LASTport

Architecture Specification

Order Number: EK–LASTP–AS–001

March 1992

This document specifies the interfaces and protocol for implementing the LASTport architecture, a computer networking transport layer model. By limiting the services that the transport layer offers to the session layer, a LASTport implementation can enhance a system application's performance and reduce its resource consumption.

Revision Update Information: This is a new document.

Software Version: LASTport Version 3.1

Digital Equipment Corporation
Maynard, Massachusetts

March 1992

This document was prepared using VAX DOCUMENT, Version 2.0.

Contents

3 Solicitation Operations

4 Circuit Operations

5 Association Operations

6 Message Formats

A Checksumming Algorithm

Glossary

Index

Examples

Figures

Tables

Preface

This document describes the LASTport architecture, a model of the software that controls the transport of messages in a local area network (LAN).

The goal of this document is to specify the LASTport protocol syntax and semantics in sufficient detail to facilitate interoperable implementations. The document does not describe implementation–specific data structures and interfaces.

Document Structure

This document contains six chapters and an appendix:

- Chapter 1 explains how the LASTport protocol differs from other transport protocols and provides an overview of the LASTport architecture and its environment.
- Chapter 2 describes how the LASTport protocol performs and controls transactions.
- Chapter 3 describes Solicitation layer functions.
- Chapter 4 describes Circuit layer functions.
- Chapter 5 describes Association layer functions.
- Chapter 6 describes LASTport messages.
- Appendix A describes the LASTport checksumming algorithm.

A glossary defines terms used in the document.

Associated Documents

For information on porting the architecture to a C–language environment, refer to the *Portable LASTport Interface Specification* document. For information on the LASTport/Disk architecture, refer to the *LASTport/Disk Architecture Specification* document.

Intended Audience

The audience for this document includes implementers of the architecture and programmers who intend to port the architecture to various environments. Readers should be familiar with the functions and terminology of network transport layers.

Conventions

Table 1 lists typographic conventions used in this document. Table 2 describes architecture format types. All numeric values are decimal unless specified otherwise. All format types are defined with the right most bit as the least significant bit of the byte.

Table 1 Typographic Conventions

Convention	Meaning
Initial Caps	Names of messages, such as Solicit Request, or names of achitecturally controlled variables, such as Service Name.
italics	Names of events, such as *AppConnect* or *AsnTransComplete*, that are generated or processed by LASTport and LASTport/Disk clients and severs, or user applications.
boldface	New terms or terms defined in the Glossary.
UPPERCASE	Names of messsage fields, such as CUR_PRTCL_VER or STATUS.
MixedCase	Names of processes, such as AsnClientRcv; names of routines and variables in process pseudocode examples, such as GetNextService or dataRequestLength; names of timers, such as SolicitResponseTimer or AsnConnectResponseTimer.
TAZIOR	Transmit As Binary Zeros and Ignore On Receipt. Specifying TAZIOR in Message Format fields allows Engineering Change Order (ECO) extensions to the protocol without changing major version numbers.

Table 2 Architecture Format Types

Type	Meaning
Bit mask	A variable-length field (in bits), in which each bit has an assigned meaning. Bit value 1 indicates “true” or “do” relative to the assigned bit meaning.
Byte (8 bits) Word (16 bits) Longword (32 bits)	These fields describe either signed or unsigned integers. Unsigned integers can range in value from 0 to 255 (bytes), 65,535 (words) and 2**32–1 (longwords). Signed integers are specified in two's complement form; bytes range from –128 to 127, words from –32,768 to 32,767 and longwords from –2**31 to 2*31–1.
Name	Unsigned byte-counted string containing the characters described in the *Local Area Disk Architecture Specification* document.
Text string	Unsigned byte-counted string containing the characters described in the *LASTport/Disk* document.

1
LASTport Concepts

On paper, transport **protocols** look much alike. In practice, however, a specialized protocol can enhance the efficiency of **system applications** by taking advantage of their message flow characteristics. The LASTport protocol is designed specifically to support applications that use **request–response semantics**.

Rather than attempting to achieve acceptable **performance** for all possible **transactions**, the LASTport protocol is designed to optimize transaction processing in one particular environment: a **local area network (LAN)** that includes many **clients** and only a few **servers**. Furthermore, only naturally **idempotent procedures**, such as disk reads, access to read-only data in general, and name translation, can use the protocol directly.

The LASTport protocol is not intended to replace a traditional transport model like DECnet Network Services Protocol (NSP) or Open Systems Interconnect (OSI) TP4. Instead, the LASTport protocol is designed to move block data between memories. For example, it operates as an efficient transport for a virtual disk service.

In the LAN environment for which it is designed, the LASTport protocol offers the following features:

- Specialized request–response semantics with minimal latency
- Reduced message flow over the interconnect
- Efficient encoding of **messages**
- Reduced memory consumption at the server system
- Support for large numbers of concurrent transactions
- High availability with redundant paths
- Effective error recovery

This chapter introduces the LASTport protocol and architecture and provides an overview of LASTport transaction management. Topics include

- LASTport and other transport protocols
- LASTport operating environment
- Functional layers
- Transaction management
- Concurrent transactions and transaction identifiers

1.1 LASTport and Other Transport Protocols

Transport protocols normally perform the following tasks:

- Create a channel that can detect and correct data loss and duplication
- Deliver data in order of transmission
- Govern the rate of data flow across a **circuit** and operate multiple circuits independently

These tasks must be performed efficiently and in a way that isolates the **Session layer** from changes in the hardware technology.

A LAN transport protocol must manage all types of data transfer that are possible in the environment. Typically, the **Transport layer** accepts data from the Session layer and then does the following:

1. Divides the data into smaller units, if necessary
2. Passes these units to the **Network layer**
3. Ensures that the units arrive correctly at the destination

The methods that the transport uses to ensure that the units arrive correctly are called **end-to-end guarantees**, and such guarantees are often made at the cost of performance.

Transport performance is usually measured by the following criteria:

- Response time per end-to-end communication
- Number of network messages required per transaction
- Throughput in bits per second
- Use of system compute power and memory
- Efficiency of encoding user data

The LASTport protocol differs significantly from other transport protocols in several ways. To clarify these differences, Sections 1.1.1 through 1.1.3 compare a traditional peer-to-peer transport protocol with the local area transport (LAT) protocol and the LASTport protocol in the following areas:

- The relationship of one **node** to another, such as peer to peer or client to server
- The number of transactions that can be conducted simultaneously on a single **connection**
- The order in which transactions complete

To facilitate comparison, all three transport systems are discussed in general terms. Actual implementations might not match the models described.

1.1.1 Traditional Peer-to-Peer Transport Protocol

In a traditional **peer-to-peer transport protocol** such as DECnet NSP, data exchanges between nodes operate symmetrically. Usually, only one transaction can be conducted at a time, and one transaction must complete before another can start. The protocol used at both nodes is identical. By contrast, in a **client–server protocol**, one node has priority over the other, and this difference is reflected in the way messages are processed.

In traditional transports, each **request** for **service** on the network requires that a circuit be established between nodes. One circuit is dedicated to one pair of communicating processes; multiple connections require multiple circuits. Each data exchange between the processes requires overhead processing to handle loss detection and error control. The transport orders messages before they are delivered to the **application**.

In these transports, a typical data exchange includes the following operations:

1. Node 1 sends a data request message to node 2.
2. Node 2 receives the data request and sends an acknowledgment message to node 1 indicating that node 2 received the request message.
3. Node 2's application processes the data from node 1 and transmits a response message to node 1.
4. Node 1 receives the response message and must send an acknowledgment message to node 2 indicating that the response message has been received.

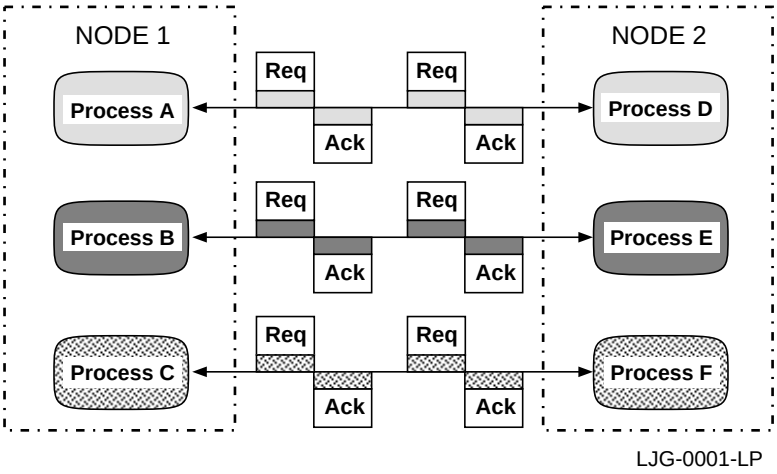
Half the messages in the data exchange are acknowledgments that do not carry transaction data; the acknowledgment messages are required for error recovery. While acknowledgment messages can often be combined with normal data messages, acknowledgment messages are always required to complete a transaction.

Note

In traditional peer-to-peer transports, receipt of an acknowledgment message in response to a request does not guarantee that data has been processed — only that the remote node received a request message. To ensure that the remote node processed the data, an additional protocol layer must usually be added for each application.

Figure 1–1 illustrates a traditional peer-to-peer transport with three separate circuits.

Figure 1–1 Traditional Transport with Three Circuits



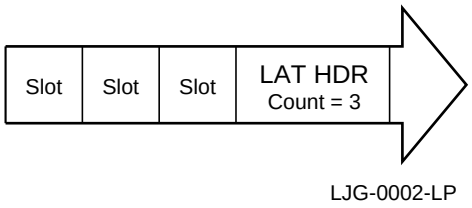
1.1.2 LAT Protocol

Unlike traditional transports, the **LAT protocol** implements node-to-node communication as a client–server relationship. However, the data is presented at the System Application layer as if the transactions are peer to peer. Several data transfers can be in progress at one time, and data is always delivered in the order it is sent over each connection.

Two other major differences between the LAT protocol and traditional transports are in the number of circuits created and the kind of packets sent.

In the LAT protocol, data transmission is based on timers. The first request for service on the network creates a new circuit between node 1 and node 2. If there are multiple requests for connections (sessions) between the same nodes, the LAT protocol combines messages from more than one session into single packets and sends them along a single node-to-node circuit. Using the timers, the LAT protocol delays transmitting data from individual sessions to accumulate the data into a single packet. As shown in Figure 1–2, a LAT packet is divided into **data slots**, which contain the data from each session.

Figure 1–2 LAT Packet



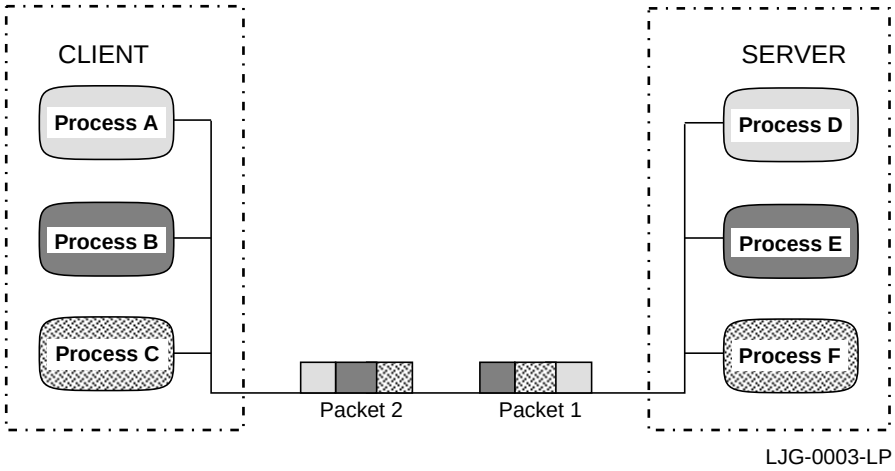
By using only one circuit and encoding data efficiently in packets, the LAT protocol reduces the need for circuit **path maintenance**. However, use of a single circuit has disadvantages. More latency occurs at each end as data accumulates to form a packet. In addition, fault tolerance is affected: if the circuit fails, all the overlying connections also fail, because they depend on that circuit.

A typical LAT data exchange still includes acknowledgment messages:

1. A timer event occurs, indicating that user data from all sessions should be sent from the terminal server client to the server.
2. The client transport builds a data packet with data loaded from each session.
3. The server transport receives the data packet and sends an acknowledgment message, which can contain response data, such as a character displayed from a keystroke.
4. The server transport responds with a "no data waiting" flag set.
5. The exchange ends if the client has no more data at the next timer event.

Figure 1–3 shows multiple processes using one LAT circuit.

Figure 1–3 Multiple Processes Using One LAT Circuit



1.1.3 LASTport Protocol

Like the LAT protocol, the LASTport protocol models communications from node to node as a client–server relationship. However, unlike the LAT protocol, the LASTport protocol distinguishes client and server up to the System Application layer and presents data to that layer as a client–server relationship. This design feature eliminates the need for acknowledgment messages, significantly reducing the number of messages needed to complete a transaction.

For example, a LAT client requires both a request acknowledgment message and a transaction response acknowledgment from the server. In contrast, a LASTport client does not require an acknowledgment message. The server's transaction response guarantees that the server received the request and processed the data successfully.

The LASTport protocol differs from the LAT protocol in other respects:

- The LASTport protocol can perform concurrent transactions independently. LASTport transactions can complete in an arbitrary order; that is, they are processed concurrently by the server system application and are returned to the client as they are completed.

- The LASTport protocol can concurrently use multiple paths to the destination node.

These differences result from certain assumptions, discussed in Section 1.2, about the LAN operating environment in which the LASTport protocol is used.

1.2 LASTport Operating Environment

The LASTport protocol is designed with the assumption that the underlying network (shared interconnect) is reliable. Transport performance depends on low-probability events (fewer than one in one thousand) occurring infrequently and very-low-probability events (fewer than one in one trillion) not occurring at all. This kind of **reliability** is available in a LAN environment with the following characteristics:

- Data Link capacity of more than 10,000 messages per second.
- Low probability of **datagram** loss or duplication. (The LASTport protocol includes a **checksum** feature.)
- Low probability of delays of more than 50 milliseconds between source and destination ports.
- Very low probability of of datagram corruption or of datagrams being delivered to the wrong destination address.

The LASTport protocol makes the following additional assumptions:

- The environment includes many clients and a small number of servers. Therefore, the LASTport protocol attempts to conserve server resources wherever possible.
- The clients and servers are system applications, not user-written applications. This approach is especially effective in an environment where clients access servers that are distributed on the network, because network-based servers can be implemented as "black box" **systems** without the typical overhead associated with crossing the user–system protection boundary.
- All transactions issued by the client are in the form of idempotent requests and are commutative relative to each other.
- The network is capable of handling multicast messages.

If these conditions exist in the environment, the LASTport protocol can enhance performance for certain types of applications, such as virtual disk services.

1.2.1 Protocol Advantages

In a LAN environment with the characteristics listed in Section 1.2, the LASTport protocol provides the following advantages:

- Reduced message flow over the interconnect
- Efficient encoding of messages
- Reduced memory consumption at the server system
- Support for large numbers of concurrent transactions
- High availability with redundant paths
- Effective error recovery

The following sections describe these advantages.

Reduced Message Flow over the Interconnect

Because the LASTport protocol is based on a client–server relationship, client and server operate using different protocols. Only the client can initiate a request, and the server must respond. In contrast, traditional transports do not require a server to respond to a request. One service call issues requests, and a separate service receives a response. Therefore, the relationship between nodes must be maintained by using acknowledgment messages. Some transports might use four messages to complete an exchange of data, as described in Section 1.1.1.

The LASTport protocol pairs data exchanges into transactions, sets of two messages resulting in a round-trip exchange of data. Because a message from a LASTport client requires a response, all messages occur in pairs. The first element is the client's request, and the second is the server's response to the request. The transport matches the request with the response as follows:

1. The client sends a data request.
2. The server sends a response.

Because the LASTport protocol requires that servers respond to client requests, no acknowledgment messages are necessary: the number of messages exchanged over the LAN interconnect can be reduced by 30 to 50 percent.

Efficient Encoding of Messages

The LASTport protocol handles requests and responses atomically. Client and server semantics allow for atomic processing of messages, because the beginning and end of requests and responses are made visible.

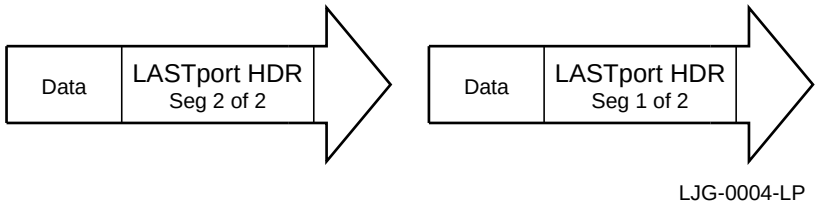
The system application is not bound to fixed-size data messages; it can send messages of almost any size. If required, the LASTport protocol segments the data into packets of the size that the Data Link layer supports. However, messages cannot be arbitrarily large. For instance, Ethernet LAN messages should not be larger than approximately 50 KB, because protocol efficiency would decrease dramatically. In general, messages should not be so large that a one-way burst could cause one or more packets to be lost.

Reduced Memory Consumption at the Server System

Transactions are assembled by the application, not by the transport. If necessary, the LASTport protocol divides transaction requests into **segments**, each of which as a **segment number**, before transmitting them on the circuit. In this respect, the protocol is similar to other transport protocols. However, segmentation is controlled by the system application, which assigns each message segment a unique identifier.

Because each segment is uniquely identified, packets can be delivered in non-sequential order and reassembled at the destination. Figure 1–4 shows how the LASTport protocol identifies message segments.

