# LOCAL AREA SYSTEM TRANSPORT ARCHITECTURE

A transport layer model for establishing associations and exchanging data is described. The model purposefully constrains the services offered by the ISO transport layer to the ISO session layer in order increase performance and reduce resource consumption. The ISO session layer users of this transport service are envisioned to be the system processes and not applications programs.

**Bruce Eric Mann**

**Engineering Products**

**Strategy and Architecture**

| | |
|---|---|
| **Revision** | 1.10 |
| **Written by** | Bruce Eric Mann, Christian Saether, Philip Wells |
| **Date** | 29–February–1988 |
| **File** | LAST.SDML |
| **Issued by** | Personal Computer Systems Group, Littleton, MA. |

# Preface

# Edit History

| Rev | Description | Author | Date |
|---|---|---|---|
| 0.1 | Added part of LAST spec | Mann | 14−Mar−85 |
| 1.0 | Preparation for PCSG | Mann, Gamache, Saether | 06−May−86 |
| 1.1 | Modification of RUN message | Wells | 01−Oct−86 |
| 1.2 | Rewrote part of introduction | Mann | 08−Dec−86 |
| 1.3 | Add message formats, complete state tables | Wells | 10−Mar−87 |
| 1.4 | General edits, added sections on system structure, message timing, updated state tables and message formats. | Mann, Wells | 20−Mar−87 |
| 1.5 | Preparation for DRG review in sufficient detail to assign a protocol type and multicast address range | Mann, Wells | 17−APR−1987 |
| 1.6 | Add Saethers review comments | Mann, Wells | 15−MAY−1987 |
| 1.7 | Add OS/2 to registered product codes. | Wells | 15−SEP−1987 |
| 1.8 | Add Group code description. Detail congestion control algorithm | Wells | 24−NOV−1987 |
| 1.9 | Convert to DOCUMENT | Wells | 01−DEC−1987 |
| 1.10 | Add Orphan Transaction detection description | Wells | 29−FEB−1988 |

# Notation

Capitalized named of the form "Transaction_timer" are architecturally defined names. They represent values, events and functions. Capitalized names of the form RESPONSE_CONNECT and MSG_FLAGS are message names or names of fields contained within messages.

:MESSAGE_LENGTH indicates the value contained within the field MESSAGE_LENGTH.

START.DST_CIR_ID indicates the destination circuit identification field within the START message.

The symbol VCH indicates the virtial circuit header present in all LAST message types.

All values in this document are to be interpreted as base 10 unless otherwise noted.

# A Transport Terminology

- datalink − A uniquely addressable network entity corresponding the the ISO Datalink and Physical Link layer.
- datagram − an atomic unit of information exchanged between nodes in the system. Datagrams are required to have a constant format consisting of: destination node address, source node address, protocol discriminator, data and an error detection code. Datagrams may get corrupted, and are therefore not always delivered to the destination node. Datagrams exist in the ISO Physical and Datalink Layers (1 and 2).
- Master/Slave (circuit entities) − addressable processes providing attachment points (ports) for detecting node incarnations. The Master initiates activity and the slave responds in the ISO network and transport layers (3 and 4). Within the LAST architecture, the network layer is essentially null; it is assumed that the network layer function is provided by switches (bridges) if necessary.
- client/server (association entities) − addressable processes providing service access points for associations. The client initiates activity and the server responds in the ISO session layer (5).
- adapter − A data link address and service access point associated with datagram. On multiaccess LANs, multiple adapters have unique addresses.
- message − a datagram which is formatted as specified by the LAST architecture.
- transaction − a request/response sequence.
- association − an binding which allows a client to send datagram or execute transactions against a specific service.
- segment − A circuit message that forms part of a transaction.
- Service binding − a reference to association management that is independent of server location and server in−stance. The client names a service, the transport selects an appropriate server, and an association is established with the service selected by the client.
- congestion control − a set of rules applied to processes which prevents a transmitting process from sending data to a receiving process that is not prepared to buffer the transmitted data.
- multicast − as applied to data links, multicast capability refers to the ability of any one port to address a sub−set of all other ports nearly simultaneously with a single datagram.
- bridge − a sub−system which allows multiple LAN segments to be interconnected and operate logically as a single segment.

# TABLE OF CONTENTS

## TABLES

# CHAPTER 1

# INTRODUCTION

This document is structured so that the general material is presented first, and the more detailed material later in the document. Thus, a reader interested in a concepts overview can read the first few sections.

This document describes an ISO level 4 transport protocol, interfaces, and the environments in which this architecture is anticipated to be useful. The purpose of the transport layer is to optimize the use of the underlying layers and to provide service to layer 5, the ISO session layer. In this way, the ISO session layer (and associated higher level applications) do not have to be modified to conform to the changing requirements of the dynamically evolving lower layers.

The Local Area System Transport (LAST) protocol is somewhat unusual in that the service and service guarantees provided by the transport layer are a subset of those offered by more traditional transports. For instance, message ordering is not preserved, the data service is inherently asymmetric and the service is transaction oriented (command/response) instead of stream oriented.

Perhaps the most unusual requirement on the ISO session layer is that service calls need to be structured as idempotent transactions to take full advantage of the architecture. Or stated differently, transactions must be repeatable with each repetition yielding an equivalent result.

## 1.1 THE ENVIRONMENT

Networks are information sharing systems. Information consistency is hard to maintain in a distributed system. For this reason, databases which maintain a high degree of information consistency tend to centralize the database and distribute client access to the database.

Such systems often have many thousands of database clients active simultaneously and must service hundreds of commands each second. Large systems have more than 50,000 simultaneous clients and must service more than one thousand transactions per second.

Not all clients are active at once. However, end−to−end arguments lead to the conclusion that the database system must maintain some information about each potential client. Multiplexing many clients commands (either in a single message or over a single virtual circuit) can reduce the number of clients seen by the database system, but this technique reduces fault tolerance by introducing new single points of failure, increases end−to−end latency due to the multiplexing/demultiplexing function, and introduces unanticipated and unpredictable dependencies between otherwise independent clients.

The LAST environment is modeled as a set of associations between clients and servers. Association is a term which refers to the shared state which exists between the client and server. Clients and servers multithread operations (asynchronous operation). It is assumed that there are potentially a large number of clients compared to servers (service providers). Therefore, client resources are expended willing to conserve server resources.

The transport service is expected to operate over multiple low−delay, low−loss, high−throughput interconnects (Local Area Networks), even though particular aspects of the transport design (selective retransmission and level of concurrency supported) may make LAST implementations less sensitive to delay and loss than virtual circuit based transports.

## 1.2 ENVIRONMENTAL CONSTRAINTS

LAST transport protocol assumes the underlying shared interconnect has very predictable attributes. The transport performance may depend on "low probability" events occurring infrequently. The transport correctness may depend on "very low probability" events not occurring at all. If "low probability" means less than one event in one thousand (10**3), and "very low probability" means less than one event in one trillion (10**12), then the LAST transport protocol assumes the underlying datalink/network layer has the following attributes:

- a low probability of datagram duplication
- a low probability of datagram corruption (and therefore non−delivery)
- a low probability of datagram being delayed more than 50 milliseconds between source and destination ports
- a low probability of datagram being delayed more than 10 seconds between source and destination ports
- a very low probability of datagram being delivered to the wrong destination address (adapter)
- a very low probability of datagram being delivered which contain undetected corrupted data

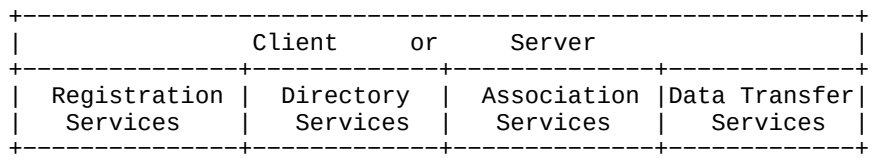
Some other important assumptions:

- an interconnect bandwidth significantly greater than the bandwidth consumed by all clients averaged over 10 seconds.
- a broadcast/multicast capability
- a syntax free name space (no hierarchical names for instance)
- Any security issues are outside the scope of the LAST architecture.

Notice that in a LAST environment which sends 1,000 messages/second, that data might be corrupted every few months and not be detected. If higher levels of data integrity are required, upper level mechanisms must be used to augment the data link integrity mechanisms.

## 1.3 SYSTEM COMMUNICATION MODEL

Transport presentation service:



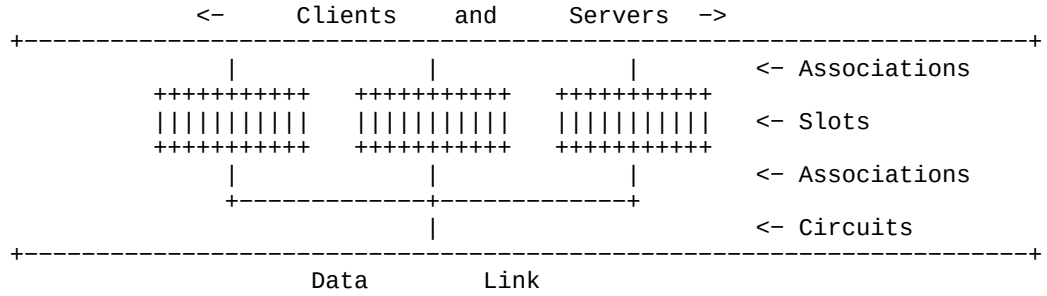

Registration services are not architected, but are an implementation issue. This initilization service must make the Directory, Association and Data Transfer services available to the session layer users.

Directory services may be decoupled from the transport, but are presented as an integral part of the transport architecture.

Association services bind single client instances to server instances.

Data Transfer services allow Clients to consume Server provided services using established associations.

Transport internal structure:



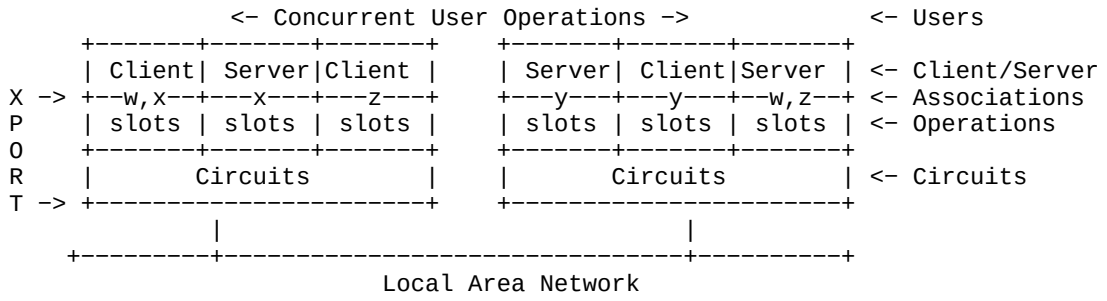

Slots are bound to associations. Associations are bound to circuits.

Slots are a transport mechanism which support multiple concurrent transactions and are not visible to users. Each concurrent operation is bound to a single slot. Two operations may not be bound to a single slot simultaneously.

Associations convey client/server bindings and are presented to users as associations. Associations are primarily an addressing mechanism.

Circuits convey operating system or node instances. One circuit will exist for each node to node association. Circuits also maintain the underlying topology.

A system run−time overview:




Client users are presented a system specific service interface. Clients originate transactions. Server users are normally implemented as system level applications, and are presented transactions initiated by the associated client instance.

Operations can be handled concurrently by the transport layer for each separate instance of an association.

There are four associations represented: w−w, x−x, y−y, and z−z. Notice that each client and server can concurrently handle multiple associations.

## 1.4 GOALS

### 1.4.1 ARCHITECTURAL GOALS

The goal of this document is to specify the LAST protocol syntax and semantics in sufficient detail to allow interoperable implementations to be based on this document, and subsequently, to certify implementations conform to this specification.

It is a goal of LAST architecture to supply the Session layer client the following three services:

- A combined directory/association management/naming service
- A datagram service (not completely specified in this document)
- A command/response data transfer service (called a transaction)

Although naming services need not be an integral part of the LAST architecture, a description of a naming service is included in this document which makes use of an assumed underlying LAN based multicast service. This naming service does not require participation by systems other than those on which the client and server services reside. The availability of a third−party general naming service would also be sufficient, although resource availability might be impacted by the dependency on the third party naming service, and association establishment could not be integrated with the naming service as is described in this architecture.

#### 1.4.1.1 CONSTRAINTS

The transport layer offers the client a transaction service. This transaction service offers a reduced set of service guarantees to clients compared to traditional transport architectures. Specifically, the constraints and the rationale for each constraint are:

- Asymmetric service − The client can initiate requests, the server cannot. If symmetry is desired, two independent associations must be established.

  This simplifies connection management, buffer management and flow control.

- Error detection/Limited error correction − The transport may indicate that the transaction failed. This is not considered to be an unusual event. The decision to attempt recovery is deferred to the client−server pair.

  This allows an architecture to be implemented recursively in the system. An intelligent adapter might attempt a transaction and fail, reporting this failure to the transport driver. The driver might retry the transaction (possibly using different adapters and interconnects), fail, and report this failure to the client. The client might know of a different instance of the transport with independent resources, and retry the very same operation on the alternate transport instance. This sort of structure is a classical fault−tolerant design.

- Granularity − The client must describe the entire operation at the time the request is made to the transport. Both the request buffer and the response buffer must be defined.

  If a client implementation makes the same requirement of the user application program, then large transactions might be segmented into a number of smaller transactions by a system agent acting on the behalf of the application with the knowledge that segmentation boundaries are naturally aligned with the application message boundaries.

  For instance, a request to read 1000 blocks starting at block 0, can be decomposed into 10 individual transactions, each of which reads 100 blocks. These 10 sub−transactions could be initiated over 10 independent paths. Each network message might be formatted directly from segmented user buffers without data copying operations. Thus, over any individual path, the execution of transactions can be tailored to optimize the performance of that individual path.

  Transaction recursion allows independent execution and recovery of each individual sub−transaction.

- Transactions are commutative − Transaction request sequencing, transaction execution sequencing and transaction response sequencing are the responsibility of the application user. This constraint allows the transport to provide maximum independence between transactions. Since transactions can be reordered, the transport layer can achieve more concurrency and can complete transactions across the session interface in the order that they actually complete. No state needs to be maintained to preserve transaction ordering.

#### 1.4.1.2 FEATURES

LAST offers the client a transaction oriented service, reduced memory consumption, and reduced CPU utilization.

At the client and server service interfaces, the transaction commands and responses are paired to reduce the number of times this interface is crossed. In most virtual circuit transport architectures, commands and responses are not matched by the transport interface. This means that the client interface issues a command as one service call, and receives a response via a separate call. The command/response matching is up to the client.

In contrast, a LAST transaction allows this command/response matching to be done by the transport. This is a more efficient mechanism, and can provide more performance and integrity if the client is an application program.

Internally, the transport service takes advantage of the transaction mechanism to reduce message buffer occupancy time and reduce the number of messages exchanged relative to virtual circuit oriented transport architectures.

