RTM	Relay Testing Module
SAD	Software Architectural Design
SDL	Specification and Description Language
TBS	To Be Supplied
TC	Telecommand
TCP	Transmission Control Protocol
TM	Telemetry
UDP	User Datagram Protocol
UT	Unitdata Transfer
VCL	Visual Component Library
XN	Transaction