Figure 1–4 LASTport Packets

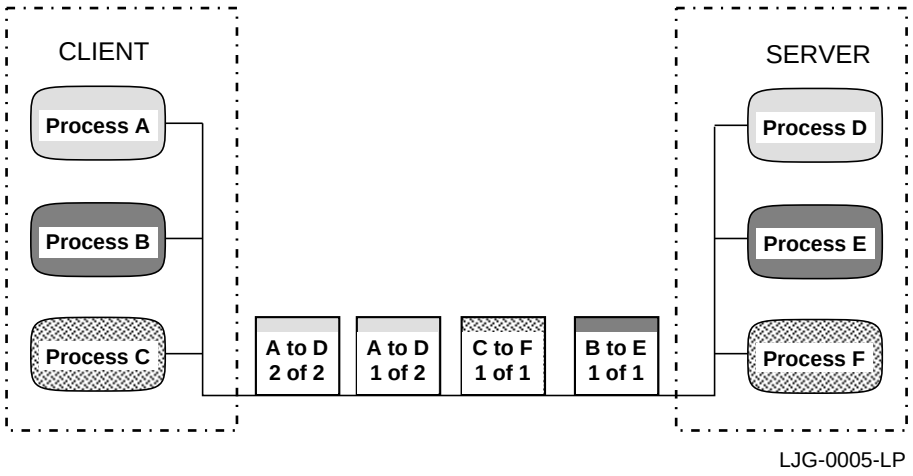


The system application reassembles the message from the packets by using information available in the LASTport message **headers**. The LASTport server's ability to process messages in arbitrary arrival order conserves server system memory.

Support for Large Numbers of Concurrent Transactions

The LASTport protocol can conduct up to 255 processes concurrently on a single circuit path and can support up to 65,536 paths concurrently. Figure 1–5 shows multiple processes using a single LASTport circuit path.

Figure 1–5 Multiple Processes Using a Single Path



High Availability with Redundant Paths

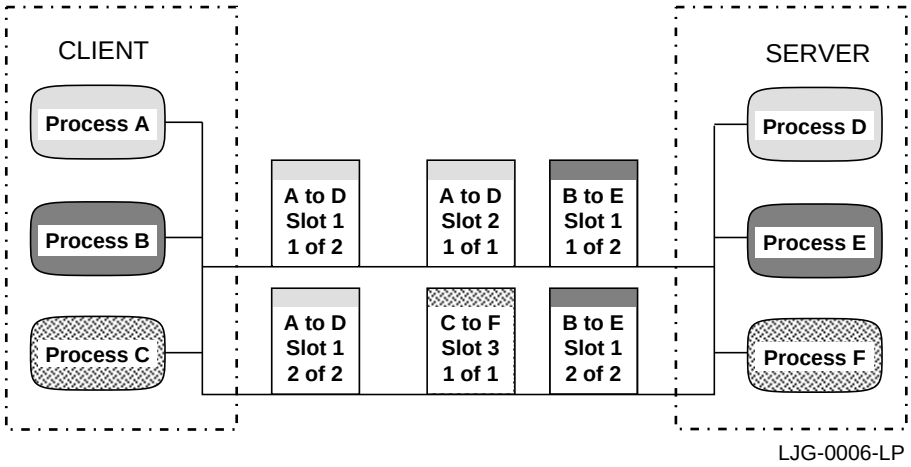
Because a LAN can support multiple adapters to each interconnect as well as multiple interconnects, more than one path to a destination is often available on a single circuit, which is maintained until the client disconnects. The LASTport protocol can detect and use all possible paths to send packets to a destination. If multiple paths between a client and a server are available, the LASTport protocol dynamically load-balances packets across all available paths.

Multiple paths also make the LASTport transport more robust than single-path transports. When one path fails, packets are still sent successfully over the other available paths.

The LASTport protocol uses a rate-based algorithm to control congestion and to prevent interconnect saturation.

Figure 1–6 shows multiple processes using two circuit paths.

Figure 1–6 Multiple Processes Using Two Paths



Effective Error Recovery

The LASTport protocol detects and controls errors by requiring that the system application generate only transactions that are both idempotent and commutative; identify transactions; and use timers.

- **Idempotent transactions** can be repeated without changing the system state. Any service can be structured to be idempotent.
- **Commutative transactions** can be processed in any order without affecting the outcome. For example, successive additions are commutative.
- A transaction identification scheme allows multiple transactions to be processed concurrently and ensures that any transaction is processed once and only once, even though the request might be transmitted more than once.
- The LASTport protocol uses timers to limit the response time to a request, limit the completion time for a transaction, and resynchronize transactions.

The LASTport protocol can recover both from path failures and from lost individual transactions. Adapter and interconnect failures do not affect the continuity or correctness of service. Such failures can delay service but do not stop it.

If a higher level of error detection and correction is needed, the system application can so specify. Note that security issues are outside the scope of the LASTport architecture.

1.2.2 Protocol Constraints

As compared with traditional transport protocols, the LASTport protocol offers a reduced set of service guarantees to clients. Constraints and rationales are as follows:

- CONSTRAINT: The client can initiate requests; the server cannot. This constraint simplifies connection management, buffer management, and flow control. Application relationships are client to server, not peer to peer.

RATIONALE: To create a transaction that operates as peer to peer, two independent **associations** must be established. Asymmetry simplifies protocol design, specifically error recovery.

- CONSTRAINT: The LASTport protocol performs error correction by retransmitting the entire request, which must be idempotent.

  RATIONALE: The environment assumes a low rate of errors. Because failure is rare, retransmission is assumed to be infrequent and therefore to consume comparatively few resources.

- CONSTRAINT: The client must describe the entire operation at the time the request is made to the transport. At the time of the request, the system application must define the address and size of both the request buffer and the response buffer.

  RATIONALE: This mechanism allows messages to be processed in arbitrary order and avoids unnecessary copying of data.

- CONSTRAINT: The system application request size is bounded by the error rate from all LAN sources: very large requests might never complete.

  RATIONALE: Although individual segments can be recovered, such recovery would complicate the protocol and consume additional CPU and memory resources.

- CONSTRAINT: The system application must implement transactions that are both idempotent and commutative, and it is responsible for transaction request, execution, and response sequencing.

  RATIONALE: The LASTport protocol identifies transactions and provides transaction concurrence: transactions complete across the session interface in the order that responses arrive. No circuit state need be maintained to preserve relative transaction ordering.

1.3 Functional Layers

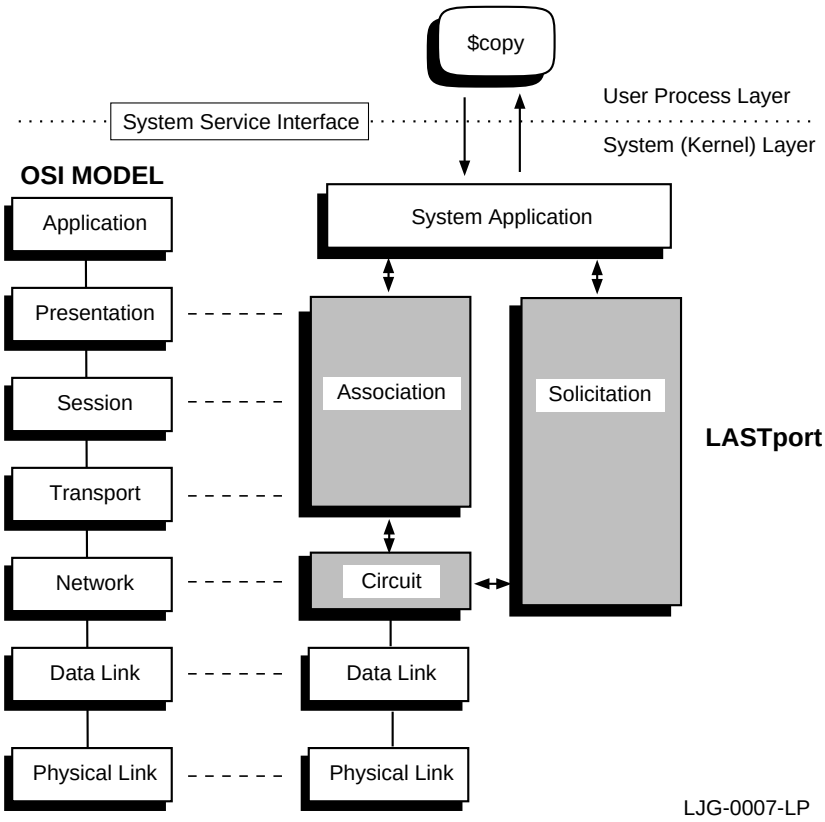
The LASTport architecture comprises three functional layers:

1. The **Solicitation layer**, which is a network naming service
2. The **Circuit layer**, which performs some functions of an OSI Network layer
3. The **Association layer**, which performs some functions of an OSI Transport layer

Sections 1.3.1 through 1.3.3 briefly describe these layers.

Figure 1–7 shows LASTport functional layers in the context of a computer system and indicates how the layers correspond to the seven-layer OSI network model. Note that the figure illustrates only formal correspondences, not the relative number of functions performed or lines of code executed in each layer.

Figure 1–7 Comparison of OSI Network Model and LASTport Architecture



As shown in Figure 1–7, LASTport layers are bounded on the top by their interface to the System Application layer and on the bottom by their interface to the **Data Link layer**. Naming services, performed by the LASTport Solicitation layer, are not illustrated in the figure.

1.3.1 Solicitation Layer

The Solicitation layer locates services on the network for the Association layer. This function is also called a directory or naming service. The Solicitation layer uses the multicast capability of the LAN to determine where a required service is located and multicasts solicit messages to all adapters on the network. The service responds with a message identifying its address, and the Solicitation layer passes this information to the Association layer. Chapter 3 describes Solicitation layer functions in detail.

1.3.2 Circuit Layer

The Circuit layer isolates the Association layer from failures in the **network topology** by detecting node failures and reporting them to the Association layer. Because Association layer functions are separated from Circuit layer functions, the Association layer continues to perform its tasks even when changes occur in path availability. If one path fails, the Circuit layer maintains any other available paths. The Circuit layer can detect when a failed path becomes available again. Chapter 4 describes Circuit layer functions in detail.

1.3.3 Association Layer

The Association layer, which manages connections between a client and a server, performs the following functions:

- Executes transactions
- Monitors transactions
- Segments and reassembles transaction data
- Assigns unique transaction identifiers to each transaction
- Handles errors

The Association layer presents four services to the system application:

1. **Registration services**. Registration services enable the system application to interact with the LASTport protocol. These services make the Directory, Association, and Data Transfer services available to the system application. Registration services are not described in the LASTport architecture, because they are implemented differently for each system.
2. **Directory services**. Directory services, which are implemented in the Solicitation layer, locate a server for the system application.
3. **Association services**. Association services enable a client to connect to and disconnect from a server.
4. **Data transfer services**. Data transfer services enable clients to send requests to servers and enable servers to respond.

Chapter 5 describes Association layer functions in detail.

2
Transaction Management

This chapter explains how the LASTport protocol performs and controls transactions. Topics include the following:

- Transaction guarantees and requirements
- Transaction timing
- Transaction identification
- Transaction error handling
- Summary of transaction processing operations

2.1 Transaction Guarantees and Requirements

A LASTport transaction consists of a request and a response, either of which can include multiple messages. In this request–response model, the client initiates all requests, which can be for connections, data processing, or **disconnections**; the server must respond to client requests. The client is responsible for detecting errors and for retransmitting the request if necessary. Using timers, system applications control the interval and retry limits for each request and response (see Section 2.1.4).

The LASTport protocol guarantees that a transaction is performed at least once by the server and that the transaction is completed only once at the client. To qualify for LASTport guarantees, transactions initiated by system applications must be both **idempotent** and **commutative**. Section 2.1.1 discusses transaction guarantees; Sections 2.1.2 and 2.1.3 describe transaction requirements. The LASTport protocol also identifies valid transactions, as discussed in Section 2.2.

2.1.1 Transaction Guarantees

In peer-to-peer transport protocols, each message is an independent event, and a message to a peer node elicits an acknowledgment. However, the acknowledgment guarantees only that the message has been delivered, not that the data has been processed.

In contrast, the LASTport protocol pairs the request with the response. A LASTport transaction is circular, consisting of the client message to the server and the server response. The client considers the entire transaction active until the client receives either a response or a notification that the transaction has aborted. The response guarantees that the server both received the transaction request and processed the transaction, and that all retry activity at the server has completed.

If the client does not receive a response or a message to resynchronize or abort the transaction within an established time, the client recovers by retransmitting the entire transaction. However, this behavior can cause the server to receive more than one transaction request.

To support the guarantee that a transaction is completed once and only once, and that retransmitting a transaction request does not endanger data integrity, the LASTport protocol requires that transactions be both idempotent and commutative. Sections 2.1.2 and 2.1.3 discuss these characteristics.

2.1.2 Idempotent Transactions

Implementers of system applications must ensure that all transactions issued by the client are idempotent.

An idempotent transaction can be repeated without changing the system state. The repeated transaction request has the "same strength" as the original request, assuming that the request is repeated in isolation from other requests.

For example, the operation $a <- a + 1$ (set the value of the variable a to $a + 1$) is not idempotent. If the initial value of a is zero, then the first time the operation is performed, the value of a is 1, and the second time the value of a is 2. Repeating the message changes the state of variable a. In contrast, the operation $a <- 1$ (set the value of the variable a to 1) is idempotent, because repeating the operation always sets the value of the variable a to 1.

The LASTport protocol is designed so that if a transaction is repeated in an attempt to receive an acknowledgment, the state of the application data remains consistent. The result is the same as if the transaction were completed only once, even if the server executes the transaction more than once.

This feature greatly simplifies error handling. The state of the server is consistent even if the operation is performed multiple times. Thus, the server state partially supports the guarantee that transactions complete only once: at the server, processing the transaction more than once is equivalent to processing it only once.

To ensure idempotency at the client, the LASTport protocol need only guarantee that the client receive the response from the server and can pair it with the correct iteration of the transaction, declaring all other transmissions of the transaction request obsolete. The LASTport protocol guarantees that, when the transaction completes at the client, all processing has completed at the server, and that multiple attempts to complete the same transaction do not endanger data integrity at the server.

Naturally idempotent procedures, such as access to read-only data, naming services, and time services, can use the LASTport protocol directly. However, any service can be structured to be idempotent. For example, end-to-end checks in the System Application layer can transform nonidempotent operations into idempotent operations. Some distributed operations that are not idempotent can be made so by inserting a layer between the procedure and the **communication interface**.

2.1.3 Commutative Transactions

Commutative transactions can be completed in any order without introducing errors in the result. The LASTport protocol requires that all concurrently executing transactions be commutative. This requirement enables the LASTport protocol to process multiple transactions concurrently, because transactions can complete across the session interface in the order that responses arrive. No state need be maintained to preserve the order of transactions.

The system application must implement transaction request sequencing, transaction execution sequencing, and transaction response sequencing. If an applications's transactions are commutative, the LASTport protocol supports the application.†

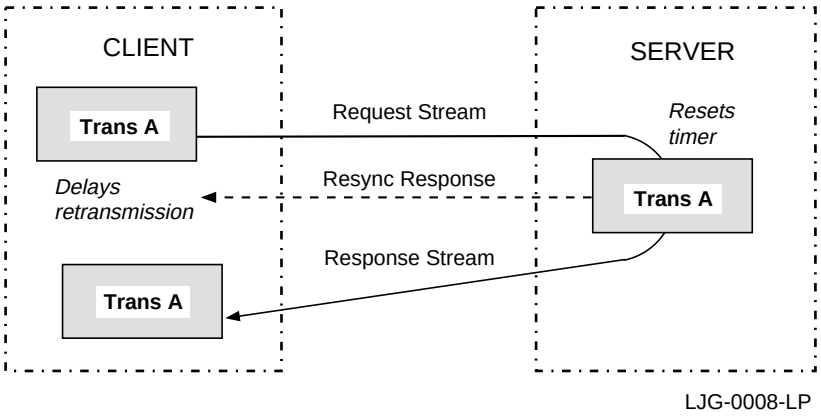
2.1.4 Transaction Timing

The LASTport protocol uses the variable AsnTransactionResponseTimer to perform error control during the execution of a transaction. When initiating a transaction, the client system application sets two values for the AsnTransactionResponseTimer:

1. The AsnTransactionResponseShortTimer value, which is an initial time limit for receipt of a Resync Response message or Transaction Response message. This value is required to determine whether the request message is lost.
2. The AsnTransactionResponseLongTimer value, which is subsequently used as a time limit for completion of the transaction and can be reset by the server using Resync Response messages. This value is required because actual application processing delay can never be known in advance.

In Figure 2–1, the server resets the AsnTransactionResponseTimer to the value of AsnTransactionResponseLongTimer to indicate a delay in processing Transaction A and notifies the client with a Resync Response message. The client delays retransmitting the transaction request.

Figure 2–1 Resync Response Message Notifies Client of Processing Delay



Note that request and response can be segmented into multiple messages if either is greater than the maximum LAN packet size. However, request and response are treated architecturally as two separate atomic message streams. In contrast, the Resync Response message is always a single message segment that the server sends to the client to indicate that a transaction is being delayed.

Timer functions for a normal transaction are typically as follows:

1. The client system application specifies the following AsnTransactionResponseTimer values:

   AsnTransactionResponseShortTimer = 3 seconds

† If an application's transaction requests are not or cannot be made commutative, the application can specify a TRANS_SLOT value of 1 (see Table 6–14) to ensure that transactions are processed sequentially.