The LAST architecture anticipates that the shared interconnects offers limited bandwidth and that adapters attached to the shared interconnect may vary in reliability. Therefore, the ISO Session layer may assume the ISO transport layer manages the optimization of the underlying interconnect by:

- Allowing command and response user messages of arbitrary lengths.
- Managing the limited bandwidth of the shared interconnect so that the bandwidth may be shared equitably among the multiple session layer clients by specifying a single rate−based congestion control algorithm for the LAST architecture which responsively allocates the available interconnect bandwidth to the active client−server pairs when few are active, and prevents interconnect saturation when many client−server pairs are simultaneously active.
- Allowing the client to specify the rate at which error correction is to be attempted (if at all) on an individual command basis.
- Supporting multiple adapters to each interconnect and multiple interconnects to achieve higher bandwidth, reduced latency and higher availability. The underlying topology of adapters and interconnects is transparent to the session layer clients in the sense that adapter and interconnect failures only affect quality of service, not the continuity or correctness of service. If any path between a client and server is operable, the transport layer will dynamically load balance across available paths.

### 1.4.2 IMPLEMENTATION GOALS

- LAST is structured for system−level implementation of clients and servers − operating systems normally constrain clients of transport implementations to operate as applications. This often exacts an unacceptable performance penalty. Just as transport level implementations operate more efficiently if they are implemented at the system level, so do clients of the transport layer.

  One significant problem is the time it takes to create a process which handle association context. Application process creation time is measured in seconds, while system−level process creation time is measured in mi−croseconds. While process creation time is not a significant problem for some applications, it is a significant problem when trying to implement "system" level capabilities such as disk, file, timing, queuing, event logging and naming services.

  A second problem is the latency and CPU overhead associated with unnecessarily crossing system levels. Transport servers (and less often clients) should be structured as system modules if that is appropriate to avoid this unnecessary overhead.

- LAST reduces CPU utilization in the server − Due to the general purpose nature of standard network archi−tectures, overhead is present in implementations which is often superfluous to the application. On paper, one transport protocol cannot realistically be demonstrated more efficient than the next. In practice, newer and more specialized implementations tend to be more CPU efficient.

  LAN based systems focus on the client−server relationship. CPU utilization in the server is already a key factor is system design, partly because it is easier to add interconnect bandwidth to a system than it is to add MIPS to a system.

- Checksumming − Because data link and data link adapter implementations vary in their ability to maintain data integrity, the LAST architecture describes a method for optional checksumming of messages.

  In the presence of faulty data links or data link which do not offer a adequate levels of error detection, LAST can offer an efficient additional level of error detection transparently to the application level.

In summary:

The goals of the LAST architecture are:

- Reduced message flow over the interconnect
- Reduced CPU consumption in the server system
- Reduced memory consumption in the server system
- Increased granularity of service requests
- Transaction based error control.
- More flexible distribution of error detection/correction within the system
- Minimum possible command/response latency
- Efficient usage of multiple independent data link paths
- Simplified implementation
- More robust implementation
- To allow implementations of system level transport clients and servers

# CHAPTER 2

# THE IMPORTANCE OF IDEMPOTENCY

Idem− means same; −potent means strength. Idempotent operations are operations which can be repeated without changing the system state in the sense that the repeated operation has the "same strength" as the original operation, assuming it is repeated in isolation from other operations.

Typically idempotent operations are replacement operations. For instance, if A=0, the operation A<−1 is idempotent, while A<−A+1 is not.

In a distributed environment, a communication system can be optimized if a remote operation is known to be idempotent. If no response is received from an application after a command, the command can simply be repeated until a response is received, with the knowledge that the remote system state is consistent even if the remote operation is repeated.

Sometimes, distributed operations which are not idempotent can be made to be idempotent. For instance, if the value of A is to be replaced by the value A+2, this can be done in two ways. The issued command can be to add 2 to the value of A. If no response is received and the original value of A is unknown, recovery from this failure is impossible (without other information) − it cannot be determined if the current value of A reflects the original value or an updated value. In contrast, the value of A can be read (idempotently), the read value can be updated by 2 and the result can be written again idempotently (this scenario assumes that the client does not fail).

In general, systems rely on idempotent operations to achieve fault tolerance. A name translation is a good example of idempotency at work. If the remote application (name server) or the communication system fails after the command has been issued, but before the response has been received, the system simply repeats the last operation. Since the naming semantics are idempotent, it is harmless to repeat the operation that was in progress when the remote system failed, since the newer response will be the same as the old response, or an equivalent, more meaningful response. The important point here is that the client system is in a known state when the command is retried, even though the exact state of the remote system is not know.

A few other common service which are idempotent (or may be structured to be idempotent) are:

- Any read−only service
- Management services
- Storage services
- Atomic queue services
- Naming services
- Timing services

Providing service exactly−once in a distributed environment in the presence of server application failures is closely related to idempotency.

If a remote operation is idempotent, exactly−once service is indistiguishable from at−least−once service (axiomati−cally). It is achieved trivially by repeating the transaction until it succeeds. The client state is sufficient to guarantee the operation completion. This scenario assumes the client and server are executing in isolation.

If the remote operation is not idempotent (perhaps the operation is to output data to a check printer) the exactly−once service requirement is difficult to achieve in the presence of server failure, but impossible in the presence of synchronized client and server failure.

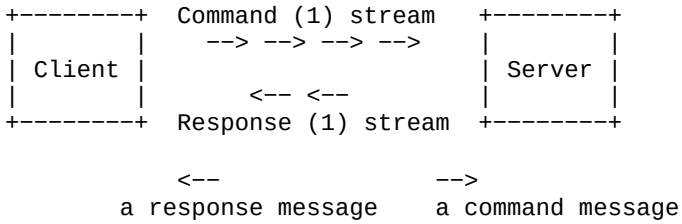
The at−least−once case of a non−idempotent transaction simply requires that transactions be repeated by the client until any one transaction succeeds. For example, writing listings to a line printer. Errors may cause multiple partial listing to be printed.

The exactly−once case of a non−idempotent transaction requires that the Transaction_transmit_limit be set to the value one. Used this way, the transaction will complete exactly once, or not at all. If the client is informed that the transaction aborted, application specific code must be used to recover. This recovery algorithm might involve server utilization of client supplied unique transaction identifiers which span transaction and transport failures.

# CHAPTER 3

# LAST OPERATION OVERVIEW

A simple pictorial view of the LAST protocol in operation:




The implications of this figure are:

- The command and response are association level interface requests which are translated into the transport command stream and the response steam.
- The Server must always reply to commands at the association interface.
- Segmentation and reassembly are handled transparently by the transport layer (commands and responses are arbitrary sizes).
- Commands and responses are matched by the association and transport layers (shown as transaction number 1 in the figure).

Any transport protocol must deal with two types of information: Its own control information, and user data supplied by the higher layers. The control information is used at the transport level to achieve transport layer goals.

The value added service offered by the transport layer is most clearly revealed by the meaning of the "transmit acknowledgement" received at the transport service interface.

There are three semantic steps possible with transport transmit acknowledgements:

- datagram service − the transmit acknowledgement means "message launched" (this is also called connectionless)
- acknowledged datagram service − the transmit acknowledgement means "message has reached the remote transport protocol module" (this is the level of acknowledgement provided by nearly all transport implemen−tations)
- remote application called − the transmit acknowledgement means "command has been received by the remote application, and this acknowledgement contains the data in reply to your command"

Internally, LAST uses all three forms of transmit acknowledgement. Clients transmit commands and receive either a command response or an application level acknowledgement, or both. Servers transmit response messages as datagram.

## 3.1 INTEGRITY SEMANTICS

It can be observed that the vast majority of transport level communications are transaction oriented (com− mand/response). That necessarily involves two messages. If no communication failures occur and the client and server do not fail, the acknowledgements to commands and responses are redundant. LAST Idempotent_mode eliminates the need for these acknowledgement messages (see section on message timing considerations).

To help understand the operation of LAST, it is helpful to review the operation of traditional transport architectures when they execute transactions. Most transport protocols operate more or less the same way:

1. An application level command message is sent.
2. An transport acknowledgement message is returned (occasionally piggybacked on a reverse path message, if one is available).
3. An application response message is sent as a reply to the command.
4. A transport acknowledgment message is sent to the response (possibly piggybacked).

An additional indirect benefit of getting rid of acknowledgment messages is the elimination of the requirement to buffer the application messages in the transport layer until the transport acknowledgment arrives (since it never will).

To accomplish these goals, it is necessary to change some of the transport interface service semantics offered to the ISO session layer. Specifically:

- Transaction orientation − Each transaction is composed of two application messages: a command message and a response message. In the normal case, the server application will always generate a response message.
- Asymmetric congestion control − The transmission of client commands and server responses are controlled via different policies.

By careful specification of the transport layer transaction semantics to what is generally described above, the potential client generated response acknowledgement message can be eliminated and the server buffering time is reduced.

In summary, the potential client response acknowledgment is not necessary because:

- the client will persist with new transaction commands if response messages are not received until a response message is receive, or until the transaction and underlying association is aborted.
- responses do not consume credits which need to be returned by a response acknowledgement message.

At certain boundary conditions, one−third of all messages can be eliminated, and in theory, only a single transmit buffer is required at the server node to service all clients (although in practice, sufficient buffering must be made available to make allowances for queuing delays as data flows across the application/system/interconnect boundaries and possibly is copied through layers).

Because transactions can be idempotent, no state is stored in an association which is necessary to error recovery attempts by a client surviving association failures. The client application may attempt recovery using a new association generated to alternate instances of the server application by using transaction identifiers that span association instances.

In contrast with transports which are connection oriented and virtual circuit based, transaction sequencing is an application level responsibility. Thus any sequencing service provided by the transport layer is a performance goal and not a correctness goal.

### 3.1.1 ORPHAN TRANSACTION DETECTION

An "orphan transaction" is created when a server sees a retransmitted request before the original request, and processes them the opposite order in which they were requested. Consider the following example:

| Client | Client Agent | Server |
|---|---|---|
| a⇐−4 | Slot 1 Seq 1 (a⇐−4) | Sees Nothing |
| | Timeout | |
| | Slot 1 Seq 2 (a⇐−4) | Slot 1 Seq 2 (a⇐−4) |
| a⇐−5 | Slot 1 Seq 3 (a⇐−5) | Slot 1 seq 3 ( a⇐−5 ) |
| | | Slot 1 Seq 1 ( a⇐−4 ) |

In this example, the client will assume a=5, but be surprised to read the value of a=4. The problem demonstated here is that the original request may be executed by the server after the client assumes all processing is complete in the server. This situation here is described as an "orphan transaction", and is peculiar to non−virtual circuit based protocols.

LAST defines the following algorithm for detecting orphans:

- All transactions are multiplexed through a slot/sequence matrix. This architecture defines 255 slots numbered 1 through 255. Transactions are then assigned a slot index when the request is created. All retries of an operation are assigned to the same slot. Within the slot index, a sequence identifier is assigned. These sequence identifiers are numbered 0 − 255. When a request is generated a slot and sequence number is assigned and the request is transmitted. If the request is timed−out before a response is received, a new request is created using the same slot and the sequence number is incremented by 1. Clients will ignore any response on a given slot unless the response sequence identifier exactly matches the request sequence identifier.
- A server records the sequence number (N) in each slot of the last successfully executed transaction. When a new request is received, the server will verify that the request sequence number is within the range N+1 to N+128.

So, in the example given, the server will ignore the "orphan transaction" because the slot sequence number is out of range.

| Client | Client Agent | Server |
|---|---|---|
| a⇐−4 | slot 1 seq 1 (a ⇐−4)slot 1 seq 1 ( a⇐−4 ) | |
| | Timeout | |
| | Slot 1 Seq 2 (a⇐−4) | Slot 1 Seq 2 ( a⇐−4 ) |

The other interesting case is show above. Even if the client receives a successful response to slot 1 seq 1, it will ignore that response waiting until it sees the response to the request corresponding to slot 1 seq 2. FThis will ensure that after the client receives successful complete, no further server activity can take place which is associated with that request.

## 3.2 TRANSACTION MESSAGE SEGMENT STRUCTURE AND OPERATION

The clients make a transaction identifier and the addresses of command and response buffers available to the transport at the time the transaction request is issued. LAST takes advantage of this requirement and generates a command/response identifier which pairs client commands with server responses.

Each command and response segment carries this identifier and segment identifiers which unambiguously determine the segment's ordering with a command or response stream. Thus each segment is completely self−describing within a stream and distinguishable from all segments in other streams (past, current, and future).

Because of this, the message transmitters (at either end of the transport) can round−robin the packets across all available data links without concern as to the order in which they are delivered. The transport receiver simply moves message segment data to destination buffers in whatever order it arrived. Also less memory is consumed since transaction granularity allows the receiver to process messages in arbitrary arrival order. The ultimate destination of message data is known to the transport layer because the buffer addresses were passed by reference to the transport layer at transaction initiation.

At the server end, sufficient transport receive buffers must be available to cache out of order commands segments until the first segment is received and can be delivered to the server application. Subsequently, packets can be copied directly into the server supplied receive buffers in any order. Notice that after looking at the first segment, the server user knows the size of the transaction command stream (assuming the application protocol fits in the first segment).

If a server application aborts a received transaction command, it notifies the transport layer so it can ignore any subsequent segments received which are associated with the aborted command stream. In this case, recovery is a client responsibility.

## 3.3 TRANSACTION MESSAGE TIMING OVERVIEW

Transaction modes (Idempotent_mode, Timed_mode and Normal_mode) utilize timers to perform error detection and correction. Non−transaction modes (Datagram_mode) is not affected by, and does not use any architected timers.

The material presented in this section is presented in detail in the state diagrams later in the document.

Both the client and server use a Transaction_timer to perform error control during the execution of a transaction. This Transaction_timer is associated with three different values: the Command_response_timer, the Transaction_ response_timer and the Round_trip_message_delay.

There are three message types involved in a transaction: The command message, the resync message, and the response message. The command and response messages might actually be segmented message streams, but architecturally are treated as two separate atomic message streams. The resync message is always a single message segment, and is sent by the server to the client.

```
Command stream
–> –> –> –>
   Resync
   <–
Response stream
  <– <– <–
```

The client specifies two timer values when a transaction is initiated. Since the command message may be lost and since actual application processing delay can never be known in advance, the client must establishes both:

- a Command_response_timer, which is an initial time limit on receipt of a command message acknowledgement or transaction response
- a Transaction_response_timer, which is subsequently used as a time limit on completion of the transaction, but can be reset by the transport server using resync messages.

The client sets the Transaction_timer to the Command_response_timer value after the transmit completion notifi− cation is delivered for the final message of the command segment stream.

For example, assume the client specifies a Command_response_timer value of 3 seconds and a Transaction_ response_timer value of 30 seconds. After the transaction stream is launched, the Transaction_timer is set to the Command_response_timer value of 3 seconds.

Whenever the client's Transaction_timer expires, it always resets the transaction timer to the value of the Com− mand_response_timer, declares a Command_failure event and reissues the transaction command (3 seconds in this example). The client attempts a total of Transaction_transmit_limit command attempts, at which point the trans− action is suspended. The transaction might be aborted, or the decision to abort can be a function the circuit layer events. A Transaction_failure event is declared to the circuit sublayer, and in response, the circuit sublayer will declare Circuit_up, Circuit_fault or Circuit_down. The Circuit_up event will cause the transaction to be retried. The Circuit_down event will cause all underlying associations (and all multiplexed transactions) to be asynchronously aborted. A Circuit_fault may cause the transaction to abort or to remain suspended.

Whenever the client receives a resync message from the server, the Transaction_timer value is reset to the Trans− action_response_timer value (30 seconds in this example). This prevents lengthy server application computation from being aborted by the client retrying the transaction with redundant command messages.

The server uses a value called the Round_trip_message_delay which is twice the estimated message transit time over the intervening data link. This timer is the sum of the estimated transit delay for the last segment of the command message and the estimated transit delay for the resync message. In this example, a value of 1 second might be appropriate.

When the server receives the final segment of a command message stream, it sets its Transaction_timer to a value which is equal to the Command_response_timer minus the Round_trip_message_delay, or a value of 2 seconds.

Should the server Transaction_timer expire, the server transmits a resync message and resets its Transaction_timer to the Transaction_response_timer (as specified in the command message) minus the Round_trip_message_delay, or 29 seconds for this example.

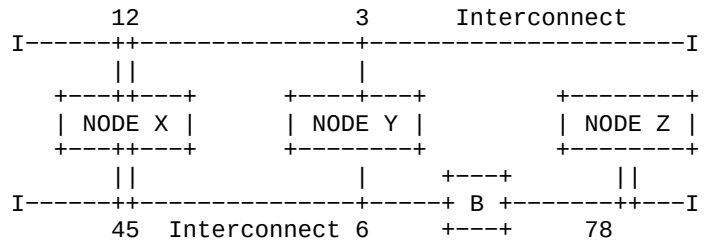
If the server association user completes the transaction, the response message stream is transmitted and the trans−action is complete.

The transaction is also completed by aborting it. Transaction are aborted because:

- Server
  - — A new transaction command arrives for the same slot
  - — No resources are available
  - — The server user aborts the association
  - — Circuit_down
- Client
  - — Retry limit reached
  - — The client user aborts the association
  - — Circuit_down or Circuit_fault

## 3.4 TOPOLOGY OVERVIEW

An example of a general case, complex topology might be:




The numbers 1−8 are interconnect physical addresses instances in the ISO data link layer. The letters X, Y, and Z represent ISO layer 4 transport instances. Normally, each node address corresponds to a single system, but a system might implement multiple node service access points.

Notice that all paths are logically point to point. Any environment which requires the ISO layer 3 network functions must utilize bridges (B) so that all paths remain logically point to point.

A path physically consists of two controllers and one interconnect segment. Thus in the diagram, node X is connected to node Y via four paths: 1−3, 2−3, 4−6 and 5−6.

LAST messages represent: entire transaction commands or responses, segments of command and responses, data−gram, control messages, and/or naming service messages. The LAST transport layer achieves the follow three goals relative to messages:

- Increase bandwidth between any single node and the interconnect by concurrently utilizing all controllers which attach to node to transmit messages.
- Decrease latency of individual operations by splitting transmitting messages across all available paths.
- Increase availability by allowing messages to flow over:
    - Multiple controllers between a node and an interconnect to transparently recover from a controller failure.
    - Multiple redundant interconnects (nodes X and Y above) to transparently recover from a failed interconnect.

This is accomplished in each node by associating each circuit with all possible paths over which a message can be successfully transmitted to the destination node. This means that a local node may receive messages from a remote node that have different data link source addresses. For instance, messages transmitted from node X to node Y may be received from interconnect source address 3 or address 6.

In the ISO model, the support for the topology represented above might properly be viewed as a network layer function, but due to the simplifying assumption made, it is represented here as a function in the transport layer.

# CHAPTER 4

# LAST FUNCTIONAL MODEL

In general, LAST concatenates control information with two different application layer messages types: The command message, generated by the client, and the response message, generated by the server in reaction to the command message.

LAST supports datagrams which are used to direct messages between LAST clients and servers. Although this document describes a transactions model, services provided with Datagram_mode do not offer command/response matching service. Note that the server can originate "transaction response" datagram if it uses Datagram_mode even if no command message was received.

The transport layer is divided into two sub−layers:

- the association sub−layer − this layer multiplexes session layer client transactions over an underlying circuit sub−layer and manages the association interface to clients of the transport layer. This sub−layer pairs command message with response messages at the association interface and performs any necessary segmentation and reassembly of the command and response messages into command and response message streams. This sub−layer also detects and corrects duplicate or lost commands and responses message (error detection and optional correction). Finally, this layer signals the circuit sublayer when transaction faults are detected.
- the circuit sub−layer − this layer detects new instances of nodes (node crash and reboot) or node paths and forces the higher layers to resynchronize when this occurs. It load balances message segments across all possible paths between the source and destination nodes. This sub−layer does not do any error correction, message sequencing or duplicate detection.

## 4.1 LAYER INTERFACES

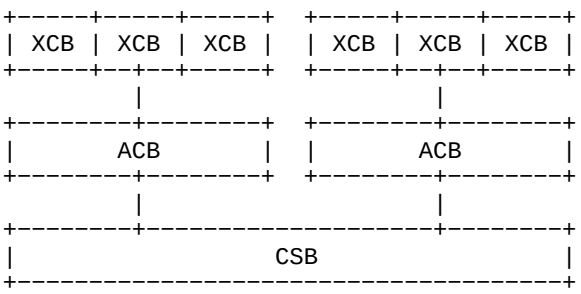
The interfaces presented in the following sections are not meant to be implemented directly. The purpose is to present the control and data flowing through the LAST layers in a way that unambiguously describes what is required at the interfaces, but allows implementations the freedom necessary to implement the functions appropriately for each different system.

The model presented implies that each side of the interface both provides service entry points and utilizes service entry points provided to it. Synchronization across interfaces is the responsibility of the implementation. Specifically, the implementation must assure that all interface events are processed serially and atomically, in both directions.

The polling model is used to show correspondence of functions and to reduce the amount of text necessary to describe the interfaces. Only one interface is described at each layer. In fact, implementations might offer only a subset of the functions described, one subset corresponding to the master and another corresponding to the slave.

## 4.2 RELATIONSHIPS OF ARCHITECTED DATA STRUCTURES

Circuit control blocks (CSBs) describe the distributed state in the circuit sublayer when at least one association (ACB) is active. Transaction control blocks are bound to association control blocks:




While the CSB reflect a symmetric relationship between nodes, the ACBs and XCBs reflect asymmetric relationships. CSBs launch messages generated by the association sublayer, but the polarity of the message is only known to the association sublayer, not the circuit sublayer.

# CHAPTER 5

# TRANSPORT ASSOCIATION SUB−LAYER

The association sub−layer provides pair−wise association management and data transfer capabilities between clients and servers. The clients of the transport are offered a session oriented data transfer service which models client service as command−response transactions. More than one type of transaction semantic is offered:

- Idempotent_mode − Transactions can succeed or fail. Client is responsible for error recovery. This is the most efficient mode if datalink is error−free. The slave end of the transport recycles transmitted responses after transmits complete. Buffer occupancy time in the slave is kept to the absolute minimum.
- Timed_mode − (not fully described in this document) Transactions can succeed of fail the transport layer will attempt recovery by commanding the slave to retain responses for a limited amount of time. Most efficient mode if data link is error−prone. The slave recycles transmitted responses after a timer expires.
- Normal_mode − (not fully described in this document) Transactions always succeed (or one of the node pair crashes). Operates as a half−duplex network service. This is the least efficient mode, but must be used if the client layer cannot recover from failed transactions. The server recycles transmitted responses after they are acknowledged by the client.
- Datagram_mode − In Datagram_mode no transaction matching is provided. Message streams are launched and delivered to the bound association entity. Error detection and error recovery is an association user responsibility.

  This is in contrast to the first three modes, which are transaction oriented in that transactions are a matched command−response pairs. Matching of commands and responses is the responsibility of the transport associ−ation sublayer. Error detection is performed by the association sublayer, while error correction may be done at the discretion of the association user and is dependent on the transaction type.

In Idempotent_mode and Timed_mode, datalink error detection and recovery is performed by the association sub−layer for each transaction independently. This provides four significant advantages when compared to error detection by a common lower layer:

- it allows the different transaction semantics described above to operate over the same circuit without conflict.
- Idempotent_mode offers a major performance advantage over Normal_mode (with reasonably error−free datalinks) by eliminating the need to acknowledge responses. Consider a simple command−response exchange where the command and response messages each consume a single message packet, and no errors occur. In Idempotent_mode, the entire exchange is completed using a total of two packets. In Normal_mode, an additional packet is required to acknowledge the response, using a total of three packets as opposed to two, a non−trivial increase. If datalink error detection were performed by the circuit sub−layer, it would be unable to distinguish one transaction from another, and therefore be required to always use Normal_mode.
- In topologies that involve multiple paths, any path can be selected since error recovery is being done by a higher layer.
- Transaction based error detection and recovery allows recovery of a specific transaction to proceed independently of all other transactions. Virtual circuits stall all pending transactions while recovering from errors. In many applications, both multiple associations per circuit and multiple transactions per association may not be uncommon, and hence transaction based error handling will offer the best performance under those loading conditions when errors do occur.

The association sub−layer is modeled in sections:

- interfaces modeled as functions
- association state variables
- definition of events
- validation of events
- operation of a state diagram (events and actions)
- action routines

## 5.1 ASSOCIATION USER INTERFACE

Each association user will register with the transport by specifying a service class identifier. This service class identifier specifies the type of services being offered or requested. The transport will provide that only 1 server/client per service class is registered. Servers and clients that attempt to register a service class that is currently registered in the mode declared (client or server) will be refused access to the transport.

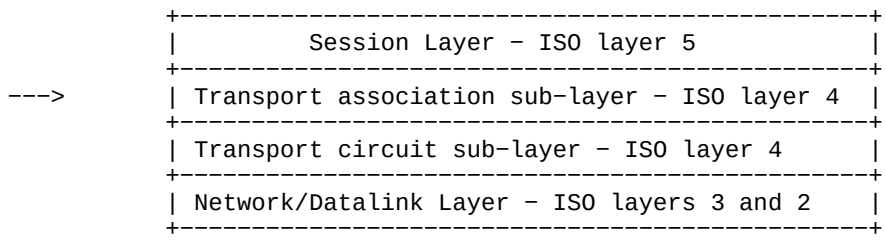
The following service class are identified by this architecture:

1. Disk Service
2. Queue Service

**NOTE**

**Service class 0 is undefined, and illegal if specified.**

## 5.2 INTERFACES MODELED AS FUNCTIONS




### 5.2.1 SESSION LAYER CLIENT INTERFACE TO TRANSPORT LAYER

The Session Layer (ISO layer 5) consists of clients and servers. A association can only be established by clients to servers. The clients and servers are referred to as users in the following section.

The association sublayer offer the following services:

- service class maintenance
- directory services
- association services
- transaction services

### 5.2.2 ASSOCIATION SUBLAYER OPERATION

The association sub−layer itself formats messages for transmission and validates received messages before passing the data to the Session layer user.

When a transaction is generated, a transaction slot is selected from those available. If a slot is available, a slot specific unique transaction sequence number is generated.

When a slot is not available, the transport will queue the transaction request on a first−come, first−serve basis. Transactions that are waiting for slots are not to be timed in any way.

When the association sublayer has selected a request message for retransmission, it must be retransmitted on the same slot as was originally selected. In addition, the sequence number field is to be incremented, causing this message to be viewed as a new transaction. Any context associated with sequence numbers previous to this is to be ignored.

If a transaction is aborted, the underlying association, and all multiplexed transactions must also be aborted. Notice that new associations will not purge the failed associations from the server circuit database. If transaction sequencing between the new association and the failed association is required, this must be managed at the application level utilizing the transaction identifiers. The problem is that the failed association's transactions may still complete successfully. (This could be solved by the transport using an association instance identifier which purges the previous failed association when new associations are formed. It is felt that this is more appropriately done by application layer mechanisms.)

An example of transaction retry:

Slots are composed of two fields: Slot number and a slot sequence number. Assume the client transport selects slot 4. A request message is generated using sequence number n, and transmitted. Some time later, the client transport detects that sequence n is still outstanding and selects it for retransmission. A new sequence number, n+1 is selected, and retransmitted, again using transaction slot 4. Subsequently, a response message for sequence number n is received. This response is to be ignored, as it is no longer current. When a response message to sequence n+1 is received, that response stream is processed, and the transaction is completed normally.

### 5.2.2.1 SUMMARY OF ASSOCIATION SUBLAYER FUNCTIONS

The association sub−layer offers the following hierarchy of functions to the client and server users in the ISO session layer:

| | Function offered: | Type of function: |
|---|---|---|
| S− | Start_service_class | directory service |
| C− | Solict_service | directory service |
| S− | Solict_service_poll | directory service |
| S− | Advertise_service | directory service |
| C− | Advertise_service_poll | directory service |
| C− | Start_association | association service |
| S− | Accept_association | association service |
| CS− | Association_status | association service |
| | Transaction_modes | |
| C− | Transaction_command | data service |
| S− | Transaction_response | data service |
| CS− | Transaction_status | data service |
| | Datagram_mode | |
| CS− | Transmit_datagram | data service |
| CS− | Receive_datagram | data service |
| CS− | End_association | association service |
| S− | End_service_class | directory service |

C = Client user S = Server user

#### 5.2.2.2 SUMMARY OF INTERFACE CALLS

In interface calls:

- input parameters are specified first
- input parameters are separated from output parameters by a semicolon
- parameters are separated by commas
- XCB − Reference to the Transaction Control Block. This contains references to the command, response and datagram buffer descriptors as well as the Command_response_timer, Transaction_response_timer, Transaction_transmit_limit, Transaction_failure_option and the Transaction_id.

  Transaction_failure_options are abort on Circuit_fault or ignore Circuit_fault.
- ACB − Association Control Block. This contains references to the remote association partner as well as references to association communication mechanism.
- "C−","S−", and "CS−" indicate that a function is available at the client user ("C−"), server user ("S−") or both ("CS−").

Within the interface calls, the "reason" argument is defined separately for each function call.

- Transaction_command (ACB, XCB, Transaction_complete_callback; Status)
- Transaction_status (ACB, XCB, command_callup; Status)
- Transaction_response ( ACB, XCB; Status)

## 5.3 ASSOCIATION STATE VARIABLES

The state transitions for associations are similar to those for the underlying circuit itself. Any of the association transitions shown in the state tables assume an underlying circuit in the Running state. If the circuit should exit the Running state, the associations immediately transfer to the halted state and the clients and servers are notified.

The state of a association is captured by each end in a data structure called the association control block (ACB). Architected state is associated with each end of a association. Changes to the ACB are caused by events which are defined in the following sections. An individual ACB is associated with either the client end of a association or the server end of a association.

There are four ACB states: Starting, Running, Aborting and Halted:

- ACBs start in the Halted state. ACBs are assumed to be in the Halted state even if they do not (yet) exist.
- When the first command to communicate with a remote server is received, the ACB enters the Starting state.
- Once an association has been established, the ACB reaches the Running state.
- When a user initiated disconnect is requested, the association enters the aborting state. This can also occur as a result of a Circuit_down event or a RESPONSE_DISCONNECT in reply to a COMMAND_CONNECT message.

The association consists of "state variables" held in client and server ACBs. The ACB are used as input to drive the association state tables.

A example of an implementation of association state variables:

- ACB_SERVICE_DESCRIPTOR − The address of the service class descriptor.
- ACB_CSB − The address of the underlying CSB.
- ACB_STATE − One of: Halted, Starting, Aborting, Running.
- ACB_DST_ACB_ID − a reference to a remote ACB.
- ACB_SRC_ACB_ID − a reference to this ACB.
- ACB_XCB_DEFAULT_SLOTS − The default number of slots for this ACB.
- ACB_XCB_SLOTS − Maximum number of transaction slots that can be used on this association.
- ACB_XCB − An table of XCB addresses and status.
- ACB_QUEUED_XACT − A queue of waiting XCBs.
- ACB_INTERFACE − A vector table for service calls (implementation dependent)
- ACB_SERVICE_NAME_DESCRIPTOR − The description of the bound service.
- ACB_DATA_SEGMENT_SIZE − Maximum user data size that can be offered to the data link in a single message.