AsnTransactionResponseLongTimer = 30 seconds

2. The client issues a transaction request, setting the AsnTransactionResponseShortTimer value to 3 seconds.
3. If the client's AsnTransactionResponseShortTimer expires, the client does the following:
   a. Resets the AsnTransactionResponseTimer to the value of the AsnTransactionResponseShortTimer.
   b. Calls the CircEvent process with a *CircPathMaintReq* event (see Table 4–2).
   c. Reissues the request up to the number of times specified by the AsnTransRetransmitLimit variable.
   d. If the transaction is still unsuccessful, the client Association layer declares an *AsnTransFailure* event to the system application (see Table 5–2). The association is then aborted and the corresponding circuit terminated.
4. In response, the Circuit layer calls the AsnClientEvent process with an *AsnCircuitUp* or *AsnCircuitDown* event (see Table 4–3).
   - An *AsnCircuitUp* event causes the transaction to be retried.
   - An *AsnCircuitDown* event causes all underlying associations and all multiplexed transactions to be asynchronously aborted.
5. If a transaction is delayed at the server beyond AsnTransactionResponseShortTimer (for example, a large compute-bound transaction is being processed), the server sets the AsnTransactionResponseTimer to the AsnTransactionResponseLongTimer value (30 seconds in this example) and sends a Resync Response message to the client. This mechanism prevents lengthy server computation from being aborted by the client reissuing redundant transaction requests. If the server AsnResyncResponseTimer expires, the server transmits a Resync Response message to the client and resets the timer.

   To avoid unnecessary transaction retries, the server accounts for the transient delay with the AsnResyncResponseTimer, an architectural constant with a value of one second.
6. After the server association system application finishes processing the transaction, the server transmits the response message to notify the client that the transaction has completed.

2.2 Transaction Identification

The LASTport protocol can accommodate concurrent transactions between a client and a server and enables transactions to be segmented and sent on the network as independent packets. The LASTport protocol must therefore include a method of identifying the packets by transaction. When the Association layer receives a request from the client system application, the Association layer assigns a **transaction identifier** to each transaction. When a node running the LASTport protocol receives a packet, the node reads the transaction identifier and passes each packet to its appropriate destination buffer.

A LASTport transaction identifier enables independent clients to determine the sequence of operations at a server. In particular, the transaction identifier is used to detect and correct problems caused by lost or delayed packets. Error handling is discussed in Section 2.3.

The transaction identifier supports the following semantics:

- **Client request semantics**. Perform the server procedure at least once or cause the association to fail. The client specifies the retry interval and the retry count.
- **Client response semantics**. Perform the procedure at least once. No execution at the server occurs after the response is delivered successfully.
- **Server request semantics**. Perform the procedure exactly once.
- **Server response semantics**. Return the result of the executing procedure exactly once.

The transaction identifier has two parts: a **transaction slot** and a **transaction sequence number**. Section 2.2.1 discusses transaction slots, and Section 2.2.2 discusses transaction sequence numbers.

2.2.1 Transaction Slots

Transaction slots are used to differentiate concurrent transactions and can be thought of as teller windows at a bank. Customers who want to make bank transactions wait in a queue until a teller is available. The number of tellers defines the number of transactions that can be performed concurrently. Whenever a teller becomes available, the teller can serve the next customer in the queue, even if other customers have not finished performing their transactions with other tellers.

In the same way, the total number of slots available determines the number of LASTport transactions that can be performed concurrently over a single association. Whenever a slot becomes available, the LASTport protocol performs the next transaction in the queue, even if earlier transactions have not completed.

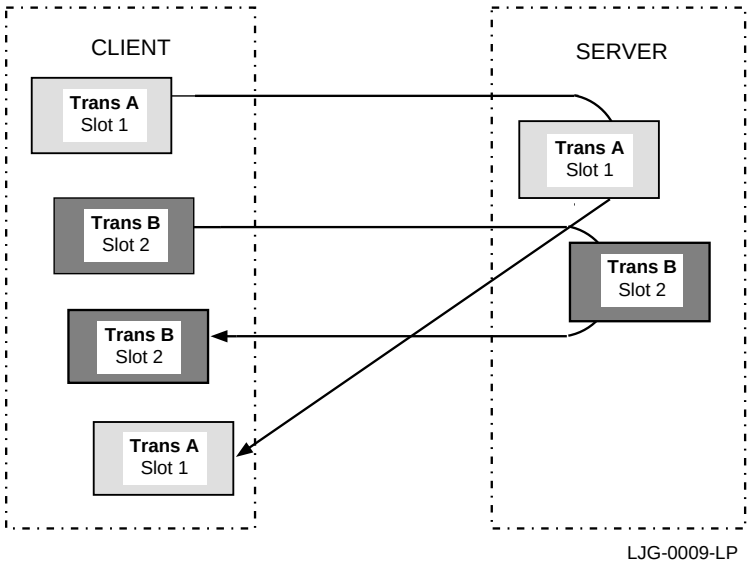
The client Association layer assigns a **transaction slot number** in the transaction request, and the server responds using the same slot number. Each transaction is assigned to one slot. If transactions must be retried, they are always retried on the same slot. In other words, if more than one transaction is being processed, the transactions and their retries are differentiated by slot number. To continue the bank analogy, each customer in a bank requests a single teller and receives responses only from that teller.

A slot is implemented as an 8-bit field, and slots are numbered from 1 to 255. The maximum number of slots at a given time in the system is negotiated between the client and the server. At the time of solicitation, the server can request that a smaller number of slots be used for the association.

The client Association layer uses all available slots to perform transactions. If all slots are full, the Association layer queues the transactions until a slot becomes available.

Figure 2–2 illustrates two transactions, A and B, that use Slots 1 and 2 concurrently. Note that B completes before A, though B was queued after A.

Figure 2–2 Two Transactions Using Two Slots Concurrently



2.2.2 Transaction Sequence Numbers

A transaction sequence number is an 8-bit field (value 1 to 255) that, along with the transaction slot number, uniquely identifies a transaction. In the client Association layer, each transaction slot has a counter to record the current sequence number. On an initial transaction attempt, the client Association layer assigns the next available slot number and sequence number. When it receives the transaction, the server Association layer records the current sequence number for each slot. Thus, for a normal successful transaction, the sequence number at both the client and server is always the same.

However, if the intitial transaction attempt fails (for example, if packets are lost or a transaction timer expires), the client Association layer increments the sequence number for the transaction slot before retrying the transaction. Thus, for a transaction retry, the slot number remains the same while the sequence number is incremented. When it receives the transaction retry with the incremented sequence number, the server processes this transaction immediately. To support the guarantee that only viable transactions are completed, the server remembers the sequence number of the most recent transaction and rejects transactions with earlier sequence numbers (see Figure 2–3).

2.2.3 Sequence Number Algorithm

A range of **forward sequence numbers** is defined for each slot at the server as $n + 1$ to $n + 127$, where n is the sequence number of the most recent transaction processed on that slot.† This range covers half of the 256 possible sequence numbers. Sequence numbers outside that range are assumed to be from previous transactions, and such transactions are rejected at the server. To accommodate the possibility of lost, and therefore, non-sequentially numbered requests, the server processes any requests in the set of forward sequence numbers.

By identifying and rejecting expired transactions, the LASTport protocol successfully conserves resources at both the server and the client.

† In the next version of the architecture, the range will be $n + 1$ to $n + (2^{31} + 1)$.

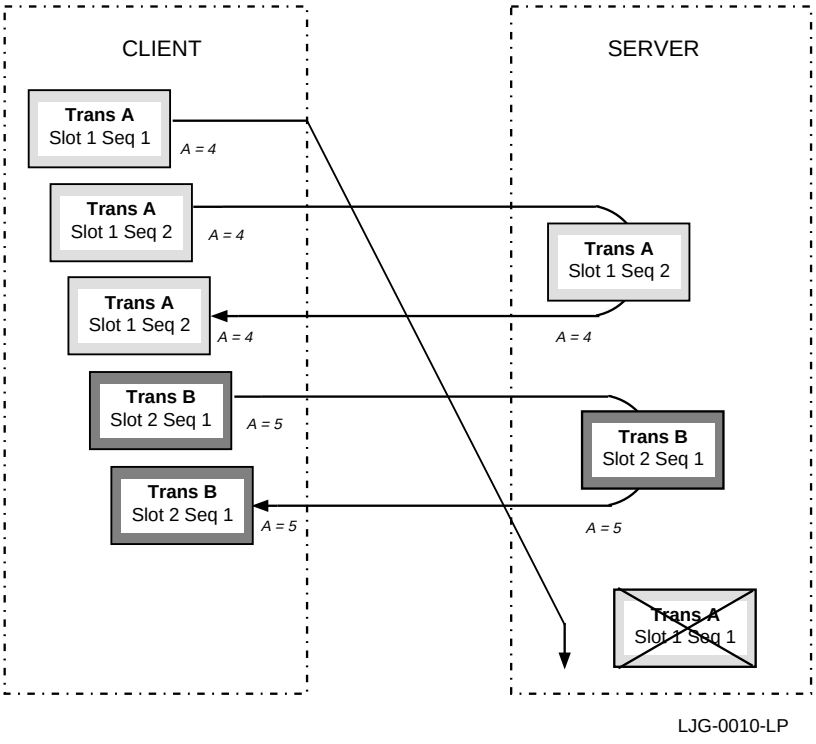
2.2.4 Concurrent Transactions

The transaction identifier, with its slot and sequence numbers, enables the LASTport protocol to handle concurrent transactions, detect errors, and ensure idempotency of transactions. When a transaction is issued, it is assigned the next available slot and sequence number. If the transaction must be retried, the slot number remains the same while the sequence number is incremented.

Figure 2–3 shows that sequence numbers must be incremented when the client reissues a transaction request. If the numbers are not incremented, recovery from lost or delayed messages would be impossible, because the server would execute a request for a transaction that has already been completed. In so doing, the server would violate the client's guarantee to the system application that a transaction is completed once and only once. Therefore, the client uses sequence numbers to identify an initial transaction request and each retry uniquely. The client increments the sequence number each time the client reissues a transaction request.

In Figure 2–3, incrementing the sequence number enables the server to discard the failed transaction A on Slot 1, Sequence 1, because the server has already processed the reissued transaction A on Slot 1, Sequence 2. Thus, the client can guarantee to the system application that the transaction has processed correctly and will not be reprocessed.

Figure 2–3 Sequence Number Incremented to Recover from Lost Message



The events shown in Figure 2–3 are as follows:

1 The client issues transaction A on Slot 1, Sequence 1. The request sets the value of variable *A* to 4. This transaction is delayed on its way to the server, and the transaction times out.

2 The client reissues transaction A using the same slot but increments the sequence number. The second attempt is issued on Slot 1, Sequence 2 and completes immediately. The value of variable *A* at the server is 4.

3 The server returns the response to the client, and the client marks transaction A as complete. The value of variable *A* at the client is 4.

4 The client issues transaction B on Slot 2, Sequence 1 and sets variable *A* to 5. The server completes the transaction and returns a value of 5 to the client.

5 The delayed transaction A request arrives at the server and is discarded.

Incrementing the sequence number for each transaction attempt, new or repeated, eliminates the problem of having two transactions with the same transaction identifier. However, the server must be able to recognize a set of sequence numbers that are higher than the ones previously received.

2.2.5 Guarantees for Concurrent Transactions

The LASTport transaction identification scheme makes possible the following guarantees for concurrent transactions:

At the client:

- The client knows which transaction is current on any slot by recording the transaction identifier, which includes the assigned sequence number.

  The client identifies and rejects responses that come from expired transactions. Expired transactions have transaction identifiers that are not identical to the current transaction identifier at the client.

- The client knows which transactions are repeated. It always attempts to receive a response from the server by retransmitting the entire transaction, but differentiates between transaction versions using the transaction identifier.

  Guarantee:

  The client guarantees that once a transaction completes at the presentation interface, activity associated with the transaction request at the server, including retries, is complete.

At the server:

- The server knows which transaction identifiers have forward sequence numbers, thereby minimizing the number of times it attempts to process expired transactions.

- Because server cannot detmine whether a particular transaction is a retransmission of a previous transaction, the server treats each valid transaction it receives as a new transaction.

  Guarantee:

  The server guarantees that it executes a transaction request only once and only if the transaction's sequence number is in the range of forward sequence numbers. The server's response to the client guarantees that the transaction has been processed exactly once.

2.3 Transaction Error Handling

The LASTport protocol handles the following error conditions:

1. Incomplete transactions
2. Orphan transactions

Incomplete transactions result when packets are lost. Refer to Section 2.3.1 for a description of this condition. **Orphan transactions** are completed transactions that are not current. These are discussed in Section 2.3.2.

All errors are corrected the same way: the client retransmits the entire transaction to the server.

2.3.1 Incomplete Transactions

Only the client can detect that a transaction has failed and perform error recovery. If a timer expires and the client has not received the complete response to a request, the client assumes that packets have been lost and reissues the transaction. This condition occurs in the following circumstances:

1. A segment of the request is lost on the way to the server. The server waits for the entire transaction to arrive before executing it; however, the lost segment never arrives. If the AsnTransactionResponseTimer expires, and the client has received neither the transaction response nor a resynchronization message, the client transmits the same transaction on the same slot with an incremented sequence number. When the server receives this transaction request with the same slot number but a higher sequence number, the server deletes the previous request from its buffers and begins to process the current transaction.
2. The response is completed at the server, but a portion of the response is lost on its way to the client. The client waits for the rest of the response, which never arrives. The process timer expires, and the client deletes the incomplete transaction from its buffers, increments the sequence number, and reissues the transaction. The server generates another response.

2.3.2 Orphan Transactions

Orphan transactions are a special error category that the LASTport protocol controls to enable independent clients to determine the sequence of operations at a server. An orphan transaction is a complete request or response that is no longer current but is still active in the circuit.

This condition differs from the case in which errors result from lost packets. When packets are lost in the circuit, the transmission never completes and retransmitting the original request results in only one response.

In contrast, orphan transactions occur when a request or response is delayed in the circuit. During the delay, the AsnTransactionResponseTimer at the client expires, and the client retransmits the request. Both a current version and an expired version of the transaction exist in the circuit. To ensure that expired (orphaned) transactions do not endanger data integrity, the LASTport protocol implements special policies to detect orphan transactions both at the server and at the client.

At the server:

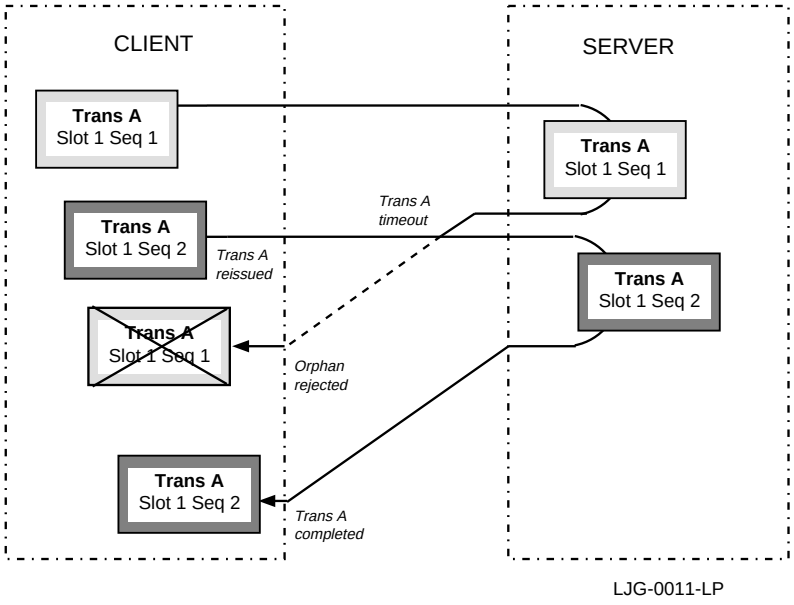
A transaction is an orphan at the server if the sequence number is not within the range of forward sequence numbers for that slot.

Typically, the first transaction request is delayed on its way to the server, and the client retransmits the request using the same slot and an incremented sequence number. The server processes and returns the second request to the client. The delayed first request arrives at the server after the server has processed and returned the second request. Because the delayed first request has the same slot number as the already processed second request but a lower sequence number, the server rejects the first request.

At the client:

As shown in Figure 2–4, a transaction is an orphan at the client if the transaction's sequence number is not equal to the sequence number of the current request on a specific slot. Note that the sequence number is incremented at each retry so that the client can distinguish the response to the most recent request from responses to previous requests.

Figure 2–4 Orphan Transaction at the Client with Delayed Response



The events shown in Figure 2–4 are as follows:

1 The client issues transaction A on Slot 1, Sequence 1.

2 The server processes the transaction and returns it on Slot 1, Sequence 1, but the transaction is delayed in transmission.

3 The client times out on transaction A, Slot 1, Sequence 1, because the client received neither a response nor a resynchronization message from the server for this transaction. The client reissues transaction A on Slot 1, Sequence 2.

4 The server processes the transaction because Sequence 2 is a forward sequence number. The server transmits the response to the client.

5 The transaction results from Slot 1, Sequence 1 arrive at the client before the results from Slot 1, Sequence 2.

6 The client discards the data that arrives on Slot 1, Sequence 1, because Sequence 1 is not the most recent request on Slot 1. The client waits for the response on Slot 1 that is labeled Sequence 2.

7 When the transaction on Slot 1, Sequence 2 arrives, the client marks the transaction as complete.

Note that if the client completed the transaction at step 5, independent clients could not determine the sequence of operations at the server. That determination would be impossible, because the requent generated at step 3 would be valid and would execute successfully at the server any time after the transaction completed in step 5.

2.4 Summary of Transaction Processing Operations

The LASTport protocol processes transactions in three phases:

1 Association start (solicit and connect to service)

2 Data transfer

3 Association disconnect

To illustrate the interaction of the Solicitation, Association, and Circuit layers during normal transaction processing, Sections 2.4.1 through 2.4.3 describe the operations performed in each phase.