Whenever an ACB is "initialized", the state variables assume the following values:

1. ACB_SERVICE_DESCRIPTOR − A value assigned by the LAST architecture.
2. ACB_CSB − the associated CSB address.
3. ACB_DST_ACB_ID − zeroed.
4. ACB_SRC_ACB_ID − a reference to this ACB.
5. ACB_STATE − set to Halted.
6. ACB_TRANSACTION_WAIT_QUEUE − empty.
7. ACB_MAX_XCB_SLOTS − Constant negotiated at association establishment.

Throughout following sections, labels that start with "ACB_" refer to the ACB state variables. Labels such as SRC_ACB_ID refer to fields in messages defined in the Message Format section.

## 5.4 TRANSACTION STATE VARIABLES

Some of the values maintained in the XCB are not visible to different layers of the architecture. The notation used below shows which layers can manipulate the variable. "U" indicates the user layer, "A" indicates the association layer.

The architecture describes the following state variables:

- XCB_USER_DESCRIPTOR (U) − User layer data.
- XCB_TRANSACTION_IDENTIFIER (U,A) − A user supplied transaction identifier.
- XCB_COMMAND_BUFFER (U) − A description of the command.
- XCB_RESPONSE_BUFFER (U) − A description of the response.
- XCB_DATAGRAM_BUFFER (U) − A description of the datagram buffer.
- XCB_COMMAND_RESPONSE_TIMER (U,A) − The Command_response_timer value.
- XCB_RESPONSE_TIMER (U,A) − The Transaction_response_timer value.
- XCB_TRANSMIT_LIMIT (U,A) − The Transaction_transmit_limit.
- XCB_TRANSMIT_COUNT (A) − The number of command attempts transmitted.
- XCB_TIMER (A) − The Transaction_timer itself.
- XCB_SEGMENT_BITMASK (A) − A map of processed response segments.
- XCB_ASSOCIATION_DESCRIPTOR (A) − Association layer data.
- XCB_CIRCUIT_DESCRIPTOR (A) − Circuit layer data.

## 5.5 DEFINITION OF EVENTS

The are three types of events possible:

- Illegal events − These are illegal message formats and illegal function call formats. Unless otherwise noted, this event type causes the Illegal event to be logged and the state machine to immediately halt (crash). This event type does not appear explicitly in the operational model. Illegal events all are labeled with an "Ill_" prefix.
- Invalid event − Invalid events are normally the result of improper synchronization between association clients and association servers. These events occur infrequently; the event is treated as described in the state diagrams. Invalid event all are labeled with an "Inv_" prefix.
- (Valid) event − All other events fall into this category. The event actually drive the operation of the association sub−layer state diagrams. Valid events are not labeled with a prefix.

There are three sources of the three event types:

- higher level interface events − This is the interface between the session layer and the transport association sub−layer. These events are modeled as function calls.
- intra−layer events − These events correspond to timers and counters reaching architecturally specified values.
- circuit sub−layer interface events − This is the interface between the transport association sub−layer and the transport circuit sub−layer. These events are modeled as "message received" event and "command queued" actions.

The higher layer interface events correspond to the association interface function calls. The following list contains the possible events corresponding to the function calls. The values in parenthesis are the abbreviation of the event name used in the state tables:

1. Start_association − a client commands a association be started. The association sub−layer implementation would allocate an ACB at this point if none existed.
2. New_association_poll − this corresponds to an action in the association state diagram.
3. End_association − a client or server commands a association be terminated.
4. Queue_transaction_command − a client queued a transaction.
5. New_transaction_poll − a server is polling for a new transaction to work on. This is modeled as an action routine in the server association state table.
6. Queue_transaction_response − a server queued a response to a transaction.
7. Poll_transaction_status − A client or server commands the status of a transaction.

There are intra−layer events:

- Transaction_timer − Timer event for a transaction.

There are four message subtypes involved in the association sub−layer, each of which corresponds to a "message received" event or a "message queued" action:

- COMMAND_CONNECT/RESPONSE_CONNECT − A message sent to start a new association.
- COMMAND_DATA/RESPONSE_DATA − A message sent to communicate data over an established associa−tion.
- COMMAND_RESYNC/RESPONSE_RESYNC − A message used to update transaction state.
- COMMAND_DISCONNECT/RESPONSE_DISCONNECT − A message used to halt a association.

## 5.6 VALIDATION OF EVENTS

The association sub−layer filters all events into one of the three different events types: Illegal, Invalid, Valid.

The validation filters are ordered. The filters must generate Illegal events first, Invalid events second and Valid events third. Illegal events are labeled with an "Ill_" prefix, Invalid events with an "Inv_" prefix and Valid events do not have a prefix.

### 5.6.1 ILLEGAL EVENT FILTER

Messages received and transmitted over the datalink must be validated:

- Ill_format − the fields in the messages must correspond to the formats listed in the message format section. If this check should fail, an Ill_format event is declared and the ACB transfers immediately to the Halted state.

The function calls are validated based on implementation supplied policies. If this check fails, an Ill_function_ format event is declared.

The intra−layer events are always assumed to be valid (clock and counters).

### 5.6.2 INVALID EVENT FILTER

COMMAND_CONNECT messages are mapped onto ACBs based on the value of the SRC_ACB_ID in the message and DST_CSIR_ID in the message. All ACBs associated with the single CSB are searched to see if any ACB_ DST_ACB_ID field matches the received SRC_ACB_ID. A match causes the "Inv_connect_rcv" event.

All other events are mapped onto the corresponding ACB based solely on the value of the DST_ACB_ID and SRC_ACB_ID field of the received message. A summary of the way received messages events sent over the datalink are invalidated ( xxx refers to any of the received message types except the COMMAND_CONNECT and RESPONSE_DISCONNECT messages):

- Inv_xxx_rcv − DST_ACB_ID not equal to ACB_SRC_ACB_ID.
- Inv_xxx_rcv − SRC_ACB_ID not equal to ACB_DST_ACB_ID.
- Inv_xxx_rcv − DST_ACB_ID and/or SRC_ACB_ID is equal to 0.

A RESPONSE_DISCONNECT is invalid if:

- Inv_Response_disconnect_rcv − DST_ACB_ID not equal to any ACB_SRC_ACB_ID
- Inv_Response_disconnect_rcv − SRC_ACB_ID not equal to 0.
- Inv_Response_disconnect_rcv − DST_ACB_ID equal to 0.

### 5.6.3 VALID EVENTS

In the case of the first Start_association event in the client and the first New_association_poll event in the server, no ACB will exist to reference. An ACB should be allocated and the state variables should be initialized as described above.

Any events which passed the first two filtering level are Valid events.

## 5.7 OPERATION OF STATE DIAGRAMS

There are two independent state diagrams associated with the operation of the association sub−layer. One state diagram operates on behalf of clients (client association state table) and the other on behalf of servers (the server association state table). The association state is always distributed between a client state table and a server state table never client−client or server−server.

Circuit_down events should be mapped as Retransmit_limit events in these state diagrams.

An Association_failure event causes the association sublayer to abort all transactions first, and then notify the user of the association failure.

RUN messages are exchanged to establish an association, transfer request and response data, maintain transaction context, and terminate associations.

Successful transmission of RUN messages may result in association state transitions as follows: When a COMMAND_CONNECT is transmitted, the association state transitions from Halted to Starting. When a RESPONSE_CONNECT message is transmitted, the association state transitions to Starting. When a RESPONSE_CONNECT message is received, the association state transitions from Starting to Running. When a RUN message is received, a state transition from Starting to Running takes place. When a COMMAND_DISCONNECT is transmitted, a association transition from the Running (or Starting) state to the Aborting state. When a COMMAND_DISCONNECT is received, a association state transition from Running (or Starting) to Aborting is made. When a RESPONSE_DISCONNECT message is received, a association is transitioned to the Halted state. Likewise, a association transition to Halted occurs when a RESPONSE_DISCONNECT message is transmitted.

This latter transition can be problematical in that the RESPONSE_DISCONNECT message may not be received by the requesting transport. As a result, when a (retransmitted) COMMAND_DISCONNECT is received for a association that is Halted, a RESPONSE_DISCONNECT message must be transmitted in response regardless. This assumes that there is a Running circuit. If no Running circuit is present in the slave, the master will receive a STOP message.

### 5.7.1 CLIENT ASSOCIATION STATE TABLE

**Table 1: Client Association State Table**

| State | Event | Action | Next State |
|---|---|---|---|
| Halted | Start_association | Init ACB, send COMMAND_CONNECT, start timer, set retransmit limit | Starting |
| | any other | none | Halted |
| Starting | RESPONSE_CONNECTECT | Process RESPONSE_CONNECT message stop timer, notify client | Idle |
| | RESPONSE_DISCONNECT | Process RESPONSE_DISCONNECT,stop timer, notify client | Halted |
| | Timer | Resend COMMAND_CONNECT | Starting |
| | Retransmit_limit | Declare association failure | Halted |
| | Any other | None | Starting |
| Idle | End_association | Send COMMAND_DISCONNECT | Aborting |
| | RESPONSE_DISCONNECT | Notify client association aborted, stop timer, process RESPONSE_DISCONNECT | Halted |
| | Any other | none | Idle |
| Aborting | Timer | Resend COMMAND_DISCONNECT | Aborting |
| | RESPONSE_DISCONNECT | Process RESPONSE_DISCONNECT, stop timer, notify client | Halted |
| | Any other | Send COMMAND_DISCONNECT | Aborting |
| | Retransmit_limit | Declare Circuit_fault event | Halted |

### 5.7.2 CLIENT TRANSACTION IDEMPOTENT_MODE STATE TABLE

The client end of a association has three independent transaction state diagrams, only one of which is described in this document: Idempotent mode.

**Table 2: Client Transaction Idempotent_mode State Table**

| State | Event | Action | Next State |
|---|---|---|---|
| Halted | Trans_command | Send COMMAND_DATA, set transaction timer to Cmd_rsp_timer, set Transaction_transmit_limit | Waiting |
| | Any other | None | Halted |
| Waiting | Timer | Resend COMMAND_DATA, declare command failure, set Transaction_timer to Command_response_timer value | Waiting |
| | RESPONSE_RESYNC | Set Transaction_timer to Transaction_response_timer value | Waiting |
| | RESPONSE_DATA | If RESPONSE_DATA is incomplete, set Transaction_timer to Command_response_value | Waiting |
| | RESPONSE_DATA | If RESPONSE_DATA is complete, complete successful transaction | Halted |
| | Retransmit_limit | Declare Transaction_failure | Circuit |
| Circuit | Circuit_fault | Abort association and all bound transactions | Halted |
| | Circuit_up | Resend COMMAND_DATA, set Transaction_timer to Command_response_timer, set Transaction_transmit_limit | Waiting |
| | Circuit_down | Abort association and all bound transactions | Halted |

### 5.7.3 CLIENT DATAGRAM MODE STATE DIAGRAM

The client may transmit and receive Datagram_mode message streams at any point in the previous state diagram. Any and all state information is a association user responsibility. Message stream launched will be signaled to the client user, and will constitute success indication.

### 5.7.4 SERVER ASSOCIATION STATE TABLE

**Table 3: Server Association State Table**

| State | Event | Action | Next State |
|---|---|---|---|
| Halted | COMMAND_CONNECT | Init ACB, deliver event to user | Starting |
| | Any other | Send RESPONSE_DISCONNECT | Halted |
| Starting | End_association | Send RESPONSE_DISCONNECT | Halted |
| | Accept_association | Send RESPONSE_CONNECT | Idle |
| | Any other | None | Starting |
| Idle | End_association | Send RESPONSE_DISCONNECT | Halted |
| | COMMAND_DISCONNECT | Deliver event to association, send RESPONSE_DISCONNECT | Halted |
| | Any other | None | Idle |

A server implementation may choose to always accept a commanded association. In this case the Starting state in the table would not exist. The COMMAND_CONNECT message event would establish a association (assuming sufficient resources). A server might then later reject the association via the End_association/RESPONSE_DISCONNECT mechanism.

A server may refuse to accept the commanded association. This will cause a REPONSE_DISCONNECT message to be sent out indicating the servers refusal and the association will be aborted.

### 5.7.5 SERVER TRANSACTION IDEMPOTENT_MODE STATE TABLE

The following state table describes the transaction flow in the transport association sublayer in response to a COMMAND_DATA for a given slot.

**Table 4: Server Transaction Idempotent_mode State Table**

| State | Event | Action | Next State |
|---|---|---|---|
| Idle | Segment_1 | Call association server | Receiving |
| | Segment_x | Queue request segment | Waiting |
| Waiting | Segment_1 | Call association user | Receiving |
| | Segment_x | Queue request segment | Waiting |
| | New_transaction | Release cached segments, carry New_transaction event forward to Idle | Idle |
| Receiving | All_segments | Call association user | Running |
| | Segment_x | Copy request data | Receiving |
| | New_transaction | Notify server of abort, release cached segments, carry New_transaction event forward to Idle | Idle |
| Running | Resource_error | Abort transaction, do not notify association user | Idle |
| | Transaction_response | Send response segment stream | Idle |
| | New_transaction | Notify server of abort, carry New_transaction event forward to Idle | Idle |

The transaction sub−layer notifies the association user of a new transaction when segment 1 of transaction is received. If segments are received out of order, this layer queues the segment to be processed after segment 1 has been received.

A client may retransmit a transaction request. Any segment of the command retransmission could be received while waiting for intermediate message segments of a transaction command stream or while the server is actually processing the current command. When such a new transaction (New_tran event) is received in this way, any previous transaction should be aborted before processing the new transaction as shown above.

A server may choose to not accept the transaction. It will signal this during the initial call. Additional segments to a server aborted request should be ignored.

If resource errors should occur, the implementation might abort the transaction (notifying the association user) and reclaim resources.

### 5.7.6 SERVER DATAGRAM MODE STATE DIAGRAM

The server may transmit and receive Datagram_mode message streams at any point in the previous state diagram. Any and all state information is a association user responsibility. Message stream launched will be signaled to the server user, and will constitute success indication.

# CHAPTER 6

# TRANSPORT CIRCUIT SUB−LAYER

The purpose of the circuit sub−layer is to detect node incarnations (crashing, rebooting and topology changes) and to provide higher level association context for multiplexing data. The circuit allows multiplexing of multiple association level commands in a single data link message.

A maximum of one circuit at a time can exist between any node pair.

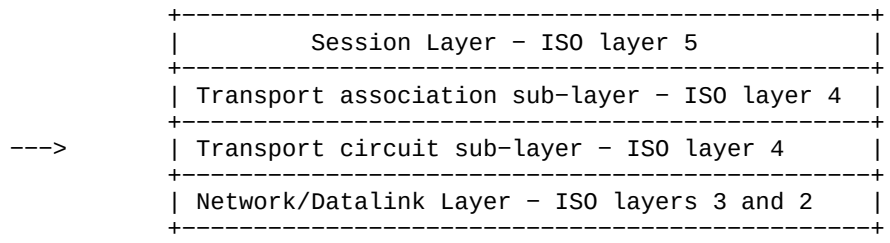
The circuit layer provides a simple form of rate based congestion control for all users of this layer.

LAST circuits do not provide a message sequencing service. Thus, duplicate detection and retransmission policy is left to the association sub−layer of the transport.

The circuit sub−layer is modeled in six sections:

- Interfaces modeled as function calls
- circuit state variables
- definition of events
- validation of events
- operation of a state diagram
- congestion control

## 6.1 INTERFACES MODELED AS FUNCTIONS



### 6.1.1 ASSOCIATION SUB−LAYER INTERFACE TO CIRCUIT SUB−LAYER

In review, the association sub−layer consists of client and server association managers. This client generates transactions into slots which are matched by the transport association to server slots.

This circuit interface provides the following services:

- A directory service
- A circuit quality service − This indicated destination unreachable.
- A circuit down service − This indicates the destination node has reinitialized.
- A datagram service

#### 6.1.1.1 SUMMARY OF FUNCTIONS

The circuit sub−layer offers the following hierarchy of functions to the client and slave users in the association layer:

```
+-----------------------------------------+
|           Function offered:             |
+-----------------------------------------+-------------------+
|C−   Solict_service                      | directory service |
|S−   Solict_service_poll                 | directory service |
|S−   Advertise_service                   | directory service |
|C−   Advertise_service_poll              | directory service |
|M−       Circuit_start                   | circuit service   |
|Sl−      Poll_circuit_start              | circuit service   |
|C−           Transaction_retry           | path service      |
|C−           Queue_command_data          | data service      |
|C−           Queue_command_data_poll     | data service      |
|S−           Queue_response_data         | data service      |
|CS−          Response_data_poll          | data service      |
|CS−          Queue_resync_data           | data service      |
|CS−          Queue_resync_data_poll      | data service      |
|MSl−     Circuit_halt                    | circuit service   |
+-----------------------------------------+-------------------+
```

#### 6.1.1.2 SUMMARY OF INTERFACE CALLS

In interface calls:

- input parameters are specified first
- input parameters are separated from output parameters by a semicolon
- parameters are separated by commas
- "M−","Sl−", and "MSl−", "C−","S−", and "CS−" indicate that a function is available at the corresponding interface.
- Transaction_retry (NDB, NDB_first_path, resync_bitmap)

### 6.1.2 CIRCUIT SUB−LAYER INTERFACE TO DATALINK

The interface to the datalink is a datagram service which supports both physically addressed packets and multicast addressed packets.

## 6.2 CIRCUIT STATE VARIABLES

The circuit consists of "state variables" held in master and slave memory structure called circuit state blocks (CSBs). The CSB are used as input to drive the circuit state table. The circuit state blocks contain the following variables:

- CSB_SOURCE_NODE_ID − A unique identification for the local node.
- CSB_DESTINATION_NODE_ID − The unique identification associated with the remote node.
- CSB_NDB − the address of the data structure describing the remote node.
- CSB_DST_CIR_ID − the index associated with the remote CSB. This value is passed in transmitted messages to assist in efficiently mapping messages onto the proper remote CSB.
- CSB_SRC_CIR_ID − the index associated with the local CSB. This value is passed in received messages to assist in efficiently mapping messages onto the proper local CSB.
- CSB_STATE − one of Halted, Start_sent, Stack_sent, or Running.
- CSB_BURST_SIZE − The number of data messages that can be transmitted onto the datalink addressed to CSB_DST_ADDRESS without delay.
- CSB_BURST_CREDITS − The count of bursts that can be transmitted.
- CSB_BURST_DELAY − The amount of time to wait between message bursts.
- CSB_TIMER − the circuit startup timer. This value counts to a specified architectural value upon which the Start_timer intra−layer event is declared.
- CSB_RESEND_LIMIT − A count of the number of Start messages retransmitted. If an architecturally defined value is reached, the intra−layer event Resend_limit is declared.

Whenever a CSB is "initialized", the state variables assume the following values:

1. CSB_SRC_CIR_ID <− Set to a value which uniquely and conveniently identifies the CBS in which the field is contained.
2. DST_CIR_ID <− zeroed.
3. CSB_STATE − Halted.
4. CSB_BURST_CREDITS <− One.
5. CSB_BURST_SIZE <− One.
6. CSB_BURST_DELAY <− Zero.
7. CSB_START_TIMER − Zeroed, started at Halted/Starting transition.
8. CSB_RESEND_LIMIT − Zeroed.

## 6.3 DEFINITION OF EVENTS

The are three types of events possible:

- Illegal events − These are illegal message formats and illegal function call formats. Unless otherwise noted, this event type causes the Illegal event to be logged and the state machine to immediately halt (crash). This event type does not appear explicitly in the operational model. Illegal events all are labeled with an "Ill_" prefix.
- Invalid event − Invalid events are normally the result of improper synchronization between nodes. These events occur infrequently the event is treated as described in the state diagrams. Invalid event all are labeled with an "Inv_" prefix.
- (Valid) event − All other events fall into this category. The events actually drive the operation of the circuit sub−layer state diagram. Valid events are not labeled with a prefix.

There are three sources of the three event types:

- association sub−layer interface events − This is the interface between the transport association sub−layer and the transport circuit sub−layer. These events are modeled as function calls.
- intra−layer events − These events correspond to timers and counters reaching architecturally specified values.
- datalink interface events − This is the interface between the transport circuit−sublayer and the datalink. This is modeled as a datagram interface and must provide asynchronous notification of transmit completion to the circuit sublayer.

  There are five types of messages involved in the circuit sub−layer:

  - START − A message sent by a master to start a new circuit.
  - STACK − A message sent by a slave to partially complete the formation of a new circuit.
  - RUN − A message sent to communicate data over an established circuit.
  - STOP − A message used to halt a Running circuit.
  - ADVERTISEMENT − A message used to communicate transport topology and directory services.

The complete list of possible events:

1. VC_start − the association sub−layer commands a circuit be started. The circuit sub−layer implementation would allocate a Circuit Block at this point if none existed.
2. VC_halt − association sub−layer commands that the circuit be halted.
3. Start_rcv − START message received from the datalink. An implementation would allocate a Circuit Block at this point, if one did not exist from a previous circuit, and initialize all circuit state variables.
4. Inv_Start_rcv − invalid START received.
5. Stack_rcv − A STACK message is received from the datalink.
6. Inv_Stack_rcv − An invalid STACK message is received from the datalink.
7. Stop_rcv − STOP message received.
8. Inv_stop_rcv − invalid STOP received.
9. Run_rcv − RUN message received.
10. Inv_run_rcv − RUN message received with invalid circuit identification (see next section).
11. Start_Timer − the circuit startup timer expires.
12. Resend_limit − The maximum number of START retransmits before the circuit is declared Halted.
13. Command_data (master only) − Client data is queued to transport layer.
14. Command_failure − An association suspects a circuit fault.
15. Transaction_failure − An association believes a circuit has faulted.

## 6.4 VALIDATION OF EVENTS

The circuit−sublayer filters all events into one of the three different events types: Illegal, Invalid, Valid.

The validation filters are ordered. The filters must generate Illegal events first, Invalid events second and Valid events third. Illegal events are labeled with an "Ill_" prefix, Invalid events with an "Inv_" prefix and Valid events do not have a prefix.

### 6.4.1 ILLEGAL EVENT FILTER

Messages received and transmitted over the datalink must be validated:

- Ill_xport_format − the LAST transport fields must conform to the message format section. If this check fails, an Illegal_msg_format event is declared.
- Ill_start_rcv − DST_CIR_ID not equal to zero or the SRC_CIR_ID is equal to 0.
- Ill_stop_rcv − SRC_CIR_ID of received message not equal to 0.

The function calls are validated based on implementation supplied message formats. If this check fails, an Illegal_ function_format event is declared.

The intra−layer events are always assumed to be valid (clock and counters).

### 6.4.2 INVALID EVENT FILTER

Any START, STACK, RUN, or STOP message received with the multicast destination address should cause and Inv_xxx event.

START messages are mapped onto circuit blocks based on the value of the START.SOURCE_NODE_IDENTIFICATION field. A match to a circuit block is found if START.SOURCE_NODE_IDENTIFICATION field equals the CSB_ DST_NODE_ID in any of the CSBs on the local system. If such a match is found for a CSB in any state other than halted, a Circuit_down event is declared for that CSB, then Start_rcv event is declared.

STACK, RUN and STOP messages are mapped onto the corresponding events based solely on the value of the DST_CIR_ID. The CSB_SOURCE_NODE_ID is compared with START.SOURCE_NODE_IDENTIFICATION of the received message. If the identification fields are unequal then the corresponding Inv_xxx event is declared (eg. Inv_run).

A summary of the way received messages sent over the datalink are invalidated:

- Inv_start_rcv − Multicast destination address.
- Inv_stack_rcv − SRC_CIR_ID not equal to CSB_DST_CIR_ID or node identification check fails.
- Inv_stack_rcv − DST_CIR_ID and/or SRC_CIR_ID is equal to 0 or node identification check fails.
- Inv_run_rcv − DST_CIR_ID not equal to CSB_SRC_CIR_ID or node identification check fails.
- Inv_run_rcv − SRC_CIR_ID not equal to CSB_DST_CIR_ID or node identification check fails.
- Inv_run_rcv − DST_CIR_ID and/or SRC_CIR_ID is equal to 0 or node identification check fails.
- Inv_stop_rcv − DST_CIR_ID not equal to CSB_SRC_CIR_ID or node identification check fails.

### 6.4.3 VALID EVENTS

In the case of the first VC_start event in the master, or the first Start_rcv event in the slave, no CSB will exist to reference. A CSB should be allocated and the state variables should be initialized as described above.

On the slave end, the START.SOURCE_NODE_IDENTIFICATION, and the START.SRC_CIRC_ID should be copied to the corresponding CSB fields. The START.DST_CIR_ID field should be ignored. If a CSB cannot be allocated, the implementation should attempt to send a STOP message that indicates no resources.

Any message which passed the first two filtering level must be Valid. The list of Valid events is presented here for completeness:

- VC_start − The circuit sub−layer generates this event when an start association is request, and there is no CSB present.
- VC_halt − Association sub−layer commands that the final association for this circuit be terminated.
- Start_rcv − Described above.
- Run_rcv − DST_CIR_ID equals CSB_SRC_CIR_ID and the STOP.SOURCE_NODE_IDENTIFICATION equals :CSB_SRC_NODE_ID
- Stop_rcv − DST_CIR_ID equals :CSB_SRC_CIR_ID and the STOP.SOURCE_NODE_IDENTIFICATION equals :CSB_SRC_NODE_ID
- Resend_limit
- Start_timer

### 6.4.4 EVENT VALIDATION SUMMARY

Illegal messages are those that do not conform to the defined message formats. These messages should not occur, but if they do, an error should be logged, as much of the message as possible should be stored in order to diagnose the failure, and the message should be discarded.

Illegal messages never become events which drive state transitions.

## 6.5 CIRCUIT STATE TABLE

In the diagram below, Start_rcv/H is a START message received in which the START.SOURCE_NODE_IDENTIFICATION is greater than CSB_SOURCE_NODE_ID. Start_rcv/L is the inverse. Start_rcv is an event in which the comparison is not relevant.

Datalink addresses are compared as unsigned integer values. The least significant bit is the bit received first over the datalink.

START/STACK messages are exchanged when no circuit association exists with a remote node. Transmission of START and STACK messages is coupled with circuit state changes. When a START message is transmitted, the circuit state transitions from Halted to Start_sent. When A STACK message is transmitted, the circuit state transitions from Halted to Stack_sent. When a STACK message is received, the circuit state transitions from Start_sent to Running. When the first RUN message is received, a circuit transitions from Stack_sent to Running.

It is possible that both ends of a circuit simultaneously attempt to establish a circuit association. In this case, START message may be received when the circuit state is Start_sent. Should this condition be detected, the node id of the remote node should be compared to the node id of the receiving node. The smaller node id should back off, and permit the higher node id to establish the circuit association. Once the circuit association is established, RUN messages can be exchanged normally.

If a START message is received from a node that already has a circuit association established, that association will be terminated, and upper association layer interfaces be notified. Once the upper layers have terminated association and user associations, the circuit can be re−established.

"process START" means copy the SRC_CIR_ID to CSB_DST_CIR_ID, copy SRC_NODE_ID to CSB_DST_NODE_ID, and copy the START.SOURCE_NODE_IDENTFICATION to the CSB_DESTINATION_NODE_ID.

"process STACK" means copy the START.SRC_CIR_ID to CSB_DST_CIR_ID and verify START.SOURCE_NODE_ID equals CSB_DESTINATION_NODE_ID.

**Table 5: Circuit State Table**

| State | Event | Action | Next State |
|---|---|---|---|
| Halted | Vc_start | Initialize, Send START | Start_sent |
| | Start_rcv | Initialize, process START and send STACK | Stack_sent |
| | Stop_rcv | None | Halted |
| | Inv_stop_rcv | None | Halted |
| | Other message events | Send STOP.DST_CIR_ID equal to SRC_CIR_ID of received message | Halted |
| | Other events | None | Halted |
| Start_sent | Stack_rcv | Process STACK | Running |
| Start_rcv/H | Initialize, process START and send STACK | Stack_sent | |
| | Start_rcv/L | Ignore message | Start_sent |
| | Resend_limit | Notify clients | Halted |
| | Start_Timer | Resend START | Start_sent |
| | Stop_rcv | Notify clients | Halted |
| | VC_halt | Notify clients | Halted |
| | Other events | Ignore | Halted |
| Stack_sent | Run_rcv | Move to Running state and process message | Running |
| | Stop_rcv | Notify clients | Halted |
| | VC_halt | Send STOP message (with reason) | Halted |
| | other events | Ignore | Stack_sent |
| Running | Run_rcv | Process received message | Running |
| | VC_halt | Send STOP (with reason) | Halted |
| | Stop_rcv | Declare Circuit_down event | Halted |
| | Start_rcv | Declare Circuit_down, initialize, process START and send STACK | Stack_sent |
| | Other events | None | Running |

# CHAPTER 7

# DIRECTORY SERVICE AND TOPOLOGY MAINTENANCE

Directory service and topology maintenance algorithms utilize a class of messages which utilize multicast addressing and are described below.

These messages are comprised of three different message sub−types: ADVERTISEMENT, SOLICIT AND SOLICIT_RESPONSE.

## 7.1 DIRECTORY SERVICE

Directory service is a mechanism available to clients of the transport to discover services offered by servers. In addition, a mechanism is available for servers to advertise service offerings. These directory services could be implemented by an external naming service. In this case, the naming service syntax and semantics must be consistent with the concepts presented in this document. For instance, protocol revision checking would become a naming service function.

Directory service messages are used for four distinct purposes:

- Advertise services
- Solicit for a service
- Respond to a solicit service request
- Map the LAST topology

The three types of directory service message formats are identical except for the MSG_TYPE indicator field in the virtual circuit header.

The SERVICE_CLASS field and the SERVICE_NAME field are specified by the transport users. The content of the SERVICE_CLASS field is architecturally defined in this document. The actual service name is ignored by the transport layer.

## 7.2 DIRECTORY SERVICE GROUPING

LAST provides for the grouping of clients and servers in the network by group codes. This code will determine the subset of nodes that will participate in naming services. By using group codes, a network can be arbitrarily segmented into classes of nodes. These classes might be established because of the type of network involved (such as extended LANs connected by bridges), or by the types of services provided.

The group code will be specified as an integer in the range 0 − 1023 that will be added to the base address word of the multicast address. The default group will should be group zero unless otherwise specified.