2.4.1 Association Start

To start an association, the LASTport protocol performs the following operations:

1 **Solicits a service**

The system application on a client instructs the Solicitation layer to locate a service on the network. In this case, a user wants to mount the device represented by the Service Name ONLINE_DOC.

Using all available adapters, the client Solicitation layer multicasts a Solicit Request message to the stations on the network.

2 **Responds to the service request**

A server that offers the service ONLINE_DOC responds with a Solicit Response message to the client Solicitation layer. The message includes the network address of the ONLINE_DOC service.

The client Circuit layer records all paths on which Solicit Response messages are received and stores them in the event that a circuit is established. This information is purged periodically.

The client Solicitation layer passes the location of the service to the system application.

The client system application evaluates all the responses, selects the preferred service, and passes its location to the Association layer.

3 **Creates a circuit for the association**

The Association layer sends a request to the Circuit layer to create a circuit with the server node. In this example, it is assumed that a circuit does not already exist.

The client Circuit layer sends a Start message to the server, and the server responds with a Stack message.

Once the circuit is established, the client Circuit layer notifies the client Association layer that the circuit is up.

If a circuit already exists between the client and the server when the client Association layer requests the client Circuit layer to create a circuit with the server, the Circuit layer immediately responds that the circuit is up and does not issue a Start message.

4 Connects to the service

The client Association layer sends a Connect Request message to the server to establish an association.

When the server sends a Connect Response message, the connection is established over the circuit.

The client Association layer notifies the system application that the connection is established. The association is started, and data can now be processed.

2.4.2 Data Transfer

The following operations assume that the client system application writes data to the server.

The system application allocates buffer space for the expected response and then requests the client Association layer to perform a transaction. The application supplies both the data to be passed to the server and the address of a buffer for the response. Note that passing the buffer address to the Association layer eliminates the need for a buffer copy between the system application and the Association layer.

To transfer data, the LASTport protocol performs the following operations:

1 Identifies the transaction

The client Association layer assigns a transaction identifier to the transaction. The transaction identifier uses slot and sequence numbers to identify all messages belonging to a particular transaction. Both the client requests and server responses carry the identifier.

2 Segments the data if necessary

If necessary, the client Association layer divides the transaction data into segments. Each segment carries both the transaction identifier and a **segment number**. The segment number identifies the position of that segment in the transaction and is used to reassemble a transaction once the transaction packets reach the destination node.

The segments are numbered 1 of *n*, 2 of *n*, ...n of *n*, and so on, where variable *n* is the total number of segments. For example, a segment number might identify a segment as number 5 of 9 segments. If the transaction can be sent within one network frame, the transaction is not segmented, but it still carries a segment number such as number 1 of 1.

Each transaction packet is therefore uniquely identified. The slot number identifies the entire transaction, the sequence number identifies the iteration of that transaction, and the segment number identifies the position of that packet within the entire transaction.

This identification mechanism enables the LASTport protocol to perform concurrent transactions and to send packets across all available paths. When a packet arrives at a node, the Association layer has the information to determine the following:

- The transaction to which the packet belongs
- The status of the transaction, which is either current or obsolete
- The placement of the packet within the transaction

3 Transmits the data

The client Association layer passes the segments to the Circuit layer and starts a timer to monitor the transaction.

The client Circuit layer transmits the packets. As a result of the solicitation process, the Circuit layer is aware of all known paths to the server. Packets using this circuit can be multiplexed across all available paths.

If there is more than one packet, and more than one path is available, the circuit load-balances the packets across the available paths.

If the client receives neither a response nor a request for resynchronization before the timer expires, the Association layer resends the entire transaction. Section 2.3 describes the conditions for repeating transactions.

4 Resegments (reassembles) the data

The server Circuit layer receives the packets and passes them to the Association layer for processing.

Until the first segment arrives, the server does not know the size of the transaction. Rather than allowing the Association layer to reassemble the entire transaction and copy it to the system application, the LASTport protocol performs the following operations:

a As the segments arrive at the server Circuit layer, the Circuit layer passes each packet to the Association layer in order of arrival.

b The server Association layer buffers the segments until it receives the first segment, number 1 of *n*, or eventually discards the transaction request. The first segment contains information about the total size of the request. The Association layer passes segment number 1 of *n* to the system application.

c The system application allocates adequate buffer space and passes the address of the buffer to the Association layer. Reassembly of message data in the buffer can now proceed in arbitrary message order.

d The server Association layer reassembles the message into sequential order in the system-level buffer and notifies the system application when reassembly is complete. Note that if some segments are lost, the Association layer eventually notifies the server that the transaction has been aborted.

The system application can now process the transaction data.

5 Transmits a response

The server system application passes the response data to the server Association layer.

The server Association layer identifies the transaction by using the same transaction identifier that was assigned by the client. If the response is larger than one network frame, the response is segmented as described in Section 2.1.4. Segments are passed to the Circuit layer.

The server Circuit layer distributes the segments across all available paths, thus making efficient use of the available bandwidth.

Note that the server never sends the client an acknowledgment message for the request, thereby conserving network bandwidth for request and response messages containing data.

6 Resynchronizes the transaction if necessary

If the client system application does not respond within the time allowed by the client request, the server issues a resynchronization message to the client. This message is an important exception to the rule that all transactions take place in pairs. The message informs the client that the transaction is processing but will take longer than the time allotted by the client. The client then resets its timer for that transaction. The resynchronization message prevents the client from sending the server redundant requests that would unnecessarily consume server resources.

7 Processes the response data

The response data arriving at the client is passed from the Circuit layer to the Association layer in the order the packets are received. The client Association layer reassembles the response data in the buffer that the system application supplied at the beginning of the transaction.

Rather than buffering segments until it receives number 1 of *n*, as the server does, the client Association layer immediately reassembles the segments in the available buffer at the system application level. The client can do this because the client system application allocated buffers for the response when it issued the request, and because all segments except the final one are equal in length. Because the LASTport protocol need not buffer response data in system buffers, latency of individual transaction operations decreases, and the number of simultaneously supported associations increases.

2.4.3 Association Disconnect

To disconnect (release) an association, the LASTport protocol performs the following operations:

1 Initiates the disconnect

The client system application notifies the client Association layer that it wants to disconnect from the server. The client Association layer constructs a Disconnect Request message and passes it to the Circuit layer for transmission to the server.

When the server receives the request, the server Association layer notifies the server system application.

2 Disconnects the server

The server system application notifies the Association layer that it is disconnected and transmits any response data related to the disconnection.

The server Association layer constructs a Disconnect Response message to confirm the disconnection and passes the message to the Circuit layer for transmission to the client.

3 Disconnects the client

When the client receives the Disconnect Response message, the client Association layer notifies the system application that the response to the disconnect request has arrived.

The Association layer also notifies the Circuit layer that the association has been disconnected. If this was the only association using the circuit, the circuit is terminated. If there are other associations still active on that circuit, it remains active.

3
Solicitation Operations

The Solicitation layer provides a combined service for directory operations, association management, and task naming.

This chapter discusses Solicitation layer operations. Topics include the following:

- LASTport naming service
- Solicitation work groups
- Solicitation event processing

3.1 LASTport Naming Service

Naming services are critical to the LASTport architecture. The LASTport naming service uses Advertisement, Solicit Request, and Solicit Response messages to enable client applications to find services offered by servers on the LAN. The naming service assumes that a LAN-based multicast service is available and requires participation by the systems on which the clients and servers reside. Servers multicast Advertisement messages to advertise node-specific services to their potential clients. Clients find services by multicasting Solicit Request messages to all servers on the LAN.

To ensure that all potential servers hear requests, clients transmit requests on all available adapters. Any server that offers the requested service answers the client with a Solicit Response message addressed back to the requesting node's physical address. If the client receives several responses to a Solicit Request message, the client selects the response that offers the best service. If no Solicit Response message is received, Solicit Request messages can require retransmission.

The Advertisement, Solicit Request, and Solicit Response messages have the same basic format but differ in the MSG_TYPE field in the circuit message header (see Chapter 6). The Solicit Request and Solicit Response messages are also used to maintain circuit topology, as described in Chapter 4.

Note

The LASTport architecture accommodates third-party general naming services. However, such services have the following drawbacks:

- They can drain resources.
- They can prevent integrating the association with the LASTport naming service.

3.1.1 Solicit Request Message

Solicit Request messages are directed to a particular server application. A client application addresses its Solicit Request message to a set of server applications on the LAN by specifying a particular **Service Class**. When the Solicitation layer receives a Solicit Request message, the message is parsed, and all the information in the message is passed to the server application. The server application searches its database and responds if it offers the requested service. To ensure that all the servers on the LAN receive the request, Solicit Request messages are transmitted on all adapters.

3.1.2 Solicit Response Message

Although the Solicit Request message is multicast to an entire Service Class, the message receives, at most, a small number of replies. The server transmits the Solicit Response message on the same adapter that received the Solicit Request message. The Solicit Response message is transmitted directly to the requesting node and is not multicast.

When the server Solicitation layer generates a Solicit Response message, the REQUEST_SEQUENCE field from the Solicit Request message is copied to the same field in the Solicit Response message (see Table 6–8).

3.1.3 Solicitation Response Policies

The system application must negotiate a **dally** protocol for Solicit Response messages. For example, the LAD architecture defines a range in which messages can be randomly transmitted. For more information, refer to the *Local Area Disk Architecture Specification* document.

3.2 Solicitation Work Groups

The LASTport architecture supports segmenting networks into classes of nodes by labeling each client and server with a work group code. These work group codes determine the subset of nodes that participate in the naming service. The Solicitation layer defines work group codes to multicast messages to a subset of the servers on the LAN, thus using the available bandwidth more efficiently.

For example, assume that LASTport nodes exist on either side of a point-to-point synchronous bridge. Creating a work group on each side of the bridge allows multicast addresses to be filtered by the bridge.

A member of a group transmits and receives a particular multicast address on the LAN. LASTport nodes can be members of several or all groups simultaneously. The LASTport architecture defines 1023 separate groups.

The LASTport protocol uses the multicast addresses 09–00–2B–04–00–00 through 09–00–2B–04–FF–FF for work groups. The work group code is added to the last two bytes of a multicast address. Table 3–1 shows the order in which the bytes must appear on the wire.

Table 3–1 Codes for Work Groups

Work Group	Multicast Address
0 (default)	09–00–2B–04–00–00
1	09–00–2B–04–01–00
2	09–00–2B–04–02–00
15	09–00–2B–04–0F–00
100	09–00–2B–04–64–00
512	09–00–2B–04–00–02
1023	09–00–2B–04–FF–03
1024–65535	Reserved

When a Solicit Request message is transmitted, it is sent to all the multicast addresses that correspond to the work group or groups of which the node is a member. Only servers in these work groups have these multicast addresses enabled and are the only nodes to receive the Solicit Request message.

3.3 Solicitation Event Processing

The Solicitation layer has two states at the client (listed in Table 3–2) and is stateless at the server. A state can change when local events occur or when a message is received from a system application.

Table 3–2 Client Solicitation States

State	Description
Halted	A solicitation is either inactive or does not exist.
Running	A solicitation is active.

A solicitation moves through the defined states by processing events. Events are either generated locally or in response to a message from a system application. Tables 3–3 through 3–5 list Solicitation layer events.

Table 3–3 Events Processed by the Client Solicitation Layer

Event	Description
SolReqMsgSend	Declared by the client system application to generate a Solicit Request message. The client system application must supply the length of time to wait for Solicit Response messages.
SolRspMsgRcv	Generated when a Solicit Response message is received.
SolTimeout	Generated when the solicit period ends.
SolCancelReq	Declared by the client system application to terminate the solicitation process.

Table 3–4 Events Processed by the Server Solicitation Layer

Event	Description
SolReqMsgRcv	Generated when the server receives a Solicit Request message.
SolRspMsgSend	Declared by the server system application after receiving a Solicit Request message. The server system application then generates a Solicit Response message.

Table 3–5 Events Generated by the Solicitation Layer

Event	Description
AppSolReqMsgRcv	Declared by the server Solicitation layer to the server system application to indicate that a Solicit Request message needs to be processed.
AppSolRspMsgRcv	Declared by the client Solicitaion layer to the client system application to indicate that the client received a Solicit Response message.
AppSolTimeout	Declared by the client Solicitation layer to the client system application to indicate that the solicit period has ended.

3.3.1 Solicitation Timer

The Solicitation layer uses a timer called *SolicitResponseTimer* to control the listening period during which Solicit Response messages are passed to the client. The length of this timer is set when the client starts the solicit process. The Solicitation layer passes all solicit responses to the client application until the *SolicitResponseTimer* expires. After the timer expires, the client ignores Solicit Response messages for the particular Solicit Request message.

3.3.2 Solicitation Processes

To manage state transitions, the Solicitation layer uses the following processes:

- SolClientRcv
- SolClientEvent
- SolClientTimer
- SolServerRcv
- SolServerEvent

Sections 3.3.2.1 through 3.3.2.5 describe these processes. The processes rely on the Solicitation Context Block (SCB) data structure, which holds the state and context of each solicitation. For example, the SolClientTimerProcess described in Section 3.3.2.3 uses the SCB to point to the system application's Session State Block (SSB), which contains association data. Note that the following process pseudocode examples do not provide all details and are intended only as rough guidelines. Note further that the "sysApp" prefix refers to the current system application.

3.3.2.1 SolClientRcv Process

The SolClientRcv process is called by the SolClientEvent process to notify the Solicitation layer when messages arrive and to effect appropriate state transitions. The process performs actions shown in the following pseudocode example.

```
void SolClientRcv(event, message, scb)
int event;
SolicitMessage *message;
SCB *scb;
{
     /*
      * Call SolClientEvent process to determine actions to be performed.
      */
     SolClientEvent(event, message, scb);
}
```

3.3.2.2 SolClientEvent Process

The SolClientEvent process is called by the SolClientRcv process and recursively to generate the solicitation protocol.

The process uses the current state and the events shown in Table 3–6 to identify the new state. (For example, if the current state is Halted and the event is *SolReqMsgSend*, the new state is Running.) The process then extracts the new state, updates the SCB with this state, and executes actions indicated by the callout (for example, ❶). These actions are described following the table. A dash (—) indicates that no action is required.

The SolClientRcv process performs actions shown in the following pseudocode example.

```
void SolClientEvent(event, message, scb)
int event;
SolicitMessage *message;
SolicitContextBlock *scb;
{
     int newState;
     int (*actionRoutine)( );

      /*
       * Using state table, retrieve new state and action routine.
       * Call action route to start or continue protocol message exchanges.
       */

     newState = solicitClientStateTable[event, scb–>state].newState;
     actionRoutine = solicitClientStateTable[event, scb–>state].actionRoutine;
     scb–>state = newState;

     (*actionRoutine)(message, scb, ...);
}
```

3.3.2.3 SolClientTimer Process

The SolClientTimer process is called by the operating system once each second. The process performs actions shown in the following pseudocode example.

```
void SolClientTimer( )
{
     /*
      * For each outstanding solicitation request, decrement time
      * remaining to wait for responses.
      */

    scb = scbListHead;
    while ((scb !=NULL)
    {
        scb–>SolicitResponseTimer--;
        if (scb–>SolicitResponseTimer == 0))
        {
         /*
          * When timer expires, notify system application that
          * solicitation process has completed.
          */
            sysAppClientEvent(SolTimeout, scb–>ssb);
        }
        scb = scb–>next;
    }
}
```

3.3.2.4 SolServerRcv Process

The SolServerRcv process is called by the SolServerEvent process to notify the Solicitation layer when messages arrive. The process performs actions shown in the following pseudocode example.

```
void SolServerRcv(event, message, scb)
int event;
SolicitMessage *message;
SCB *scb;
{
    if (message–>SERVICE_CLASS == 0)
    {
        CircRcv(message, scb–>CircuitId, scb–>nodeId);
    }
    else
    {
         /*
          * Call SolServerEvent process to determine actions to be performed.
          */

        SolServerEvent(event, message, scb);
    }
}
```

3.3.2.5 SolServerEvent Process

The SolServerEvent process is called by the SolServerRcv process and recursively to generate the solicitation protocol. The SolServerRcv process performs actions shown in the following pseudocode example.

```
void SolServerEvent(event, message, scb)
int event;
SolicitMessage *message;
SolicitContextBlock *scb;
{
    int newState;
    int (*actionRoutine)( );

     /*
      * Using state table, retrieve new state and action routine.
      * Call action routine to start or continue protocol message exchanges.
      */

    sysAppServerSolicitRcv(message)
}
```

3.3.3 Solicitation Client State Transitions and Server Actions

Table 3–6 shows client events and resulting state transitions. Callouts indicate any actions that result.

Table 3–6 Solicitation Client Events and Resulting State Transitions

	Current State	
	Halted	**Running**
Events		
SolReqMsgSend	Running ❶	—
SolRspMsgRcv	Halted	Running ❷
SolTimeout	—	Halted ❸
SolCancelReq	—	Halted

Actions for the state transitions shown in Table 3–6 are as follows:

❶ Generate a Solicit Request message (see Table 6–8) based on data supplied with the *SolReqMsgSend* event.

Start the SolicitResponseTimer to measure the solicitation period.

❷ Parse the Solicit Response message (see Table 6–8) and call the targeted client system application with an *AppSolRspMsgRcv* event.

❸ Call the client system application with a *SolTimeout* event.

Table 3–7 shows server events and actions.

Table 3–7 Solicitation Server Events and Actions

Event	Action
SolReqMsgRcv	Parse the Solicit Request message (see Table 6–8) and call the targeted server system application with an *AppSolReqMsgRcv* event.
SolRspMsgSend	Generate a Solicit Response message (see Table 6–8) based on data supplied with the *SolRspMsgSend* event.