Minimally, each implementation must be able to participate in one selected group, but may optionally participate in multiple groups.

### 7.2.1 ADVERTISING ALGORITHM

Advertising is indicated when new services are made available and during path maintenance.

Whenever advertising is indicated the ADVERTISEMENT message should be transmitted over all adapters a minimum of 3 times at one second intervals. Additionally, it should be transmitted to all directory service groups enabled.

### 7.2.2 SOLICITING ALGORITHM

Soliciting is used to discover services in the network and to maintain topology. When a client wants to use a particular service, and it doesn't know where the service might exist, it will utilize the soliciting algorithm.

The SOLICIT_RESPONSE.REQUEST_SEQUENCE field is always set to the value of the SOLICIT.REQUEST_ SEQUENCE field specified by the circuit master entity.

Servers providing the indicated service will respond using the directory service interface and the transport slave will generate a physically addressed SOLICIT_RESPONSE message.

Solicitation messages may require retransmission when no response is received. Implementations will solicit on enabled directory service groups.

## 7.3 TOPOLOGY MAINTENANCE

All directory service messages are used by the circuit sublayer to maintain the topology database.

The topology database is implementation specific and maintains:

- Node incarnation
- Node identification
- Node name
- Protocol version information
- Paths to each node (local and remote).
  - — Local adapter data structures
  - — For each local adapter, a remote node datalink address list

It is important to note that the topology database is maintained independently of existing circuits.

The circuit master is responsible for maintaining the topology database. Before a START message is sent, the master may detect that the circuit slave is addressable on multiple paths by using SOLICIT messages on each local adapter. Soliciting for topology maintenance should only use the directory service group in which the service was found, and not all enabled service groups.

The circuit slave will similarly map the paths to the circuit master as SOLICT messages are received.

The topology database maintenance messages set the SERVICE_CLASS and SERVICE_NAME_LENGTH fields equal to zero. If the SOLICIT.SERVICE_CLASS is zero, the circuit slave must generate a physically addressed SOLICIT_RESPONSE message to the circuit master over the corresponding path.

Path failure detection is the responsibility of the circuit master as described in the next section.

### 7.3.1 ASSOCIATION SUBLAYER PATH ALGORITHMS

The LAST architecture specifies that there may exist multiple adapters connected for any node in the network. Any two adapters form what is termed a path. The circuit sublayer maintains a database of all paths from the local node to any destination node.

On occasion, paths will become unusable for one reason or another. The client association sublayer detects such command failures when the Command_response_timer expires or when RESYNC messages are received, and declares a Command_failure event. The transaction retries are continued normally.

The association sublayer is unaware of the actual path database. It simply offers an XCB containing a descriptor of the user transaction data to the circuit layer interface routine Queue_command_data.

The Command_failure event is associated with the transaction control block (XCB) when it is declared. The XCB contains state information specific to the circuit layer which is used to help identify which path may be faulting.

If the client reaches the Transaction_transmit_limit, it declares a client user, the transaction may be aborted or suspended at this time. If suspended, the transaction execution is deferred until either a Circuit_fault, Circuit_down, or Circuit_up event is declared. And then if the transaction was suspended, the transaction is restarted with its initial timer and transmit limit values.

**TRANSACTION INTEGRITY CONSIDERATION**

**The transaction failure causes the old association to be aborted. The client user must be aware that if a new association is formed, old transactions on the previous association might still succeed after transactions successfully complete on the new association. This might be a problem for update transactions. This is very unlikely, but must be dealt with by the higher level protocol if operational integrity if affected by this scenario.**

**The transport layer does not have sufficient information to effectively deal with this problem. The server user must terminate any old associations before accepting new associations to prevent this scenario.**

### 7.3.2 CIRCUIT LAYER PATH ALGORITHMS

The circuit layer maintains an incarnation value associated with each remote node. The value is unique and changes each time a new transport instance is created. Should a message be received from an existing remote node with a different incarnation value, the circuit state associated with the node (i.e., all association and active transactions) will be aborted. This same message may be used to form a new circuit to the remote node.

The circuit layer is selective about including paths to remote nodes. For instance, circuit masters refuse to map paths to other master transport instances to conserve resources.

In order to utilize multiple paths concurrently, the circuit layer exchanges periodic messages which are used to maintain the circuit path database. When a node is detected to be addressable via two or more paths, the circuit layer will record each individual path to the remote node, and will transmit messages across all available paths.

The circuit sublayer, when called at the Transaction_retry routine, uses that information to determine which paths may have failed. The circuit layer tabulates these events in the PATH_QUALITY_COUNTER. When this value exceed Path_quality_threshold, the suspect path is tested (as described below) to see if it is inoperative. The PATH_QUALITY_COUNTER is zeroed when any message is received over the path.

During path maintenance, the circuit master will transmit a physically addressed SOLICIT message over the suspect path. The circuit master will retry this operation at Round_trip_message delay intervals, for Circuit_transmit_limit times, or until a SOLICIT_RESPONSE message is received.

At this point in time, if multiple paths are available, the suspect path is omitted by the master. Notice that during path maintenance, the slave is unaware of possible faulting paths.

If a SOLICIT_RESPONSE message is not received, the Master will generate a physically addressed SOLICIT message with the PURGE_ALL_PATHS_FLAG set on all paths the Slave is known to exist on. After transmitting the final SOLICIT message, the Master will then execute the advertising algorithm.

On receipt of a SOLICIT_RESPONSE, the circuit master will declare a Circuit_up event on the circuit associated with the path.

# CHAPTER 8

# RATE BASED ALGORITHMS

## 8.1 CIRCUIT RATE BASED CONGESTION CONTROL

The congestion control algorithm is symmetrical and is based on three architected values: Maximum_message_rate, Current_message_rate and Message_rate.

Maximum_message_rate is a 16 bit integer and represents an estimatation of the maximum sustained throughput (measured in messages per second) of the datalink controller type, and is unique only to that datalink. It is established during the datalink initialization phase and remains constant while that datalink is in use.

Current_Message_rate is a 16 bit integer representing the current receiver based congestion control policy for a datalink. It establishes the maximum number of RUN message segments per second that all remote nodes may offer to the receiver.

Message_rate is a 16 bit integer representing the maximum number of RUN message segments per second a single remote node may offer to the receiver. A Message_rate can vary from 0 to 32767 messages per second.

A Message_rate is associated with each datalink available to the transport. Different Message_rates may be concurrently enforced based on congestion of the various datalinks involved.

### 8.1.1 CONGESTION DETECTION

Congestion occurs when a datalink is unable to receive all messages destined to its address (see also DNA Line Counter; System Buffer Unavailable). When congestion occurs, the implemention will voluntarily restrict its utilization of the datalink by enforcing a Message_rate for that datalink to all connected nodes. This Message_rate will restrict the number of RUN message segments per second that may be transmitted to the datalink enforcing the congestion policy, but will not effect other datalinks available to the receiver or other message types.

The transport will periodically (no more than once per second) perform a datalink congestion check. This check will determine whether a datalink experienced congestion during the previous interval.

If congestion was detected, the congestion control algorithm will be enabled as described below.

A remote node will detect a datalink congestion policy in normal traffic with that node. The VCH.SB field will be set to one and the VCH.RATE_VALUE field will contain the number of RUN message segments per second that may transmitted to the datalink enforcing the congestion control. The congested datalink will be determined by the source address of the message containing the new Message_rate.

Only RUN message are affected by congestion control policies. All other messages types may be exchanged without regard to any congestion control policy in effect.

### 8.1.2 CONGESTION CONTROL ALGORITHM

Initially, a datalink is assumed to be not congested. When a datalink is not congested, the receiver will enforce a Message_rate of zero. A Message_rate of zero is architecturally defined as a no−congestion policy.

When a transport initializes, it will compute two congestion variables; Dll_maximum_rate and Current_message_rate. The Dll_maximum_rate is computed by multipling the Maximum_message_rate by 3. The Current_message_rate is initially set to the Maximum_message_rate and is modified as described below.

When congestion is detected, the transport will halve the Current_message_rate and recompute a Message_rate. This new Message_rate will be copied to the VCH.RATE_VALUE field and the VCH.SB bit set to one for all traffic over that datalink. When the rate has successfully be comunicated to the remote node (VCH.LAST_RATE_VALUE equals VCH.RATE_VALUE), the VCH.SB bit will be cleared until a new Message_rate is established.

If congestion persists, the Current_message_rate will again be halved and a new Message_rate communicated in normal traffic as decribed above. This process will continue until the Message_rate equals a minimum value of 1 or until congestion is no longer detected.

If congestion is not detected during the periodic check, the Current_message_rate will be incremented by 10 messages per second and a new Message_rate will be established and communicated in normal traffic as described above.

When the Current_message_rate equals the Dll_maximum_rate, the congestion algorithm will be disabled and a Message_rate of zero will be communicated to in normal traffic as decribed above.

Detection of a congestion control policy should be implemented in such a way as to introduce little per message overhead when a Message_rate is not being set. This is architecturally facilitated by using the sign bit of the word containing the Message_rate being set. If the sign bit is not set, the recevier may summarily ignore the Message_rate specified in the VCH.RATE_VALUE, regardless of its absolute value.

A transmitter will always set the VCH.LAST_RATE_VALUE to the Message_rate last established for the remote nodes datalink. A receiver will always check that the VCH.LAST_RATE_VALUE equals the Message_rate currently in effect. If the fields are not equal, the transport enforcing the Message_rate will cause the correct rate to be communicated in return traffic by setting the VCH.SB field to one and the VCH.RATE_VALUE to the applicable Message_rate.

### 8.1.3 CIRCUIT SUBLAYER DETAILS

When a receiver detects that a Message_rate is being enforced by the remote datalink, it will immediatly compute two variables; Remote_message_rate and Remote_rate_left for the datalink enforcing the Message_rate. The Remote_message_rate will be set to the VCH.RATE_VALUE. The Remote_rate_left will initially be set to the Remote_message_rate and will be modified as described below.

As decribed in the section title Transport Circuit Sub−layer, all message traffic is multiplexed through the circuit entity. It is at this level that the congestion policy is implemented. When higher layers attempt to transmit RUN messages to a remote node, the circuit sub−layer will first select a remote datalink to transmit the message to. If the remote datalink is enforcing a Message_rate, the circuit layer will check that Remote_rate_left for that datalink is not zero. If a positive rate exists, Remote_rate_left will be decremented and the message will be transmitted. If Remote_rate_left equals zero, another remote datalink will be selected and if congested its Remote_rate_left field checked until a datalink is selected with a positive rate. If no remote datalink has a positive Remote_rate_left, the circuit sub−layer will locally buffer the message.

Once per second, the circuit sub−layer will set Remote_rate_left to Remote_message_rate for each congested remote datalink. At this time, it will attempt to transmit all locally buffered messages as described above.

## 8.2 ASSOCIATION RATE BASED TRANSACTION FLOW CONTROL

The TRANS_TIMER field of each RESPONSE_DATA and RESPONSE_RESYNC is copied into the ACB_TRANS_TIMER field and establishes the delay between COMMAND_DATA streams.

When the server user aborts transactions due to resource errors, the server might increase the TRANS_TIMER issued to the association client to prevent future resource errors.

# CHAPTER 9

# CHECKSUM ALGORITHM

For all messages, checksumming is controlled by the checksumming flag contained in the virtual circuit header. Notice that even directory service messages contain the checksumming flag.

The option to require checksumming under circuit control is negotiated at circuit startup. If either end of the circuit requires checksumming (see START/STACK message format), both are obligated to generate checksummed messages and to verify received message checksum. This is designed to protect user data when communication equipment is known to be faulty. Note that every message provided in the LAST Architecture can be checksummed.

The following code fragment shows an example of a correct implementation of the calculation of a checksum. As checksum calculation on large packets is a lengthy process, care should be used in each implementation to match hardware capabilities. The algorithm presented is used in the VAX/VMS implementation, and is designed to take advantage of certain VAX processor instruction pre−fetch implementations. It is designed to checksum groups of eight longwords. The $DISPATCH macro used is a commonly used macro that generates a CASEx instruction.

```
        MOVAL   LAST_MESSAGE, R0        ; Get begin address of message
        MOVZWL  LAST_MSG_LEN(R0), R4    ; Get byte count
        ASHL    #−2, R4,R4              ; Make it a longword count
        BICL3   #^C^B111, R4, −(SP)     ; Compute entry into checksum loop
        ASHL    #−3,R4,R4               ; Remaining group of 8 longwords
        CLRL    R1                      ; Init accumulator

        $DISPATCH       (SP)+, TYPE=L,− ; Dispatch into summing vector
        <       <0              90$>,−
                <1              91$>,−
                <2              92$>,−
                <3              93$>,−
                <4              94$>,−
                <5              95$>,−
                <6              96$>,−
                <7              97$>>

; The following is the summing vector.  R4 contains the count of
; Eight−longword groups to be checksummed.  Entry into the table
; is based on the remainder of ((:MESSAGE_LENGTH/4)/8

98$:    ADDL    (R0)+,R1                ; Accumulate sum
97$:    ADDL    (R0)+,R1                ; Accumulate sum
96$:    ADDL    (R0)+,R1                ; Accumulate sum
95$:    ADDL    (R0)+,R1                ; Accumulate sum
94$:    ADDL    (R0)+,R1                ; Accumulate sum
93$:    ADDL    (R0)+,R1                ; Accumulate sum
92$:    ADDL    (R0)+,R1                ; Accumulate sum
91$:    ADDL    (R0)+,R1                ; Accumulate sum
90$:    SOBGEQ  R4,98$                  ; Loop
        MOVL    R1, (R0)                ; store checksum at end of
                                        ;   message stream
```

Checksums are calculated as a cumulative total of 32−bit unsigned integer values; carries are dropped. The 32−bit values start with the low−order bit of the virtual circuit message header MESSAGE_LENGTH field, which is added to each successive 32−bit value for (MESSAGE_LENGTH+3)/4 32−bit integer values. The actual checksum itself is a 32−bit unsigned integer value which is located at offset MESSAGE_LENGTH + ((MESSAGE_LENGTH+3)/4).

# CHAPTER 10

# PROGRESS TIMER

The purpose of this timer is to recycle the resources bound to a circuit if the circuit quality should become unsatisfactory. The PROGRESS_TIMER is specified in seconds.

The progress timer will be negotiated in the start/stack message exchange. At this time, the master will specify a progress timer. The server will use this timer unless it cannot support the value. As a result, the progress timer specified in the STACK message may negotiate a higher value.

The master will compute a progress timer that is half the negotiated circuit PROGRESS_TIMER. When the master's progress timer expires, it will transmit a SOLICIT message which requires a SOLICIT_RESPONSE from the server if no other messages have been exchanged during the interval. If no response has been received for Round_trip_ message_delay time, this same algorithm will be repeated Circuit_transmit_limit times at which point a Circuit_fault will be declared.

When a progress timer expires on the server end, it will declare a Circuit_fault event.

# CHAPTER 11

# TRANSACTION RESOURCE RECLAMATION

As previously specified, the server will cache segments received out of order until the first transaction segment is received. At this time, the server user is notified, and all cached buffers are copied to the server supplied buffers.

When a transaction is initiated, the client will specify a Transaction_response_timer. If large multiple of this timer (at least ten times) should elapse without receiving the entire segment stream, the server may ignore the transaction and release the cached transaction segments to free local resources.

This will also be a problem if the message stream is using Datagram_mode and the first segment is not received. A similar policy should be implemented.

# CHAPTER 12

# PROTOCOL REVISION CONTROL

The advertising message contains four fields which are used to maintain interoperability of LAST implementations. These are the current, high and low protocol version fields and the ECO level field.

Each time an ECO which is compatible within an existing version is done, the ECO_LEVEL field should be incremented to reflect that this change has been made. ECOs are always compatible with all messages for any given protocol version. ECOs that cause incompatibility must generate a new protocol version.

When a new version of the LAST architecture is created, these fields can be used to "talk down" to older version of the protocol by sending advertisement messages with the CUR_VERSION field set to the old protocol version value and a second message with the CUR_VERSION field set to the new protocol version field.

# CHAPTER 13

# MESSAGE FORMATS

Bits are transmitted onto the Ethernet low order bit first. When fields are concatenated, the right hand field is transmitted first. Numeric fields more that 8−bits long are transmitted least significant byte first.

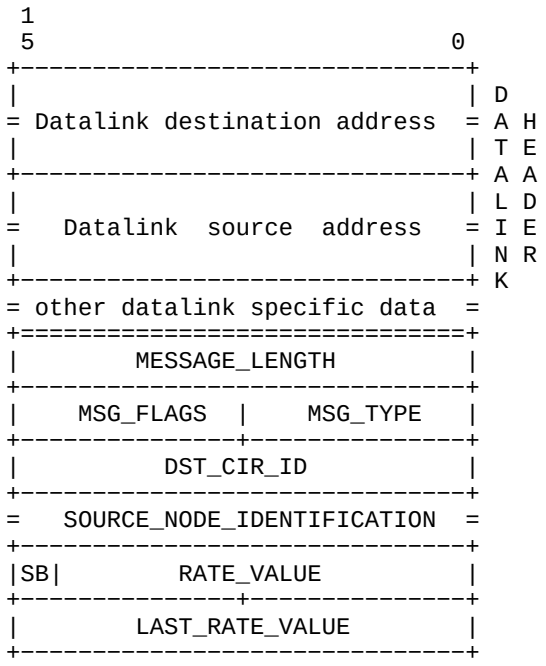
Fields are represented as bit streams, right to left. All fields are an integer multiple of eight bits. The symbol "=" is used to indicate fields of varying or indeterminate length.

LAST messages must be padded to the Ethernet 64 byte minimum. The data used to pad the frame to achieve this 64−byte minimum are unpredictable. In addition, data link padding, if available, is not used in this architecture.

Fields labeled MBZ must be zeroed on transmit and ignored on receive.

## 13.1 VIRTUAL CIRCUIT MESSAGE HEADER

All messages have the same header format (including the directory messages):



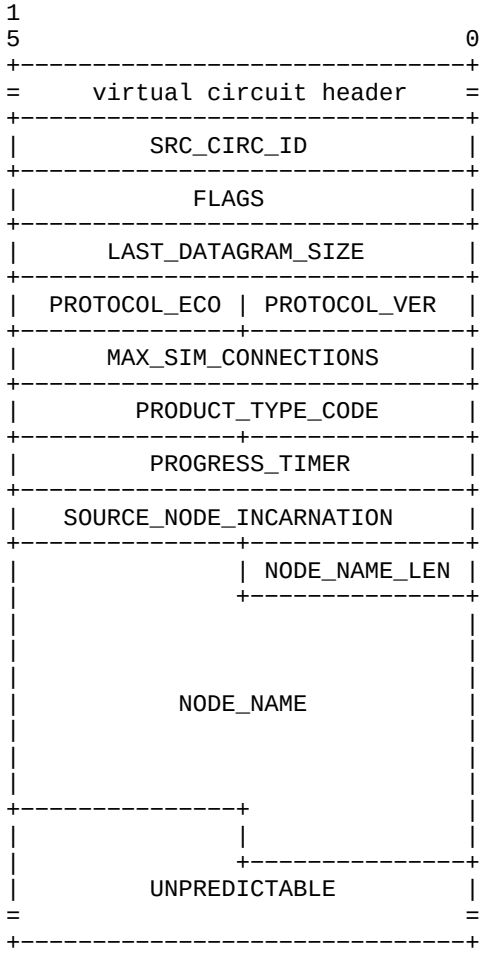
- MESSAGE_LENGTH (2 bytes unsigned) − Length of the message to follow.
- MSG_TYPE (1 byte) − the message type field:
  - RUN message have this field set to 0.
  - START message have this field set to 1.
  - STACK message have this field set to 2.
  - STOP message have this field set to 3.
  - LOOP messages have this field set to 4.
  - ADVERTISEMENT messages have this field set to 5.
  - SOLICIT messages have this field set to 6.
  - SOLICIT_RESPONSE messages have this field set to 7.
- MSG_FLAGS − The following bits are valid for flags.
  - — Bit 0 = 1. The data, beginning at MESSAGE_LENGTH, through MESSAGE_LENGTH bytes, will be checksummed. The checksum will follow the message (after rounding up modulus 4), and is not included in the MESSAGE_LENGTH count. The checksum is a 4 byte, unsigned value.
  - — Bits 1 through 7 are MBZ.
- DST_CIR_ID (2 bytes)− A reference to the destination node CSB.
- SOURCE_NODE_IDENTIFICATION (6 bytes) − A unique identification of the source node which remains constant across transport incarnations.
- SB (1 bit) − When set, indicates that RATE_VALUE is being established.
- RATE_VALUE (15 bit unsigned − A segment per second count
- LAST_RATE_VALUE (16 bits unsigned) − A segment per second count.

**NOTE**

**The LAN header is typically supplied by the datalink driver software, and is subject to change in newer LAN message formats. As a result, it is not included in a LAST implementation. It is assumed that the message header minimally provides the source address of the transmitter, and is addressable by this transport.**

### 13.1.1 START/STACK MESSAGE FORMAT

START message headers have MSG_TYPE fixed at 1. STACK message have this value set 2. Otherwise, the format and meaning of the values in the START and STACK message are equivalent and symmetric.



- SRC_CIRC_ID (2 bytes) − A reference to the source node CSB, to be used by the circuit partner when generating circuit based messages.
- FLAGS (16 bits) − Flags used to negotiate circuit level features.
  - — Bit 0 − INTRA_BURST_DELAY is a bit flag which indicates how the transmitter should pipeline transmit message segments. If set to 1, the transmitter should transmit message segments one at a time, waiting for a transmit complete event for each message segment. If set to 0, the transmitter may pipeline Burst_size message segments. Notice that this flag is not negotiable and affect all circuit based transmit operations for node receiving this message.
  - — Bit 1 − FORCE_CHECKSUMMING Either end of a virtual circuit may require data checksumming. The presence of this bit in either the START or the STACK message will force checksumming verification.
  - — Bits 2−15 are MBZ.
- LAST_DATAGRAM_SIZE (2 bytes) − The minimum and maximum sizes of frames are restricted by each datalink implementation. An implementation must specify the maximum size datagram size that it supports. The circuit master will set its datagram size in the START message. The circuit slave will not offer larger messages for transmission to the circuit master than is specified in the START message. The circuit slave might specify a different value in the STACK message, which in turn limits the size of datagram offered to the datalink layer by the circuit master.
- PROTOCOL_VER (1 byte) − The protocol version of this message and of all messages transmitted on this circuit. An exact match of protocol version must exist before the circuit can be created. If a protocol mismatch exists, the circuit layer should ignore the message. The START message protocol version should be selected by utilization information supplied through the directory service messages.
- PROTOCOL_ECO (1 byte) − The protocol version ECO (Engineering Change Order) of this message and of all messages transmitted during this circuit. For any given protocol version, ECOs are backward compatible. The PROTOCOL_ECO level is intended to be used to reflect patches made in the field by automatic updates or by field software specialists.

- MAX_SIM_CONNECTS (2 bytes unsigned) − maximum number of simultaneous associations that can be opened on this virtual circuit. The master offers a number, and the slave may negotiate a lower value if it cannot support the value suggested by the master. The value returned in the STACK message will be the maximum number of associations supported over this circuit. This value corresponds to the ACB vector length in the CSB.
- PRODUCT_TYPE_CODE (2 byte unsigned) − The follow type codes are currently assigned (a value of zero is used as an unspecified product code):

  1. Digital VAX/VMS client and server
  2. Digital VAX/VMS client
  3. Digital VAX/VMS server
  4. Digital MS/DOS client and server
  5. Digital MS/DOS client
  6. Digital MS/DOS server
  7. Digital OS/2 client and server
  8. Digital OS/2 client
  9. Digital OS/2 server

  The term "client" identifies the transport as supporting client associations, while the term "client and server" identifies the transport as supporting client associations, server associations, or both.
- PROGRESS_TIMER (2 unsigned bytes) − A timer, specified in seconds. This value is negotiated the same way as the MAX_SIM_CONNECTS.
- SOURCE_NODE_INCARNATION (2 Bytes) − The unique value assigned at transport initialization. This value must be different from any recent value used by a previous incarnation of this transport.
- NODE_NAME_LEN (1 byte unsigned) − Byte count of NODE_NAME field.
- NODE_NAME (16 bytes, fixed length) − The text within this field should describe the name of the master or slave node.

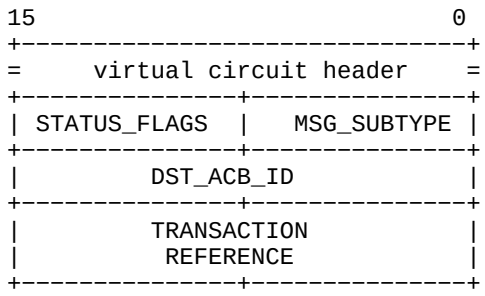
### 13.1.2 RUN MESSAGE FORMAT

RUN messages have MSG_TYPE set to 0. RUN messages are exchanged once the START/STACK message exchange is completed and continue until a STOP message deletes the virtual circuit state.

A Transaction Reference is assigned by the Master. It is maintained in all RUN messages and is not used by the LAST transport architecture. LAST server transports will copy the transaction reference value from a request message to a response message and to the XCB_TRANSACTION_ID.

RUN messages sub−types are COMMAND_CONNECT, RESPONSE_CONNECT, COMMAND_DATA, RESPONSE_DATA, COMMAND_RESYNC, RESPONSE_RESYNC, COMMAND_DISCONNECT, RESPONSE_DISCONNECT.

RUN messages have the following common format:

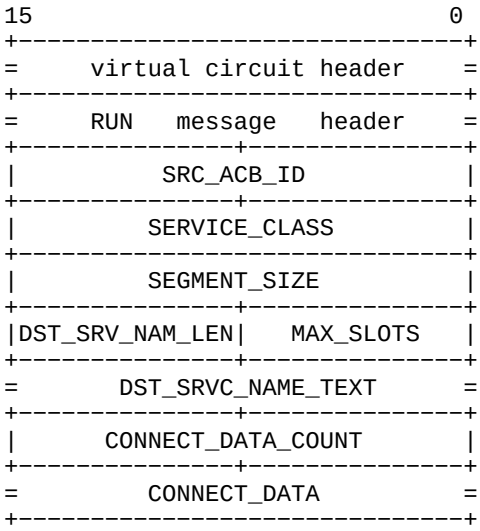



- MSG_SUBTYPE (8 bits unsigned) − the value for this message type the following message subtypes are defined:
  - — COMMAND_DATA messages have a value 0
  - — RESPONSE_DATA messages have a value 1
  - — COMMAND_CONNECT messages have a value 2
  - — RESPONSE_CONNECT messages have a value 3
  - — COMMAND_RESYNC messages have a value 4
  - — RESPONSE_RESYNC messages have a value 5
  - — COMMAND_DISCONNECT messages have a value 6
  - — RESPONSE_DISCONNECT messages have a value 8

- STATUS_FLAGS (8 bits) − Association modifiers:
  - bit 0−1 (2 bit unsigned integer) − Mode indicator.
    - Mode 0 is Idempotent_mode: the slave may discard transaction responses immediately, without waiting for acknowledgement.
    - Mode 1 is Timed_mode (partially described): In this mode, the slave may discard transaction responses after a few milliseconds.
    - Mode 2 is Normal_mode (partially described): In this mode the slave may discard transaction responses only after they are explicitly acknowledged.
    - Mode 3 is datagram_mode − In this mode, no command/response matching is done.
  - bits 3−7 − These bits are MBZ.
- DST_ACB_ID (2 bytes unsigned) − a handle on (an index to) the remote ACB. This value must be zero for a connect request message, and must not be zero for any other RUN message.
- TRANSACTION_REFERENCE (4 bytes) − A client specified value. This value is supplied by the client on all RUN messages. It must be copied from each request message to the response message for every message received by the slave transport. Client transports will verify that the TRANSACTION_REFERENCE issued on the request message equals the TRANSACTION_REFERENCE on the response message. If it is not, the client is to ignore the response message.

#### 13.1.2.1 COMMAND_CONNECT MESSAGE

If a COMMAND_CONNECT is received, the format of the message is:




- SRC_ACB_ID (2 bytes unsigned) − a handle on (an index to) the local client association control block.
- SEGMENT_SIZE (2 bytes unsigned) − The segment size used to fragment transaction requests and represents the amount of user data transferred in a COMMAND_DATA/RESPONSE_DATA message. A segment size is computed using the following formula:

  SS = DS − RH − 7

  Where:

  SS = Segment Size

  DS = Datagram size (from START message)

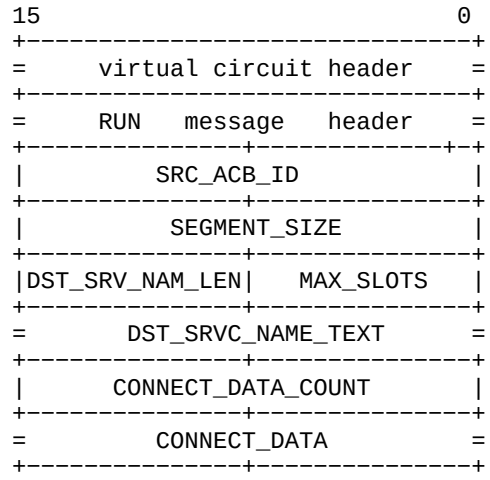
  RH = RUN request header size (from RUN.MESSAGE_LENGTH

  through RUN.DATA_COUNT).

  7 = Allowance for data checksumming and rounding.
- MAX_SLOTS (1 byte unsigned) − The maximum number of transaction that can be pipelined by a client. This number represents the maximum number of transactions that are concurrently supported by this client. If this value exceeds that supported by the server, it will replace its value in the RESPONSE_CONNECT message. The value associated with the RESPONSE_CONNECT will be the maximum number of concurrent transactions available for this association.
- REQ_SERVICE_NAME_LEN (1 byte unsigned) − The number of characters in the REQ_SERVICE_NAME_ TEXT field.
- REQ_SERVICE_NAME_TEXT (SERVICE_NAME_LEN bytes) − The destination service name used in form− ing the connection.
- DATA_COUNT (2 bytes) − Byte count of application data.
- CONNECT_DATA − DATA_COUNT bytes of application data.