4
Circuit Operations

The Circuit layer performs the following functions:

- Defines and maintains circuit topology.
- Detects new instances of nodes or node paths and forces the higher layers to resynchronize when nodes crash and reboot.
- Distributes message segments across all possible paths between the source and destination nodes — a process called load balancing. A maximum of one circuit at a time can exist between any node pair.
- Provides a simple form of rate–based congestion control (see Section 4.4).

Because the Association layer determines the policy for detecting duplicate messages and retransmitting messages. The Circuit layer does no error correction, message sequencing, or duplicate detection.

This chapter discusses the following topics:

- Circuit topology
- Circuit states
- Circuit event processing
- Congestion control

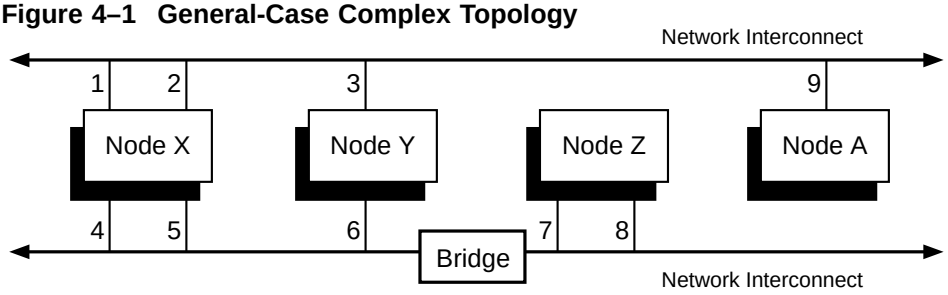
4.1 Circuit Topology

Before an association is created between a client and a server, the client Circuit layer must establish a circuit to the server. The client system application locates the server by using the solicitation services described in Chapter 3.

During the solicitation process, the LASTport protocol uses Solicit Request and Solicit Response messages to map the paths between client and server. The set of network paths connecting the client and server forms the **circuit topology**. This topology can change depending on which paths (logical links) and network adapters are available at a given time.

Figure 4–1 shows a general-case complex topology.

Figure 4–1 General-Case Complex Topology



In Figure 4–1, the numbers 1 to 9 are network physical address instances in the Data Link layer. Usually, each node address corresponds to a single system, but a system can implement multiple-node service access points.

Notice that all paths are logically point to point. Any environment that requires OSI Layer 3 Network functions (see Figure 1–7) must use bridges so that all paths remain logically point to point.

A path consists physically of two controllers and one network segment. Thus, in Figure 4–1, nodes X and Y can communicate with all other nodes, and nodes Z and A can communicate with nodes X and Y. But there is no way for nodes Z and A to communicate using the LASTport architecture. Support for the topology shown in Figure 4–1 might be viewed as a network layer function, but because of the simplifying assumptions made, it is represented here as a function in the transport layer.

4.1.1 Path Maintenance

Path maintenance is the process of determining which paths remain viable between a client and server after the solicitation process completes. A circuit is maintained as long as messages arrive on a path during a timed interval.

The client and server follow the same protocol for maintaining the circuit topology. In general, the client Circuit layer builds a list of available paths based on the responses from the solicitation process and load-balances the packets over all paths.

Path failures are detected at either the client or server Association layer when a transaction fails to complete because of missing packets. The Association layer informs the Circuit layer that a failure occurred, forcing the Circuit layer to perform a path maintenance procedure.

The Circuit layer sends a special Solicit Request message (with SERVICE_CLASS equal to zero) to the destination node on all available paths, and the destination node responds. The source node rebuilds the topology based on the responses to the special solicitation. To recover from the error, the entire failed transaction is repeated.

Note that LASTport path maintenance does not involve identifying the failed path and resending a particular packet. The LASTport protocol resolicits the remote node to identify currently available paths and resends the transaction on those paths. The LASTport protocol takes no action to restore a path or to test a path: any message that arrives at its destination confirms that a path is available. Occasionally, new paths are discovered as advertisement messages arrive from nodes with active associations.

4.1.2 Path Availability

By supporting multiple paths between client and server, the LASTport Circuit layer achieves the following goals:

- Increases availability by allowing messages to arrive at the destination over multiple paths. Individual paths are free to transmit again as soon as they are available and can fail without affecting applications.
- Increases bandwidth between any single node and the network by transmitting messages concurrently, using all adapters through which the destination node can be reached.
- Decreases latency of individual transaction operations by splitting messages into segments that travel across all available paths.

As shown in Figure 4–1, multiple paths are created in the following ways:

- A client or server can be redundantly connected to the LAN. For example, a client is redundantly connected when it has two adapters connected to the same LAN. If either adapter fails, communications can continue.
- Clients and servers can be connected redundantly to separate LAN segments. For example, a server has one adapter on each of two LAN segments. If either adapter fails, or if either segment fails, communications can continue.

Each LASTport circuit is associated with all possible paths over which messages can be transmitted successfully to the destination node, so that a node can receive messages from remote nodes that have different data link source addresses. For instance, in Figure 4–1, messages transmitted from node X to node Y can be received from LAN address 3 or address 6.

4.2 Circuit States

A circuit can be in one of the states described in Table 4–1. The circuit can change state when local events occur or when it receives a message from a remote node.

Table 4–1 Circuit States

State	Description
Halted	A Circuit is either inactive or does not exist.
Started	A Circuit is being started. The client node is waiting for the server to acknowledge a Start message.
Stack Sent	A Circuit is being started. The server node has sent a Stack message and is waiting for a Run message to enable circuit transition to the Running state.
Running	A Circuit is active. Associations can use the circuit.

4.3 Circuit Event Processing

The Association layer and the Circuit layer communicate with events. The Association layer declares events to the Circuit layer to start and stop circuits and to perform path maintenance. The Circuit layer declares events to the Association layer to indicate circuit up and circuit down events. By processing events, a circuit passes through the states defined in Table 4–1.

Because the Circuit layer cannot detect transaction or path failures, the Association layer must notify the Circuit layer to perform path maintenance. When a transaction fails more than once, the Association layer notifies the Circuit layer with an *AsnTransFailure* event on a particular circuit. This event causes the Circuit layer to perform path maintenance, verifying all paths between the client and server.

Table 4–2 describes events processed by the Circuit layer; Table 4–3 describes events generated by the Circuit layer and passed to the Association layer.

Table 4–2 Events Processed by the Circuit Layer

Event	Description
CircCreate	Declared by the Association layer to create a circuit to a server. If a circuit to the targeted server already exists, the Circuit layer need not create the circuit.
CircTerminate	Declared by the Association layer to indicate that an association is finished using a circuit. Once all the associations using a circuit delcare a halt event, the Circuit layer can terminate the circuit.
CircRunMsgSend	Declared by the client Association layer to send a Run message.
CircCreateTimeout	Indicates that a Start message must be sent to the server. The server failed to send a Stack message in response to a previous Start message.
CircKeepaliveTimeout	Indicates that the client Circuit layer must send a Solicit Request message (with SERVICE_CLASS equal to zero) to the server. The server responds with a Solicit Response message to indicate that it received the request. The purpose of this exchange is for the client and server to acknowledge that they are both still up. This event occurs only when the circuit has been idle.
CircTimeout	Indicates that no messages have been received on this circuit in the time set by the negotiated progress timer. The circuit is terminated.
CircPathMaintReq	Declared by the client Association layer to indicate that a Data Request message did not generate a Data Response or Resync Response message in the negotiated time limit. This event causes path maintenance to occur on the circuit.
CircPurgeAllPaths	Declared by the CircRcv process to direct the Circuit layer to delete all known paths.

Table 4–3 Events Generated by the Circuit Layer

Event	Description
CircStartMsgRcv	Generated when a Start message is received.
CircStackMsgRcv	Generated when a Stack message is received in response to a Start message.
CircStopMsgRcv	Generated when a Stop message is received. A Stop message terminates the circuit and all associations using the circuit.

(continued on next page)

Table 4–3 (Cont.) Events Generated by the Circuit Layer

Event	Description
CircRunMsgRcv	Generated when a Run message is received at the Circuit layer. Run messages are passed to the association layer.
CircSolReqMsgRcv	Generated when a Solicit Request message is received with SERVICE_CLASS equal to zero.
CircAdvMsgRcv	Generated when an Advertisement message is received with SERVICE_CLASS equal to zero.
CircSolRspMsgRcv	Generated when a Solicit Response message is received with SERVICE_CLASS equal to zero.
AsnCircuitUp	Notifies the Association layer that a circuit is up.
AsnCircuitDown	Notifies the Association layer that a circuit is down.

4.3.1 Circuit Timers

The Circuit layer uses timers for error control. Timers are used to retry delivering the data and to abort an attempt to communicate between the client and server. Table 4–4 describes circuit timers events.

Table 4–4 Circuit Layer Timers

Timer	Function
CircCreateTimer	Notifies the client when to send a Start message to the server.
CircKeepaliveTimer	Notifies the client when to send a Solicit Request message to the server in order to maintain connectivity. Note that this timer is one-third of the PROGRESS_TIMER value negotiated in the Start/Stack message exchange.
CircDownTimer	Notifies the client that a circuit has timed out.

4.3.2 Circuit Processes

To manage state transitions, the Circuit layer uses the following processes:

- CircRcv
- CircEvent
- CircTimer
- CircPathMaintReq
- CircPathMaintRsp

Sections 4.3.2.1 through 4.3.2.5 describe these processes. he processes rely on the Circuit State Block (CSB) data structure, which contains the state and context of each circuit. Note that the accompanying pseudocode examples do not provide all details and are intended only as rough guidelines.

4.3.2.1 CircRcv Process

The CircRcv process is called by the Data Link layer receive process to deliver LASTport packets to the Circuit layer. The process performs actions shown in the following pseudocode example.

Note

Some memory optimizations can be implemented when storing other nodes in the NDB cache. For example, client-only implementations need only store servers, because they never connect to other clients.

```
void CircRcv(message, circuitId, nodeId)
CircuitMessage *message;
char circuitId;
char nodeId[6];
{
    NDB *ndb;
    SolicitMessage solicitmessage = message + sizeof(circuitMessageHeader);
    CSB *csb;

    /* For each message received: */

    NDB = findNDB(nodeId);
    if (ndb == 0)
    {
        CreateAndInitNdb(nodeId, message, CSB);
    }
    else
    {
        /* Validate circuit header fields with NDB values. */

        if (CheckCircuitHeader(ndb, message) == FALSE)
        {
            CircEvent(CircHalt,message, nodeId, csb);
            return;
        }
    }
    csb = ndb–>csb;
    ResetTimer(csb–>CircDownTimer);
    ResetTimer(csb–>CircCreateTimer);
    AddPathToCircuit(circuitId, csb);

    /* For each message type, call CircEvent process with appropriate event. */

    switch (message–>type)
    {
      case Start:
        CircEvent(CircStartMsgRcv, message, nodeId, csb);
        break;
      case Stack:
        CircEvent(CircStackMsgRcv, message, nodeId, csb);
        break;
```

```
        case Run:
          CircEvent(CircRunMsgRcv, message, nodeId, csb);
          break;

        case Solicit:
          if (solicitmessage–>flags & PurgeAllPaths) == 0)
            CircEvent(CircSolReqMsgRcv, message, nodeId, csb);
          else
            CircEvent(CircPurgeAllPaths, message, nodeId, csb);
          break;

        case Advertisement:
        case SolicitResponse:
          if (csb–>commandFailure > 0)
          {
            csb–>commandFailure = 0;
            AsnClientRcv(AsnCircuitUp, NULL, nodeId, csb–>ndb);
          }
          break;
      }
    }
```

4.3.2.2 CircEvent Process

The CircEvent process is called by the CircRcv process and recursively to generate the circuit protocol.

The process uses the current state and the events shown in Table 4–5 to identify the new state. (For example, if the current state is Started and the event is *CircHalt*, the new state is Halted.) The process then extracts the new state, updates the CSB with this state, and executes actions indicated by the callout (for example, ❺). These actions are described following the table. A dash (—) indicates that no action is required.

The CircEvent process performs actions shown in the following pseudocode example.

```
void CircEvent(event, message, nodeId, csb)
int event;
SolicitMessage *message;
char nodeId[6];
CSB *csb;
{
    int newState;
    int (*actionRoutine)( );

     /*
      * Using state table, retrieve new state and action routine.
      * Call action route to start or continue protocol message exchanges.
      */

    newState = circuitStateTable[event, csb–>state].newState;
    actionRoutine = circuitStateTable[event, csb–>state].actionRoutine;
    csb–>state = newState;
    (*actionRoutine)(message, csb, ...);
}
```

4.3.2.3 CircTimer Process

The CircTimer process is called by the operating system once each second. The process performs actions shown in the following pseudocode example.

```
void CircTimer( )
{
    CSB *csb = csbListHead;

     /*
      * For each circuit starting or running, perform maintenance
      * on its timers.
      */

    while (csb != NULL)
    {
       if ((csb–>circuitState == Started)
          or (csb–>circuitState == StackSent))
       {
          csb–>CircCreateTimer--;
          if (csb–>CircCreateTimer == 0)
          {
             csb–>StartRetransmitCounter ++;
             if (csb–>StartRetransmitCounter == csb–>StartRetransmitLimit)
             {
                CircEvent(CircStartFailed, message, nodeId, csb);
             }
             else
             {
                CircEvent(CircCreateTimeout, message, nodeId, csb);
             }
          }
       }

       if (csb–>circuitState == Running)
       {
          csb–>CircKeepaliveTimer--;
          if (CircKeepaliveTimer == 0)
          {
             CircEvent(CircKeepaliveTimeout, message, nodeId, csb);
          }
          csb–>CircDownTimer--;
          if (CircDownTimer == 0)
          {
             CircEvent(CircTimeout, message, nodeId, csb);
          }
       }
       csb = csb–>next;
    }
}
```

4.3.2.4 CircPathMaintReq Process

The CircPathMaintReq process is called by the CircEvent process if path maintenance is required by the Association layer after several transaction failures. The process performs actions shown in the following pseudocode example.

```
void CircPathMaintReq(event, message, nodeId, csb)
int event;
char *message;
char nodeId[6];
CSB *csb;
{
    SolicitMessage *solicitMessage;

     /*
      * CircRcv process resets timer or forces Path Maintenance Response.
      */
    csb–>CircDownTimer = PathMaintTimeout;
    csb–>commandFailure ++;
    if (csb–>commandFailure > 1)
    {
       PurgeAllPaths(csb);
       csb–>solicitId ++;
        /*
         * Table 6–8 describes Solicit message fields.
         */
       solicitMessage = constructSolicitMessage(csb, PurgeAllPaths);

       SendSolicitOnAllPaths(solicitMessage, LastportMulticastAddress);
    }
}
```

4.3.2.5 CircPathMaintRsp Process

The CircPathMaintRsp process is called by the CircEvent process to generate messages that maintain the circuit topology. The process performs actions shown in the following pseudocode example.

```
void CircPathMaintRsp(event, message, nodeId, csb)
int event;
SolicitMessage *message;
char nodeId[6];
CSB *csb;
{
    AdvertisementMessage *advertisementMessage;

     /*
      * Compare Solicit ID stored in CSB with Solicit ID in received message.
      */
    if (message–>solicitId = = csb–>solicitId)
    {
        AddPathToCircuit(message, csb);
    }
    else
    {
        csb–>solicitId = message–>solicitId;
```

```
            PurgeAllPaths(csb)
            AddPathToCircuit(csb);
            advertisementMessage = constructAdvertisementMessage(csb);

            SendMessageOnAllPaths(advertisementMessage,
    LastportMulticastAddress);
        }
    }
```

4.3.3 Circuit State Transitions

An event can, but does not always, change a circuit's state. Using the states defined in Table 4–1 and the events defined in Table 4–2, Table 4–5 shows the new states that can result from the occurrence of each event in each state.

Table 4–5 Circuit Events and Resulting State Transitions

	Current State			
	Halted	**Started**	**Stack Sent**	**Running**
Command Events				
CircCreate	Started ❶	Started ⓭	Stack Sent ⓭	Running ❽
CircTerminate	—	Started ❸	Stack Sent ❸	Running ❸
CircHalt	Halted	Halted ❺	Halted ❺	Halted ❺
Message Events				
CircStartMsgRcv	Stack Sent ❷	Stack Sent ⓮	Stack Sent ⓮	Halted ❺
CircStackMsgRcv	—	Running ❾	Running ❾	Running
CircRunMsgRcv	—	Started	Running ⓯	Running ❹
CircSolMsgRcv	—	—	—	Running ⓱
CircAdvMsgRcv	—	—	—	Running
CircSolRspMsgRcv	—	—	—	Running
CircStopMsgRcv	—	Halted ⓫	Halted ⓫	Halted ⓫
CircRunMsgSend	—	—	—	Running ❻
Timer Events				
CircCreateTimeout	—	Started ⓾	Stack Sent ⓮	Running
CircKeepaliveTimeout	—	—	Stack Sent ⓬	Running ⓬
CircTimeout	—	—	Halted ❺	Halted ❺
Path Maintenance Events				
CircPathMaintReq	—	—	—	Running ❼
CircStartFailed	—	Halted ⓫	—	—
CircPurgeAllPaths	—	—	—	Running ⓰

Actions for the state transitions shown in Table 4–5 are as follows:

❶ Allocate a CSB for the new circuit.

Assign a unique circuit ID to represent this circuit.

Bind the association to the CSB.

Increment the reference count in the CSB.

Construct a Start message, initializing appropriate message fields (see Table 6–2).

Construct a circuit message header for the Start message, initializing appropriate message fields (see Table 6–1) and send the message to the target node.

Enable the CircCreateTimer to indicate when a Start message needs to be transmitted.

❷ Initialize a CSB.

Increment the reference count in the CSB.