#### 13.1.2.2 RESPONSE_CONNECT MESSAGE

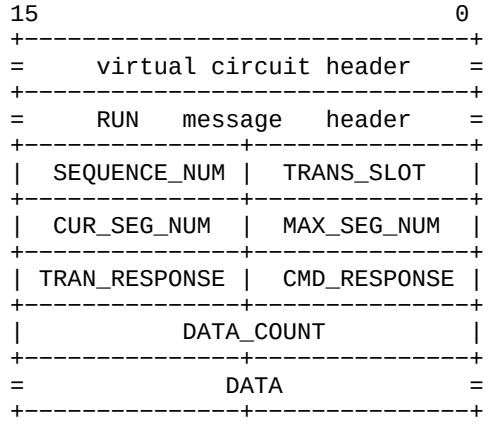
If a RESPONSE_CONNECT is received, the format of the message is:




- SRC_ACB_ID (2 bytes unsigned) − a handle on (an index to) the Slave Connect block.
- SEGMENT_SIZE (2 Bytes) − The maximum amount of data contained in a RUN request message.
- MAX_SLOTS (1 byte unsigned) − The maximum number of transaction that can be pipelined by the server. This number represents the maximum number of transactions that can be simultaneously supported by the server and may be different from the client value. If the Server software is unable to support the client specified number of slots, it will change it to a value that it can support. This value then becomes the SSB_ MAX_XID_SLOTS value.
- DST_SERVICE_NAME_LEN (1 byte unsigned) − The number of characters in the DST_SERVICE_NAME_ TEXT field.
- DST_SERVICE_NAME_TEXT (SERVICE_NAME_LEN bytes) − The destination service name used in form− ing the connection.
- DATA_COUNT. Byte count of application data.
- DATA. Application data

#### 13.1.2.3 COMMAND_DATA MESSAGE

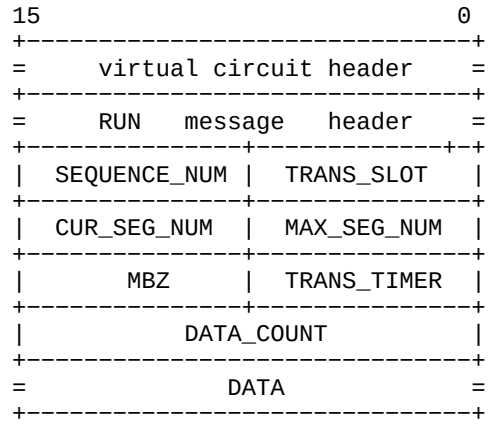
If a COMMAND_DATA is received, the format of the message is:




- TRANS_SLOT (1 byte) − Multiple transactions may be pipelined through a association. This value provides a handle on (and index to) the current transaction slot for this message.
- SEQUENCE_NUM (1 byte) − The unique sequence number associated with this message for this transaction id. This field is incremented for each new transaction within a slot.
- MAX_SEG_NUM (1 byte unsigned) − Used to identify the number of segments in this transaction command. May not be zero.
- CUR_SEG_NUM (1 byte unsigned) − Used to identify this segment number. May not by zero.
- CMD_RESPONSE (1 byte unsigned) − The Command_response_timer.
- TRAN_RESPONSE (1 byte unsigned) − The Transaction_response_timer.
- DATA_COUNT (2 bytes unsigned) − The count in bytes of the next field.
- DATA (DATA_COUNT bytes) − The user supplied data.

#### 13.1.2.4 RESPONSE_DATA MESSAGE

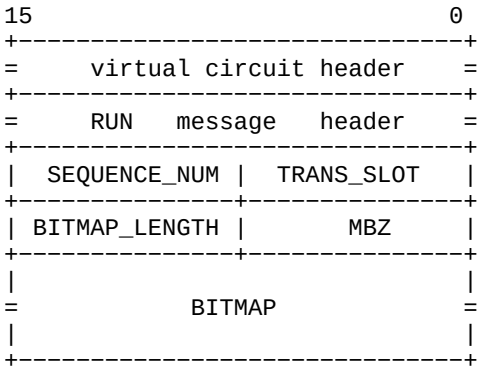
If a RESPONSE_DATA is received, the format of the message is:




- TRANS_SLOT (1 byte) − Multiple associations may be pipelined through a association. This value provides a handle on (and index to) the current transaction id for this message.
- SEQUENCE_NUM (1 byte) − The unique sequence number associated with this message for this transaction id. This field is incremented for each unique transaction. Value is copied from the COMMAND_DATA message.
- MAX_SEG_NUM (1 byte unsigned) − Used to identify the number of segments in this transaction response. May not be zero.
- CUR_SEG_NUM (1 byte unsigned) − Used to identify this segment number. May not by zero.
- TRANS_TIMER(1 byte unsigned) − An unsigned byte. The units are 10 milliseconds intervals.
- MBZ (1 byte)
- DATA_COUNT (2 bytes unsigned) − The count in bytes of the next field.
- DATA (DATA_COUNT bytes) − The user supplied data.

#### 13.1.2.5 COMMAND_RESYNC MESSAGE

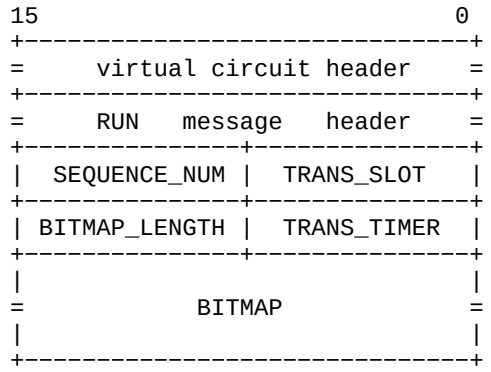
If a COMMAND_RESYNC is received, the format of the message is:




- TRANS_SLOT (1 byte) − Multiple associations may be pipelined through an association. This value provides a handle on (and index to) the current transaction id for this message.
- SEQUENCE_NUM (1 byte) − The unique sequence number associated with this message for this transaction id. This field is incremented for each unique transaction.
- MBZ (1 byte) − Must be zero.
- BITMAP_LENGTH (1 byte signed) − A byte which indicated the length of the following field.
- BITMAP (variable length) − A bitmap of transaction segments.

#### 13.1.2.6 RESPONSE_RESYNC MESSAGE

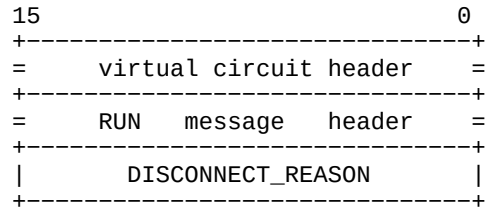
If a RESPONSE_RESYNC is received, the format of the message is:




- TRANS_SLOT (1 byte) − Multiple associations may be pipelined through a association. This value provides a handle on (and index to) the current transaction id for this message.
- SEQUENCE_NUM (1 byte) − The unique sequence number associated with this message for this transaction id. This field is incremented for each unique transaction.
- TRANS_TIMER (1 byte unsigned) − An unsigned byte. The units are 10 millisecond intervals.
- BITMAP_LENGTH (1 byte signed) − A byte which indicated the length of the following field.
- BITMAP (variable length) − A bitmap of transaction segments.

#### 13.1.2.7 COMMAND_DISCONNECT MESSAGE

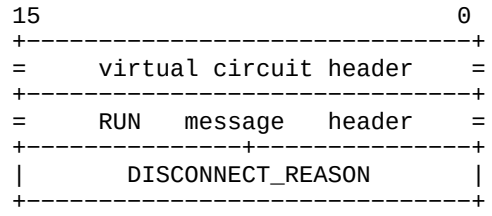
If a COMMAND_DISCONNECT is received, the format of the message is:




- DISCONNECT_REASON (2 bytes unsigned) − The disconnect reason. A reason of 0 is undefined.
  1. invalid message format received
  2. invalid local connection identification received
  3. invalid remote connection identification received
  4. Association_halt from client
  5. no progress is being made
  6. time limit expired
  7. retransmit limit reached
  8. no resources
  9. Service name has changed
  10. (make up your own reasons, but please get them added to this document)

#### 13.1.2.8 RESPONSE_DISCONNECT MESSAGE

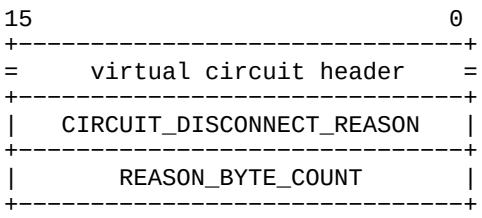
If a RESPONSE_DISCONNECT is received, the format of the message is:




- DISCONNECT_REASON (2 bytes unsigned) − The disconnect reason.
  1. invalid message format received
  2. invalid local connection identification received
  3. invalid remote connection identification received
  4. Association_halt from client
  5. no progress is being made
  6. time limit expired
  7. retransmit limit reached
  8. no resources
  9. Service name has changed
  10. Server reject connect
  11. (make up your own reasons, but please get them added to this document)

### 13.1.3 STOP MESSAGE FORMAT

STOP message headers have the following format:

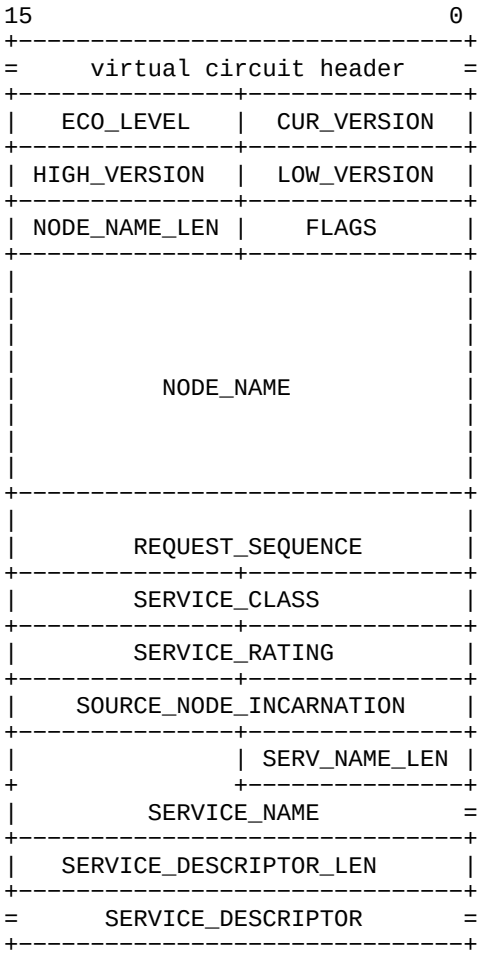



- CIRCUIT_DISCONNECT_REASON (2 bytes unsigned) − A zero value means no reason is given. The currently defined reasons are:

  1. Invalid response timer
  2. VC stop from user
  3. Protocol error
  4. No resources
  5. (make up your own reasons, but please get them added to this document)

### 13.1.4 ADVERTISEMENT, SOLICIT AND SOLICIT_RESPONSE MESSAGE FORMATS

ADVERTISEMENT, SOLICIT and SOLICIT_RESPONSE message formats are:



- CUR_VERSION (1 byte unsigned) − The protocol version of this message. This node supports messages exchanges using this protocol version.
- ECO_LEVEL (1 byte unsigned) − The engineering change order of this message and for this protocol version.
- LOW_VERSION (1 byte unsigned) − The lowest version supported by this node.
- HIGH_VERSION (1 byte unsigned) − The highest version supported by this node.
- FLAGS (1 byte unsigned) − Bitfield describing the transmitters transport type and flags
  - Bit 0 = 1 (MASTER_FLAG) if transport type is Master.
  - Bit 1 = 1 (SLAVE_FLAG) if transport type is Slave.

    Note that both bits may be set in a message. Transports may use this field to determine if the transmitter is a node that it wants to maintaining in their topology. This field is required on all advertisement messages.
  - Bit 2 = 1 (PURGE_ALL_PATHS_FLAG) All known path data should be purged for the SOURCE_ NODE_IDENTIFICATION node (except the one the message came in on)
  - Bits 3 through 7 are MBZ
- NODE_NAME_LEN (1 byte unsigned) − In the range 1 − 16. This is the length of the transport node name. This field is required for all ADVERTISEMENT messages.
- NODE_NAME (character 16 bytes) − The name associated with the transmitting transport. This field is required for all ADVERTISEMENT messages.
- REQUEST_SEQUENCE (4 bytes) − This field is reserved for use by the transmitter and must be copied to any response message.
- SERVICE_CLASS (2 bytes unsigned) − A unique identifier indicating the service type. Masters and servers are registered with the transport and are assigned service class values. If :SOLICIT.SERVICE_CLASS is zero, then a SOLICIT_RESPONSE message is required.
- SERVICE_RATING (2 bytes) − An indication of service quality.
- SOURCE_NODE_INCARNATION (2 Bytes) − The unique value assigned at transport initialization. This value must be different from any recent value used by a previous incarnation of this transport.
- SERVICE_NAME_LEN (1 byte unsigned) − This filed is valid for all directory service messages. If :SER−VICE_NAME_LEN is zero, the message is interpreted as a topology message. If the SERVICE_NAME_LEN is non−zero, it is interpreted Service related.
- SERVICE_NAME (character, limited only by message size) − The name of the service of interest. The SERVICE_CLASS user is free to construct the service name to suit their application.
- SERVICE_DESCRIPTOR_LEN (2 bytes unsigned) − The length of the following SERVICE_DESCRIPTOR.
- SERVICE_DESCRIPTOR (size limited by Last_datragram_size) − Data associated with the service name described above.

# CHAPTER 14

# DEFINED AND RECOMMENDED CONSTANTS

The following values are architecturally defined constants:

- Protocol Type − 8041 (hexadecimal) or as a bit stream, first bit on LAN at left (1000.0000.0100.0001)
- Multicast Address Range − 09−00−2B−04−00−00 thru 09−00−2B−04−03−FF (hexadecimal).

The following values must be specified before an implementation is operational. The architecture requires the values be within the ranges specified:

- CIRCUIT_TIMER − In the range 10−900 milliseconds (80 milliseconds recommended) for the master. In the range 1 to 2 seconds for the slave.
- CIRCUIT_RETRANSMIT_LIMIT
- TRANSACTION_TRANSMIT_LIMIT − In the range 2−20 transactions (5 transactions recommended) for the master.
- ADVERTISEMENT_INTERVAL − In the range 10 to infinite seconds (120 seconds recommended). This value can be supplied via the start_service_class function.
- LAST_MIN_RCV_DATAGRAM_SIZE − The datalink must supply maximum size buffers.
- NBR_DL_BUFFERS − The number of data link buffers assigned minus one. A value of 20 is recommended.
- TRANSACTION_COMMAND_TIMER − Client delay before retransmitting a request in the range of 3 to 255 (3 recommended).
- CUR_VERSION − Set to 2
- ECO_LEVEL − Set to 0.
- LOW_VERSION − Set to 2.
- HIGH_VERSION − Set to 2.
- PROGRESS_TIMER − In the range 100 to 65,000 seconds.
- Transaction_response_timer − In the range 3 − 255 seconds.
- Command_response_timer − 2 seconds or less
- Path_quality_threshold − In the range of 2 to 5 Command_failure events.
- Round_trip_message_delay − Implementation specific. Should be independently measured.
- Transaction_transmit_limit − Specified by Client user.
- Maximum_message_rate − A datalink specific value representing an estimation of the datalinks maximum throughput in messages per second.
- Current_message_rate − A datalink message per second rate currently in effect for all remote nodes.

- Message_rate − A datalink message per second rate currently in force for a given remote node.
- Dll_maximum_rate − Three times the Maximum_message_rate for the given datalink.
- Remote_message_rate − The Message_rate in effect for a remote datalink
- Remote_rate_left − The amount of Remote_message_rate still available for the remote node.

[End of document]