Construct a Start Acknowledgement (Stack) message, initializing appropriate message fields (see Table 6–2).

Construct a circuit message header for the Stack message, initializing appropriate message fields (see Table 6–1), and send the message to the target node.

❸ Decrement the reference count in the CSB.

If the count is zero, call the *CircEvent* process with the *CircHalt* event.

❹ Call the AsnClientRcv process with the *AsnRunMsgRcv* event message and NDB.

❺ Construct a Stop message, initializing appropriate message fields (see Table 6–7).

Construct a circuit message header for the Stop message, initializing appropriate message fields (see Table 6–1), and send the message to the target node.

For each association bound to the circuit, call the *AsnClientRcv* or *AsnServerRcv* process with the *AsnCircuitDown* event.

Return all resources.

❻ Construct a circuit message header for the Run message, initializing appropriate fields (see Table 6–1) and transmit the Run message.

Reset the *CircKeepaliveTimer*.

❼ Call the CircPathMaintReq process.

❽ Bind the association to the CSB.

Increment the reference count on the CSB.

Call the *AsnClientRcv* or *AsnServerRcv* process with the *AsnCircuitUp* event.

❾ For each association bound to the circuit, call the *AsnClientRcv* or *AsnServerRcv* process with the *AsnCircuitUp* event.

Enable the CircDownTimer and CircKeepaliveTimer.

Disable the CircCreateTimer.

❿ Construct a Start message, initializing appropriate message fields (see Table 6–2).

Construct a circuit message header for the Start message, initializing appropriate message fields (see Table 6–1) and send the message to the target node.

Reset the CircCreateTimer.

⓫ For each association bound to the circuit, call the *AsnClientRcv* or *AsnServerRcv* process with the *AsnCircuitDown* event.

Return all resources to the system.

⓬ Construct a Advertisement message, initializing appropriate message fields (see Table 6–8).

Construct a circuit message header for the Advertisement message, initializing appropriate message fields (see Table 6–1), and send the message to the target node.

Send a *CircAdvMsgRcv* advertisement message to the the circuit partner.

⓭ Increment the reference count in the CSB.

Bind the association to the CSB.

⓮ Compare remote SOURCE_NODE_ID from Start message with the local node Id. If the remote SOURCE_NODE_ID is greater than local node Id, construct a Stack message, initializing appropriate message fields (see Table 6–2).

Construct a circuit message header for the Stack message, initializing appropriate message fields (see Table 6–1) and send the message to the target node.

If the remote SOURCE_NODE_ID is less than local node Id, ignore the message.

⓯ For each association bound to the circuit, call the *AsnClientRcv* or *AsnServerRcv* process with the *AsnCircuitUp* event.

Enable/Reset the CircDownTimer.

Call the AsnClientRcv or AsnServerRcv process (described in Chapter 5) with the *AsnRunMsgRcv* event.

⓰ Call the CircPathMaintRsp process.

⓱ Construct a Solicit Response message, initializing appropriate message fields (see Table 6–8).

Construct a circuit message header for the Solicit Response message, initializing appropriate message fields (see Table 6–1) and send the message to the target node.

4.3.4 Circuit State Transition Examples

Sections 4.3.4.1 through 4.3.4.3 contain examples illustrating several common circuit state transitions and actions taken by the client and server during the transitions. The state transitions discussed are as follows:

- Normal circuit start
- Attempt by two clients to connect to each other simultaneously
- Normal circuit terminate

4.3.4.1 Normal Circuit Start

This example describes the state transitions and actions at the client and server during a normal circuit start.

The following activities occur on the client:

1 The Association layer directs the Circuit layer to start a circuit by declaring a *CircStart* event.

2. The client builds a CSB and sends a Start message to the server. The client's circuit changes to the Started state.
3. Later, the client receives a Stack message, processes it, and changes state to Running. The Circuit layer declares an *AsnCircuitUp* event to the Association layer.
4. The Association layer declares a *CircRunMsgSend* event to the Circuit layer to transmit a Run message.

The following activities occur on the server:

1. The server receives a Start message, creates a CSB, sends a Stack message, and changes the circuit state to Stack Sent.
2. The server receives a Run message, changes the circuit state to Running, and passes the Run message to the Association layer.

4.3.4.2 Attempt by Two Clients to Connect to Each Other Simultaneously

This example describes the state transitions and actions that occur when two clients attempt to start circuits to each other at the same time. Note that to prevent a deadlock starting scenario, one node is forced to become the slave end of the circuit. This node sends a Stack message, and the other node (master) ignores Start requests from the slave node. The algorithm uses the SOURCE_NODE_ID in the Circuit Message Header to determine master and slave.

The following activities occur on Client A:

1. The Association layer directs the Circuit layer to start a circuit with Client B by calling the CircEvent process with the *CircCreate* event.
2. Client A builds a CSB and sends a Start message to the server. Client A's circuit changes to the Started state.
3. Later, Client A receives a Start message. Client A compares its SOURCE_NODE_ID with Client B's. Because Client A's is less than Client B's, Client A sends a Stack message and changes state to Stack Sent.
4. Later, Client A receives a Run message and changes state to Running. The Circuit layer declares an *AsnCircuitUp* event to the Association layer.

The following activities occur on Client B:

1. The Association layer directs the Circuit layer to start a circuit with Client A by calling the CircEvent process with a *CircCreate* event.
2. Client B builds a CSB and sends a Start message to the server. Client B's circuit changes state to Started.
3. Later, Client B receives a Start message, Client B compares its SOURCE_NODE_ID with Client A's. Because Client B's is greater than Client A's, Client B ignores the Start message.
4. Later, Client B receives a Stack message and changes state to Running. The Circuit layer declares an *AsnCircuitUp* event to the Association layer.
5. Client B sends a Run message (Connect Request) to Client A.

4.3.4.3 Normal Circuit Terminate

This example describes the state transitions and actions performed to terminate a circuit.

1. The Association layer on the client directs the Circuit layer to terminate a circuit. Because this is the only association using the circuit, the client sends a Stop message to the server.
2. The server receives a Stop message and terminates the circuit.

4.4 Congestion Control

The goal of congestion control is to manage the use of a congested adapter by limiting transmitters to a message-per-second rate. The algorithm used to control congestion across circuits and multiple paths 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 overestimation of the maximum sustained throughput, measured in messages per second, of the data link It is established during the data link initialization phase and remains constant while that data link is in use.
- Current_Message_Rate is a 16 bit integer representing the current receiver based congestion control policy for a data link. 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 32,767 messages per second.

  A Message_Rate is associated with each data link available to the transport. Different Message_Rates may be concurrently enforced based on congestion of the various data links involved.

4.4.1 Congestion Detection

Congestion occurs when a data link is unable to receive all messages directed to its address. When congestion occurs, the implementation voluntarily restricts its use of the data link by enforcing a Message_Rate for that data link to all connected nodes. This Message_Rate restricts the number of Run message segments per second that can be transmitted to the data link. The Message_Rate setting does not affect other data links available to the receiver.

The LASTport protocol periodically performs a data link congestion check. This check determines whether a data link experienced congestion during the previous interval. This check occurs no more than once per second.

If congestion is detected, the congestion control algorithm is enabled.

For example, when a node detects congestion, the following events occur:

1. The SB field in the circuit message header is set to one (see Chapter 6).
2. The RATE_VALUE field in the circuit message header is set to the number of Run message segments per second that can be transmitted to the data link, enforcing the congestion control.
3. The congested data link is identified by the source address of the message containing the new Message_Rate value.

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

4.4.2 Congestion Control Algorithm

Initially, a data link is assumed to be uncongested. When a data link is uncongested, the receiver enforces a Message_Rate of zero. A Message_Rate of zero is architecturally defined as a no-congestion policy.

When the LASTport transport initializes, it computes two congestion variables: Maximum_Message_Rate and Current_Message_Rate:

- The Maximum_Message_Rate is computed by multiplying the a controller specific value by 2.
- The Current_Message_Rate is initially set to the Maximum_Message_Rate and is modified when congestion is detected.

When congestion is detected, the following events occur:

1. The LASTport protocol sets the Current_Message_Rate to one half the Maximum_Message_Rate and recomputes the Message_Rate.
2. This new Message_Rate is copied to the circuit header RATE_VALUE field.
3. The SB bit is set to one for all traffic over that data link.
4. When the rate is successfully communicated to the remote node, that is, when LAST_RATE_VALUE equals RATE_VALUE, the SB bit is cleared until a new Message_Rate is established.
5. If congestion persists, the Current_Message_Rate is halved and a new Message_Rate is communicated in normal traffic.

This process continues 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 following events occur:

1. The Current_Message_Rate is incremented by 10 messages per second.
2. A new Message_Rate is established and communicated in normal traffic as described above.

When the Current_Message_Rate equals the Maximum_Message_Rate, the congestion algorithm is disabled and a Message_Rate of zero is communicated in normal traffic.

4.4.3 Congestion Control Policy

Effective congestion control must ensure that minimal per-message overhead occurs when a Message_Rate is not being set. The LASTport architecture facilitates congestion control implementation by using the signed bit of the word containing the Message_Rate being set. If the bit is not set, the receiver can ignore the Message_Rate specified in the RATE_VALUE, regardless of its absolute value.

A transmitter always sets the LAST_RATE_VALUE to the Message_Rate most recently established for the remote node's data link. A receiver always checks whether the LAST_RATE_VALUE equals the Message_Rate currently in effect. If the fields are not equal, the LASTport protocol causes the correct rate to be communicated in return traffic by performing the following operations:

1. Sets the SB field to one

2 Sets the RATE_VALUE field to the applicable Message_Rate

Table 4–6 describes the effect on Message_Rate as congestion is detected on the sample network shown in Figure 4–2. The network has one server with two data links and six associated clients, each with a single data link.

Figure 4–2 Network with One Server and Six Clients

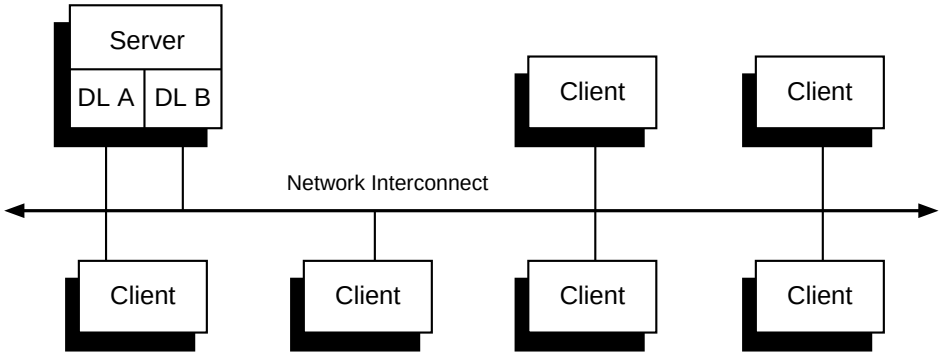


Table 4–6 Congestion Example

Network Event	Maximum_Message_Rate		Current_Message_Rate		Message_Rate	
	DL A	DL B	DL A	DL B	DL A	DL B
Transport Initialization ❶	700	700	∞	∞	∞	∞
Congestion on DL A ❷	700	700	350	∞	58	∞
Next second ❸	700	700	360	∞	60	∞
Next second ❹	700	700	370	∞	61	∞
Next second ❹	700	700	380	∞	63	∞
Next second ❹	700	700	390	∞	65	∞
Second *n* ❺	700	700	∞	∞	∞	∞

❶ At transport initialization, the Maximum_Message_Rate is set to 700 packets per second. Typically, this value should be set to twice the known capacity of the data link controller. The table shows a controller capable of forwarding 350 packets per second. The Current_Message_Rate and Message_Rate are set to infinite.

❷ Congestion is detected on controller A.

❸ The LASTport protocol starts congestion control on controller A, but the B controller is unaffected. Client systems are allowed to transmit only 60 Run message segments to the server on controller A, but they can transmit any number of segments to controller B.

❹ Each second thereafter, the Current_Message_Rate is incremented by 10 and a new Message_Rate is created.

❺ The process continues until congestion is again detected, or until the Current_Message_Rate equals the Max_Message_Rate. The Message_Rate is then set to infinite, and congestion control is disabled.

When a client or a server detects that a Message_Rate is being enforced by the remote data link, the client or server immediately computes two variables for the data link enforcing the Message_Rate:

- Remote_Message_Rate. The Remote_Message_Rate is set to the RATE_VALUE.
- Remote_Rate_Left. The Remote_Rate_Left is initially set to the Remote_Message_Rate and can be modified as described below.

All message traffic is multiplexed through the circuit; congestion control policy is implemented at the circuit level.

For example, when higher layers attempt to transmit Run messages to a remote node, the Circuit layer performs following operations:

1. Selects a remote data link to receive the transmitted message.
2. If the remote data link is enforcing a Message_Rate, the Circuit layer checks whether the Remote_Rate_Left for that data link is zero and performs the following operations:
   - If a positive rate exists, the Circuit layer decrements the Remote_Rate_Left value and transmits the message.
   - If Remote_Rate_Left is equal to zero, the Circuit layer selects another remote data link. If the remote data link is congested, the Circuit layer checks the Remote_Rate_Left field until it finds a data link with a positive rate.
   - If it cannot find a remote data link with a positive Remote_Rate_Left, the Circuit layer buffers the message locally.
3. Once per second, the Circuit layer sets Remote_Rate_Left to Remote_Message_Rate for each congested remote data link.
4. At this time, the Circuit layer attempts to transmit all locally buffered messages.

5
Association Operations

The Association layer performs the following tasks:

- Pairs request messages with response messages at the association interface
- Performs any necessary segmentation and reassembly of the request and response messages into request and response message streams
- Detects and optionally corrects duplicate or lost requests and responses (client only)
- Multiplexes client transactions over the Circuit layer
- Signals the Circuit layer when transaction faults are detected
- Manages the association interface to system applications

This chapter describes Association layer operations. Topics include the following:

- Connection services
- Association states
- Association event processing

5.1 Connection Services

The Association layer provides a simple mechanism to connect a client to a server. After locating the server, the client system application instructs the Association layer to connect to the server. The connect event causes the Association layer to perform the following operations:

1. Instruct the Circuit layer to establish a circuit with the server node.
2. After the circuit is established, transmit Connect Request messages to the server at the frequency and rate instructed by the client system application.
3. Notify the client system application when either of the following events occurs:
   - The client receives a Connect Response message from the server indicating that the circuit is up.
   - The connect process times out.

Once the circuit is up, an association is started between client and server, and transactions can be executed. Section 5.2 describes association states; Section 5.3 describes association event processing.

5.2 Association States

Every association has a client and a server. Because each member of the association has different responsibilities, client and server have different events and can be in different states. Table 5–1 describes association states.

Table 5–1 Association States

State	Description
Halted	An association is either inactive or does not exist.
Started	An association is starting. The association is waiting for the circuit to be declared up and for the server to respond to the Connect Request message.
Running	An association is up; the system application can execute transactions.
Stopped	An association is disconnecting (being torn down).

5.3 Association Event Processing

An association passes through the defined states by processing events. Events are either locally generated or result from receiving a message.

Some events, such as *AsnConnect*, carry data as parameters when the event is declared. For example, *AsnConnect* includes the connect data as a parameter.

Table 5–2 lists and describes events processed by the client Association layer. Table 5–3 lists and describes events processed by the server Association layer. Table 5–4 lists and describes events generated by the Association layer and passed to the Circuit layer and the system application.

Table 5–2 Events Processed by the Client Association Layer

Event	Description
AsnConnect	Declared by the system application to create an association to a server.
AsnConRspMsgRcv	Generated when a Connect Response message is received from a client.
AsnConTimeout	Indicates that a Connect Request message timed out. The Connect Request message is sent multiple times before a *AsnConFailure* event is generated.
AsnConFailure	Generated when a Connect Request attempt fails to generate a Connect Response from a server.
AsnRunMsgRcv	Generated when any Run message is received. The Circuit layer generates this event to the Association layer. The Association layer parses the Run message and declares an event to itself based on the Run message type.
AsnDisconnect	Generated by a system application to stop an association.
AsnDisconTimeout	Indicates that a Disconnect Request message timed out. The Disconnect Response message is sent multiple times before an *AsnDisConFailure* event is generated.
AsnDisconFailure	Generated when a Disconnect Request attempt fails to elicit a Disconnect Response from a server.

(continued on next page)

Table 5–2 (Cont.) Events Processed by the Client Association Layer

Event	Description
AsnDisconRspMsgRcv	Generated when a Disconnect Response message is received.
AsnTransSend	Generated by a system application to initiate a transaction to a server.
AsnDataRspMsgRcv	Generated when a client receives a Data Response message.
AsnTransTimeout	Indicates that a transaction attempt failed to generate a transaction response from a server. The transaction is sent multiple times before a *AsnTransFailure* event is generated to the system application.
AsnTransFailure	Generated to indicate that a transaction failed. Once a transaction fails, the association must be aborted.
AsnResyncMsgRcv	Generated when a client receives a Resync Response message. The Resync Response message acknowledges a transaction if a server does not generate a Data Response message.
AsnCircuitDown	Generated by the Circuit layer to notify an association that a circuit is down.
AsnCircuitUp	Generated by the Circuit layer to notify an association that a circuit is up.

Table 5–3 Events Processed by the Server Association Layer

Event	Description
AsnRunMsgRcv	Generated by the Circuit layer when it receives a Run message and passed to the Association layer. The Association layer parses the Run message and declares an event to itself based on the Run message type.
AsnConReqMsgRcv	Generated when the server receives a Connect Request message from a client.
AsnConRspMsgSend	Generated to send a Connect Response message.
AsnDisconRspMsgSend	Generated to send a Disconnect Response message.
AsnDisconReqMsgRcv	Generated when the server receives a Disconnect Response message.
AsnDataMsgRcv	Generated when the server receives a transaction.
AsnTransRspMsgSend	Generated by a system application to respond to a transaction.
AsnResyncTimeout	Indicates that a Resync Response message must be sent to prevent a transaction attempt from failing. A server sends a Resync Response message to notify the client that it received the transaction and is processing it.
AsnCircuitDown	Generated by the Circuit layer to notify an association that a circuit is down.

Table 5–4 Events Generated by the Association Layer

Event	Description
AppAsnUp	Declared to a system application to indicate that the association is up.
AppAsnStopped	Declared to a system application to indicate that the association terminated correctly.
AppAsnAborted	Declared to a system application to indicate that the association terminated prematurely.
AppTransReq	Declared by the server Association layer to indicate that a transaction was received and needs to be processed.
AppTransComplete	Declared by the client Association layer to indicate that a transaction completed.
AppConFailure	Declared to a system application to indicate that a connect attempt failed.
AppDisconFailure	Declared to a system application to indicate that a disconnect attempt failed.
AppTransFailure	Declared to a system application to indicate that a transaction failed. When a transaction fails, the association is aborted. To continue communicating with the server, the system application must create a new association to the server.
CircCreate	Creates a circuit to a server. If a circuit already exists to the targeted server, the Circuit layer need not create the circuit.
CircTerminate	Indicates that an association is finished using a circuit. Once all the associations using a circuit delcare a halt event, the Circuit layer terminates the circuit.
CircRunMsgSend	Declared by the client Association layer to send a Run message.
CircPathMaintReq	Declared by the client Association layer to indicate that a Data Request message did not generate a Data Response or Resync Response message in the negotiated time limit. This event causes path maintenance to occur on the circuit.

5.3.1 Association Timers

The Association layer uses timers for error control. Timers manage attempts to retry delivering data and can abort attempts to communicate between the client and server. Table 5–5 describes Association layer timers. For information on timer policies, refer to Section 2.1.4.

Table 5–5 Association Layer Timers

Timer	Function
AsnConnectResponseTimer	Notifies the client when to send a Connect Request message to the server.
AsnDisconnectResponseTimer	Notifies the client when to send a Disconnect Request message to the server.
AsnTransactionResponseTimer	Notifies the client that transaction processing is delayed. This timer has two values: **1** The AsnTransactionResponseShortTimer, which is an initial time limit for receipt of a request message acknowledgment or transaction response message. **2** The AsnTransactionResponseLongTimer value, which is subsequently used as a time limit for completion of the transaction, but can be reset by the server using Resync Response messages.
AsnResyncResponseTimer	Notifies the server when to send a Resync Response message to the client to avoid a retransmission of the current transaction.

5.3.2 Association Client Processes

To manage state transitions, the Association layer uses the following client processes:

- AsnClientRcv
- AsnClientEvent
- AsnClientTimer
- AsnClientTransSend
- AsnClientTransSendRetry
- AsnClientTransRcv

Sections 5.3.2.1 through 5.3.2.6 describe these processes, which rely on the following data structures:

- Association State Block (ASB), which contains the state and context of each association.
- Session State Block (SSB), which is the Handle used by the system application to reference an association.
- Transaction Context Block (TSB), which contains the state and context for each transaction in an association.

Note that the following process pseudocode examples do not provide all details and are intended only as rough guidelines.

5.3.2.1 AsnClientRcv Process

The AsnClientRcv process is called by the CircEvent process to notify the Association layer when messages arrive and to effect appropriate circuit state transitions. The process performs actions shown in the following pseudocode example.

```
void AsnClientRcv(event, message, NDB)
int event;
RunMessage *runMessage;
NDB *ndb;
{
     /*
      * Association State Block holds state and context for each association.
      */
     ASB *asb

      /* For each event received: */

     asb = locateAsb(runMessage –>asbId);
     if (event == AsnRunMsgRcv)
     {

      /*
       * Parse Run message header, verifying fields in Table 6–10.
       * For each message subtype, call AsnClientEvent process
       * with appropriate event.
       */

        switch (runMessage–>subtype)
        {
           case ConnectResponse:
              AsnClientEvent(AsnConRspMsgRcv, message, ndb–>nodeId, asb);
              break;

           case DataResponse:
              AsnClientEvent(AsnDataRspMsgRcv, message, ndb–>nodeId, asb);
              break;

           case ResyncResponse:
              AsnClientEvent(AsnResyncMsgRcv, message, ndb–>nodeId, asb);
              break;

           case DisconnectResponse:
              AsnClientEvent(AsnDisconReqMsgRcv, message, ndb–>nodeId, asb);
              break;
        }
     }

     if ((event == AsnCircuitUp) || (event == AsnCircuitDown))
     {
          asb = asbListHead;
          while (asb != NULL)
          {
              AsnClientEvent(event, message, ndb–>nodeId, asb);
              asb = asb–>next;
```

```
            }
        }
    }
```

5.3.2.2 AsnClientEvent Process

The AsnClientEvent process is called by the AsnClientRcv process and recursively to generate the association protocol.

The AsnClientEvent process uses the current state and the events shown in Table 5–6 to identify the new state. (For example, if the current state is Halted and the event is *AsnConnect*, the new state is Started.) The process extracts the new state, updates the ASB with this state, and executes actions indicated by the callout (for example, ❶). These actions are described following the table. A dash (—) indicates that no action is required.

The AsnClientEvent process performs actions shown in the following pseudocode example.

```
void AsnClientEvent(event, message, nodeId, asb)
int event;
Message *message;
ASB *asb;
{
    int newState;
    int (*actionRoutine)( );

     /*
      * Using state table, retrieve new state and action routine.
      * Call action route to start or continue protocol message exchanges.
      */
    newState = AsnClientstateTable[event, asb–>state].newState;
    actionRoutine = AsnClientStateTable[event, asb–>state].actionRoutine;
    asb–>state = newState;
    (*actionRoutine)(message, asb, ...);
}
```

5.3.2.3 AsnClientTimer Process

The AsnClientTimer process is called by the operating system once each second. The process performs actions shown in the following pseudocode example.

```
void AsnClientTimer( )
{
    ASB *asb = asbListHead;
    while(asb != NULL)
    {
       if (asb–>state == Started)
       {
          asb–>AsnConnectResponseTimer--;
          if (asb–>AsnConnectResponseTimer == 0)
          {
             asb–>AsnConnectRetransmitCounter++;
             if (asb–>AsnConnectRetransmitCounter ==
                 asb–>AsnConnectRetransmitLimit)
             {
                 AsnClientEvent(AsnConFailure, NULL, NULL, asb);
             }
             else
```

```
                {
                    AsnClientEvent(AsnConTimeout, NULL, NULL, asb);
                }
            }
        }

        if (asb–>state == Running)
        {
            for (i = 1; i < asb–>AsnMaxSlots; i++)
            {
                if (asb–>slot[i] == 1)
                {
                    asb–>AsnTransactionResponseTimer[i]--;
                    if (asb–>AsnTransactionResponseTimer[i] == 0)
                    {
                        asb–>AsnTransactionRetransmitCounter[i]++;
                    }
                    if (asb–>AsnTransactionRetransmitCounter[i] ==
                            asb–>AsnTransactionRetransmitLimit[i])
                    {
                        AsnClientEvent(AsnTransFailure, NULL, NULL, asb);
                    }
                    else
                    {
                        AsnClientEvent(AsnTransTimeout, NULL, NULL, asb);
                    }
                }
            }
        }

        if (asb–>state == Stopped)      /* For each association: */
        {
            asb–>AsnDisconnect ResponseTimer--;
            if (asb–>AsnDisconnect ResponseTimer == 0)
            {
                asb–>AsnDisconnectRetransmitCounter++;
                if (asb–>AsnDisconnectRetransmitCounter ==
                        asb–>AsnDisconnectRetransmitLimit)
                {
                    AsnClientEvent(AsnDisconFailure, NULL, NULL, asb);
                }
                else
                {
                    AsnClientEvent(AsnDisconTimeout, NULL, NULL, asb);
                }
            }
        }
    asb = asb–>next;
    }
}
```

5.3.2.4 AsnClientTransSend Process

The AsnClientTransSend process is called by the AsnClientEvent process to send a transaction. The process performs actions shown in the following pseudocode example.

```
void AsnClientTransSend(dataRequest, dataRequestLength, asb)
char *dataRequest;
int dataRequestLength;
ASB *asb;
{
    int slot;
    tsb *tsb;

     /* Serialize all transaction requests in case all slots are busy. */
    QueueTransaction(asb);
    tsb = allocateTsb( );
    slot = FindFreeSlot( );
    if (slot != 0)
    {
       DequeueTransaction(asb);
       AsnClientTransmitTransaction(dataRequest, dataRequestLength,
           tsb, asb, slot);
    }
}

void AsnClientTransmitTransaction(dataRequest, dataRequestLength,
          tsb, asb)
char *dataRequest;
int dataRequestLength;
TSB *tsb;
ASB *asb;
{
    char *segment;
    char *message;

    asb–>sequenceNumber[slot]++; †

     /*
      * Based on negotiated segment size:
      */
    tsb–>maximumSegments = (dataRequestLength/asb–>segmentSize) + 1;
    DivideIntoSegments(dataRequest, dataRequestLength,
        asb–>segmentSize, tsb);

    segment = tsb–>segmentList;
    while (segment != NULL)
    {
       message = constructDataRequestMsgHeader(segment);
       /* See Table 6–14. */

       message = constructRunMsgHeader(message);
       /* See Table 6–1. */
```

† A transaction sequence number is associated with a particular slot and is independent of all other sequence numbers on all other slots.

```
            CircEvent(CircRunMsgSend, message, asb–>csb–>nodeId, asb–>csb);
            AsnEnableTimer(AsnTransactionResponseTimer);
            segment = segment–>next;
        }
        asb–>flags  | = ActiveTransaction;
    }
```

5.3.2.5 AsnClientTransSendRetry Process

The AsnClientTransSendRetry process is called by the AsnClientEvent process to retry sending a transaction on failure of an earlier attempt. Using the same inputs as the previous transaction attempt, the process performs actions shown in the following pseudocode example.

```
void AsnClientTransSendRetry(asb, tsb)
ASB *asb;
TSB *tsb;
{
    CircEvent(CircPathMaintReq, NULL, asb–>csb–>nodeId, csb);
    AsnClientTransmitTransaction(tsb–>dataRequest,
      tsb–>dataRequestLength, tsb, asb);
}
```

5.3.2.6 AsnClientTransRcv Process

The AsnClientTransRcv process is called by the AsnClientEvent process to reassemble a transaction response. The process performs actions shown in the following pseudocode example.

```
void AsnClientTransRcv(dataResponseSegment, asb)
char *dataResponseSegment;
ASB *asb;
{
    tsb *tsb;

    /* Verify that data response sequence number
         matches current request, discarding orphan responses. */
    if (CurrentSequenceNumber(dataResponseSegment) = = TRUE)
    {
        tsb = LocateTsb(dataResponseSegment);

        CopySegmentToResponseBuffer(dataResponseSegment,
          tsb–>responseBuffer); ‡
        if (TransactionResponseComplete(tsb) = = TRUE)
        {
            sysAppClientRcv(AppTransComplete, tsb–>responseBuffer, asb–>ssb);
            AsnDisableTimer(AsnTransResponseTimer);

            /* Check whether transactions are waiting to be executed.
                 If so, initiate them on this free slot. */
            if (TransactionsQueued(asb))
                 StartNextTransaction(asb);
        }
    }
}
```

‡ This buffer is supplied by the client system application when the transaction is initiated (see Section 2.4.2).

5.3.3 Association Client State Transitions

Table 5–6 shows association client events and resulting state transitions. Use the current state and the event to determine the new state.

Table 5–6 Association Client Events and Resulting State Transitions

	Current State			
	Halted	**Started**	**Running**	**Stopped**
Command Events				
AsnConnect	Started ❶	—	—	—
AsnDisconnect	—	Halted ❾	Stopped ❸	—
AsnCircuitUp	—	Started ❷	Running	Stopped
AsnCircuitDown	—	Halted ❺	Halted ❺	Halted ❺
AsnConFailure	—	Halted ❹	—	—
AsnDisconFailure	—	—	—	Halted ❼
AsnTransSend	—	—	Running ❽	—
AsnTransFailure	—	—	Halted ⓫	—
Message Events				
AsnConRspMsgRcv	—	Running ❻	Running	Stopped
AsnDataRspMsgRcv	—	—	Running ❿	Stopped
AsnResyncMsgRcv	—	—	Running ⓬	Stopped
AsnDisonReqMsgRcv	—	—	Halted ⓭	Halted ⓭
Timer Events				
AsnConTimeout	—	Started ❷	—	—
AsnDisconTimeout	—	—	—	Stopped ❸
AsnTransTimeout	—	—	Running ⓮	Stopped

Actions for the state transitions shown in Table 5–6 are as follows:

❶ Initialize an ASB for the new association.

Assign a unique association id to represent this association.

Bind the client system application to the ASB.

Call the CircEvent process with the *CircCreate* event.

❷ Construct a Connect Request message, initializing appropriate message fields (see Table 6–12).

Construct an association Run message header for the message, initializing appropriate message fields (see Table 6–10), and call the CircEvent process with the *CircRunMsgSend* event.

Activate and initialize the ConnectResponseTimer in the ASB for this association.

❸ Construct a Disconnect Request message, initializing appropriate message fields (see Table 6–12).

Construct an association Run message header for the message, initializing appropriate message fields (see Table 6–10), and call the CircEvent process with the *CircRunMsgSend* event.

Activate and initialize the ConnectResponseTimer in the ASB for this association.

❹ Call the sysAppClientRcv process with the *AsnConFailed* event and SSB.

Call the CircEvent process with a *CircTerminate* event.

Return all resources.

❺ Call the sysAppClientRcv process with the *AppAsnAborted* event and SSB.

Return all resources.

❻ Call the sysAppClientRcv process with the *AppAsnUp* event, passing the ConnectResponseData buffer and SSB.

Disable the ConnectResponseTimer.

❼ Call the sysAppClientRcv process with the *AsnDisconFailed* event.

Call the CircEvent process with a *CircTerminate* event.

Return all resources.

❽ Call the AsnTransSend process with the transaction data and the ASB.

❾ Call the sysAppClientRcv process with the *AsnAborted* event.

Call the CircEvent process with a *CircTerminate* event.

Return all resources.

❿ Call the AsnClientTransRcv process with the segment and the ASB.

⓫ Call the sysAppClientReceive process with the *AppTransFailed* event and SSB.

Call the sysAppClientReceive process with the *AppTransAborted* event and SSB.

Call the CircEvent process with the *CircTerminate* event.

Return all resources.

⓬ Locate the TSB for this transaction and reset the AsnTransactionResponseTimer in the TSB.

⓭ Call the sysAppClientReceive process with the *AppAsnStopped* event and return the disconnect data.

Call the CircEvent process with the *CircTerminate* event.

Return allocated resources.

⓮ Call the sysAppClientReceive process with the *AsnTransSendRetry* event, ASB, and TSB.

5.3.4 Association Server Processes

To manage state transitions, the Association layer uses the following server processes:

- AsnServerRcv
- AsnServerEvent
- AsnServerTimer

- AsnServerTransRcv
- AsnServerTransRspSend

These processes are described in Sections 5.3.4.1 through 5.3.4.5. Note that the following pseudocode examples do not provide all details and are intended only as rough guidelines.

5.3.4.1 AsnServerRcv Process

The AsnServerRcv process is called by the CircEvent process to notify the Association layer when messages arrive and to effect appropriate circuit state transitions. The process performs actions shown in the following pseudocode example.

```
void AsnServerRcv(event, message, ndb)
int event;
char *message;
NDB *ndb;
{
     /*
      * For each event received:
      */

    ASB = locateAsb(runMessage–>asbId);
    if (event == AsnRunMsgRcv)
    {
     /*
      * Parse Run message header, verifying fields in Table 6–10.
      * For each message subtype, call AsnServerEvent process
      * with appropriate event.
      */

       if (runMessage–>subtype == ConnectRequest)
          AsnServerEvent(AsnConReqMsgRcv, message, ndb–>nodeId, asb);

       if (runMessage–>subtype == DataRequest)
          AsnServerEvent(AsnDataReqMsgRcv, message, ndb–>nodeId, asb);

       if (runMessage–>subtype == Disconnect Request)
          AsnServerEvent(AsnDisonReqMsgRcv, message, ndb–>nodeId, asb);
    }

    if ((event == AsnCircuitUp) || (event == AsnCircuitDown))
    {
        asb = asbListHead;
        while (asb != NULL)
        {
            AsnServerEvent(event, message, ndb–>nodeId, asb);
            asb = asb–>next;
        }
    }
}
```

5.3.4.2 AsnServerEvent Process

The AsnServerEvent process is called by the AsnServer process and recursively to generate the association protocol.

The process uses the current state and the events shown in Table 5–7 to identify the new state. For example, if the current state is Halted and the event is *AsnConReqMsgRcv*, the new state is Started. The process extracst the new state, updates the ASB with this state, and executes actions indicated by the callout (for example, ❷). These actions are described following the table. A dash (—) indicates that no action is required.

The AsnServerEvent process performs actions shown in the following pseudocode example.

```
void AsnServerEvent(event, message, nodeId, asb)
int event;
Message *message;
char nodeId[6];
ASB *asb;
{
    int newState;
    int (*actionRoutine)();

     /*
      * Using state table, retrieve new state and action routine.
      * Call action route to start or continue protocol message exchanges.
      */
    newState = AsnServerstateTable[event, asb–>state].newState;
    actionRoutine = AsnServerStateTable[event, asb–>state].actionRoutine;
      asb–>state = newState;
    (*actionRoutine)(message, asb, ...);
}
```

5.3.4.3 AsnServerTimer Process

The AsnServerTimer process is called by the operating system once each second. The process performs actions shown in the following pseudocode example.

```
void AsnServerTimer()
{
    ASB *asb = asbListHead;

     /*
      * For every association, do the following:
      */
    while(asb != NULL)
    {
       if (asb–>state == Running)
       {
        /*
         * For every association transaction, do the following:
         */
          tsb = asb–>tsbListHead;
          while (tsb != NULL)
          {
             tsb–>AsnResyncResponseTimer--;
             if (tsb–>AsnResyncResponseTimer == 0)
```

```
                {
                   AsnServerEvent(AsnResyncTimeout, NULL, asb–>nodeId, asb);
                }
                tsb = tsb–>next
             }
          }
          asb = asb–>next
       }
    }
```

5.3.4.4 AsnServerTransRcv Process

The AsnServerTransRcv process is called by the AsnServerEvent process to reassemble a transaction request. The process performs actions shown in the following pseudocode example.

```
void AsnServerTransRcv(dataRequestSegment, asb)
DataRequest *dataRequestSegment;
ASB *asb;
{
     /*
      * Detect orphan transaction requests at the server. Verify
      * that the transaction sequence number is forward in number space.
      */

    if (CurrentSequenceNumber(dataRequestSegment, asb) == FALSE)
    {
       return; /* Drop this orphan request. */
    }
     /*
      * Find transaction for which data is intended. When first segment
      * arrives at server, notify system application. After inspecting
      * first segment, system application supplies receive buffer for
      * remaining data.
      */
    tsb = LocateTsb(asb, dataRequestSegment);
    if (dataRequestMessage–>sequenceNumber == 1)
    {
       tsb–>receiveBuffer = sysAppServerRcv(AppFirstSegment,
           dataRequestSegment,asb–>ssb); †
       CopyAllSegmentsFromTsb(tsb, tsb–>receiveBuffer);
       tsb–>segment1Arrived = TRUE;
    }

    if (tsb–>segment1Arrived == TRUE)
    {
       CopySegmentToRcvBuffer(dataRequestSegment, tsb–>receiveBuffer);
    }
    else
    {
       QueueSegment(tsb);
    }

    if (TransactionRequestComplete(tsb) == TRUE)
```

† This process returns a receive buffer for reassembly of the transaction request.

```
        {
            sysAppServerRcv(AppTransReq, tsb–>receiveBuffer,asb–>ssb);
            AsnEnableTimer(AsnResyncResponseTimer);
        }
    }
```

5.3.4.5 AsnServerTransRspSend Process

The AsnServerTransRspSend process is called by the AsnServerEvent process to fragment and send a transaction. The process performs actions shown in the following pseudocode example.

```
voidAsnServerTransRspSend (dataResponse, dataResponseLength, asb, tsb)
char *dataResponseSegment;
int dataResponseLength;
ASB *asb;
TSB *tsb;
{
    char *segment;
    char *message;

     /*
      * Based on negotiated segment size:
      */

    tsb–>maximumSegments = (dataResponseLength/asb–>segmentSize) + 1;
    DivideIntoSegments(dataResponse, dataResponseLength,
         asb–>segmentSize, tsb);

    segment = tsb–>segmentList;
    while (segment != NULL)
    {
       message = constructDataResponseMsgHeader(segment);
       / * See Table 6–14. */

       message = constructRunMsgHeader(message);
       / * See Table 6–1. */

       CircEvent(CircRunMsgSend, message, asb–>csb–>nodeId, asb–>csb);
       AsnDisableTimer(AsnResyncResponseTimer);
       segment = segment–>next;
    }
}
```

5.3.5 Association Server State Transitions

Table 5–7 shows association server events and resulting state transitions.

Table 5–7 Association Server Events and Resulting State Transitions

	Current State			
	Halted	**Started**	**Running**	**Stopped**
Command Event				
AsnCircuitDown	—	Halted ❽	Halted ❽	Halted ❾
Message Events				
AsnConReqMsgRcv	Started ❷	Started ❸	Running	—
AsnConRspMsgSend	—	Started ❸	—	—
AsnDisonReqMsgRcv	—	Stopped ❼	Stopped ❼	Stopped ⓫
AsnDisconRspMsgSend	—	Halted ❶	Halted ❶	Stopped ❿
AsnDataMsgRcv	—	Running ❹	Running ❹	—
AsnTransRspMsgSend	—	—	Running ❺	—
Timer Event				
AsnResyncTimeout	—	—	Running ❻	—

❶ Construct a Disconnect Response message (see Table 6–18) and send the message by calling the CircEvent process with a *CircTerminate* event.

Return all resources allocated for this association.

❷ Initialize an ASB for the new association.

Assign a unique association id to represent this association.

Bind the server system application association to the ASB.

Call the sysAppServerRcv process with the *AppAsnUp* event and connect data.

❸ Construct a ConnectResponse message, initializing appropriate message fields (see Table 6–12).

Construct an association Run message header for the message, initializing appropriate message fields (see Table 6–10), and call the CircEvent process with the *CircRunMsgSend* event.

Activate and initialize the ConnectResponseTimer in the ASB for this association.

Save the Connect Response data in the ASB.

❹ Call the AsnServerTransRspSend process with Data Request message and ASB.

❺ Call the AsnServerTransRspSend process with the Data Response message, ASB, and TSB.

❻ Construct a Resync Response message, filling in appropriate fields (see Table 6–16), and send the message by calling the CircEvent process with a *CircRunMsgSend* event.

❼ Call the sysAppServerRcv process with the *AppAsnStopped* event and the disconnect data.

❽ Call the sysAppServerRcv process with the *AppAsnStopped* event.

Return all resources to the system.

❾ Return all resources to the system.

❿ Construct a Disconnect Response message using the disconnect response data supplied by the server system application and filling in appropriate fields (see Table 6–18).

Send the message by caliing the CircEvent process with a *CircRunMsgSend* event.

⓫ Construct a Disconnect Response message using the disconnect response data in the ASB and filling in appropriate fields (see Table 6–18).

Send the message by caliing the CircEvent process with a *CircRunMsgSend* event.

5.3.6 Association State Transition Example

The following events occur during a normal association startup:

At the Client:

1 The client system application declares an *AsnConnect* event to the Association layer.

2 The Association layer allocates and initializes an ASB for the association. The Association layer declares a *CircCreate* event (see Chapter 4) to the Circuit layer and waits.

3 Sometime later, the Circuit layer declares an *AsnCircuitUp* event to the Association layer. The Association layer sends the Connect Request message to the server to establish an association. The Association layer starts the AsnConnectResponseTimer. If a Connect Response message is not received, the timer sends additional Connect Request messages to the server.

4 Sometime later, the Association layer receives a Connect Response message and passes the Connect Response data to the client.

5 The client system application declares an *AppTransSend* event and passes data to the Association layer. The Association layer segments and transmits the data. It starts a transaction response timer to detect a transaction failure.

6 Sometime later, the Association layer receives a transaction response. It stops the transaction response timer and declares an *AppTransComplete* event to the system application.

7 The client declares an *AsnDisconnect* event. This causes the Association layer to send a Disconnect Response message to the server and starts the AsnDisconnectResponseTimer to detect the loss of the Disconnect Request message.

8 Sometime later, the client receives a Disconnect Response and notifies the client that the association is gone. The client returns all resources to the system.

At the Server:

1 The Circuit layer at the server receives a start message to create a circuit and establishes a circuit with the client system (see Chapter 4).

2. The Association layer at the server receives a Connect Request message and declares an *AppAsnUp* event to the server system application.
3. Sometime later, the sysAppServerEvent process declares an *AsnConRspMsgSend* event, which causes the Association layer to send a Connect Response message to the client.
4. Sometime later, the server receives a transaction from the client. The Association layer declares an *AppTransReq* event, passing the data request to the sysAppServerRcv process. This process starts an AsnResyncResponseTimer to indicate when a Resync message might need to be sent.
5. Sometime later, the sysAppServerRsp process declares an *AsnTransRspMsgSend* event, causing the Association layer to send one or more Data Response messages to the client.
6. The Association layer receives a Disconnect Response message and declares an *AppAsnStopped* event, passing the disconnect data to the sysAppServerRcv process. The process declares an *AsnDisconRspMsgSend* event and sends a Disconnect Response message.

6
Message Formats

This chapter describes the following LASTport messages:

- Circuit layer messages
- Solicitation layer messages
- Association layer messages

All messages contain the Circuit message header.

In the messages formats shown, bit 0 is transmitted first on the wire.

6.1 Circuit Layer Messages

There are two types of Circuit layer messages: the Start/Stack message (described in Section 6.1.1), and the Stop message (described in Section 6.1.3).

Figure 6–1 shows the Circuit message header format; Table 6–1 describes the fields.

Figure 6–1 Circuit Message Header Format

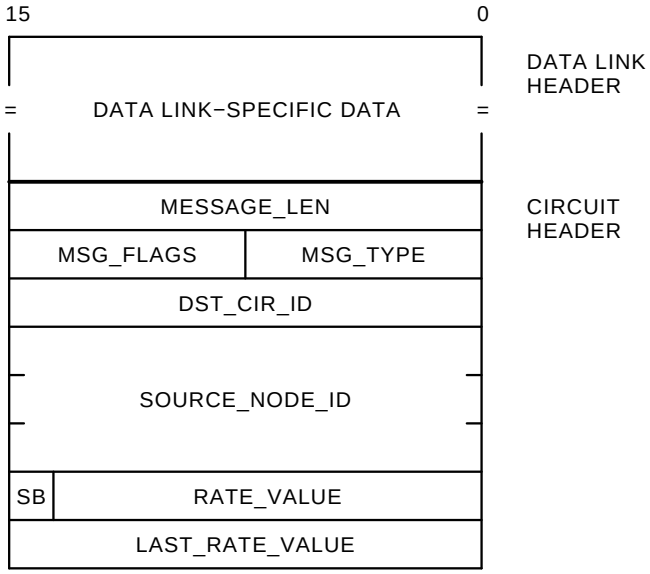


Table 6–1 Circuit Message Header Fields

Field	Length	Description
MESSAGE_LEN	2 bytes unsigned	Number of bytes in the segment (buffer) including field itself. See MSG_FLAGS for additional information.
MSG_TYPE	1 byte	The following values define message types: 0 = Run 1 = Start 2 = Stack 3 = Stop 4 = Loop 5 = Advertisement 6 = Solicit Request 7 = Solicit Response
MSG_FLAGS	1 byte	The following bits are valid for flags: • Bit 0 = 1. The data is checksummed from MESSAGE_LEN through MESSAGE_LEN bytes. The checksum follows the message after rounding up modulus 4 and is not included in the MESSAGE_LEN count. The checksum is a 4-byte unsigned value. Appendix A describes checksumming. • Bits 1–7 = TAZIOR.
DST_CIR_ID	2 bytes	A reference to the destination node CSB.
SOURCE_NODE_ID	6 bytes	A LAN-wide unique identification of the source node, which remains constant across transport incarnations. Typically, implementations use a physical hardware address for this value. Note that FDDI physical hardware addresses are specified in reverse bit order relative to Ethernet addresses.
SB	1 bit	When set, indicates that RATE_VALUE is being established.
RATE_VALUE	15 bits unsigned	A segment-per-second count.
LAST_RATE_VALUE	16 bits unsigned	A segment-per-second count.

Note that the data link driver software typically supplies the LAN header. Because this header is subject to change in newer LAN message formats, it is not included in LASTport implementations. It is assumed that the message header minimally provides the source address of the transmitter and that LAST can address it.

6.1.1 Start/Stack Message

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

Figure 6–2 shows the format of the Start/Stack message; Table 6–2 describes messsage fields. Table 6–3 describes the Start/Stack message flags.

Figure 6–2 Start/Stack Message Format

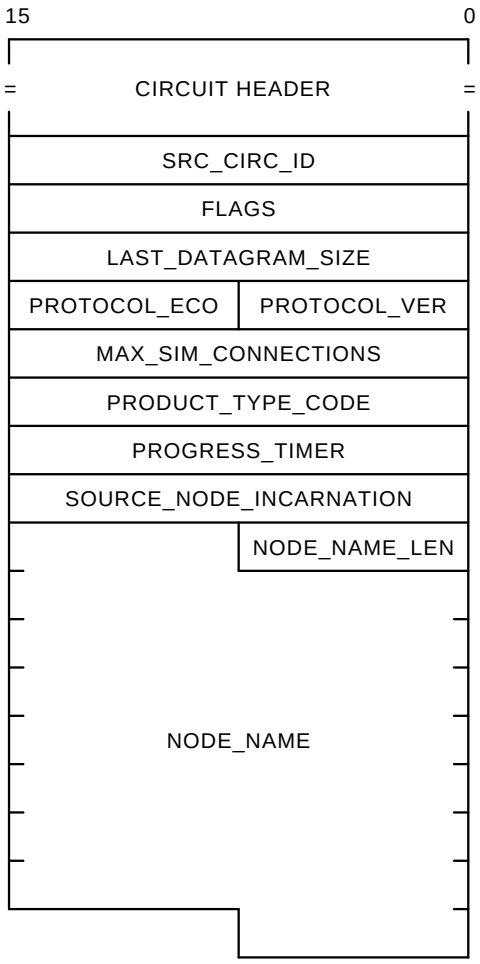


Table 6–2 Start/Stack Message Fields

Field	Length	Description
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. See Table 6–3.
LAST_DATAGRAM_SIZE	2 bytes unsigned	Each data link implementation restricts the minimum and maximum sizes of frames. An implementation must specify the maximum datagram size that it supports. The circuit client sets its datagram size in the Start message. The server does not offer larger messages for transmission to the client. The server can specify a lower value in the Stack message, instructing the client to use this value. This value determines the size of datagrams that the client offers to the Data Link layer.
PROTOCOL_VER	1 byte	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 ignores the message. Select the Start message protocol version using information supplied by the Solicit (directory service) messages.
PROTOCOL_ECO	1 byte	Protocol version ECO of this message and of all messages transmitted using this circuit. For any given protocol version, ECOs are backward compatible. The PROTOCOL_ ECO level reflects 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 circuit. The client offers a number, and the server can negotiate a lower value if it cannot support the offered value. The value returned in the Stack message is the maximum number of associations supported on this circuit. This value corresponds to the ASB vector length in the CSB.
PRODUCT_TYPE_CODE	2 bytes unsigned	See Section 6.1.2.
PROGRESS_TIMER	2 bytes unsigned	A timer, specified in seconds. This value is negotiated in the same way as the MAX_SIM_CONNECTS. The Circuit layer is required to transmit a message every PROGRESS_TIMER seconds divided by 3 if no other traffic has been transmitted on the circuit.
SOURCE_NODE_INCARNATION	2 bytes	Unique value assigned at transport initialization. This value must be different from any recent value used by a previous incarnation of this transport.

(continued on next page)

Table 6–2 (Cont.) Start/Stack Message Fields

Field	Length	Description
NODE_NAME_LEN	1 byte unsigned	Length in bytes of NODE_NAME field.
NODE_NAME	16 bytes, fixed length	Text in this field specifies the name of the client or server node.

Table 6–3 Start/Stack Message Flags

Flag	Bits	Description
INTRA_BURST_DELAY	0	Indicates how the transmitter should pipeline transmit message segments. If the flag is set to 1, message segments are transmitted one at a time, and the transmitter waits for a transmit complete event for each message segment. If the flag is set to 0, the transmitter can pipeline BURST_SIZE message segments. Notice that this flag is not negotiable and affects all circuit-based transmit operations for nodes receiving this message.
FORCE_CHECKSUMMING	1	If the FORCE_CHECKSUMMING bit is set, checksumming verification is enabled in the Start/Stack message. Either end of a circuit can require data checksumming.
TAZIOR	2–15	Transmit as zero and ignore on receipt.

6.1.2 Product Type Codes for Start/Stack Messages

The LASTport protocol supports client associations, server associations, or both for various products. Figure 6–3 shows the format of the 16-bit PRODUCT_TYPE field. Table 6–4 describes the field segments; Table 6–5 lists PRODUCT_TYPE codes for Digital COMPANY code 0, and Table 6–6 lists COMPANY codes. In the future, other companies may be assigned their own PRODUCT_TYPE codes.

Figure 6–3 Format of PRODUCT_TYPE_CODE Field

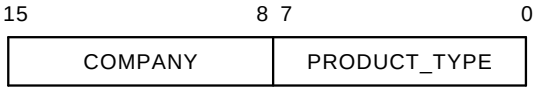


Table 6–4 PRODUCT_TYPE_CODE Field Segments

Segment	Bits	Description
PRODUCT_TYPE	0–7	Product type (256 values). A value of zero indicates that no product type is specified. See Table 6–5.
COMPANY	8–15	Company name (256 values). See Table 6–6.

Table 6–5 PRODUCT_TYPE Codes for Digital COMPANY Code 0

Product Type	Codes
Digital VAX/VMS client/server system	1
Digital VAX/VMS client system	2
Digital VAX/VMS server system	3
PATHWORKS for DOS client	4
PATHWORKS for DOS server	5
PATHWORKS for DOS client/server	6
Digital VAX/VMS small timesharing system	7
Digital VAX/VMS medium timesharing system	8
Digital VAX/VMS large timesharing system	9
Digital VAX/VMS mainframe timesharing system	10
OS/2 PC	11
UNIX small system	12
UNIX medium system	13
UNIX large system	14
UNIX mainframe system	15
Routers	16
Terminal servers	17
Network server	18
InfoServer 50	19
InfoServer 100	20
InfoServer 200	21
InfoServer 300	22
VXT 2000	23
Other	24–255

Table 6–6 COMPANY Codes

Company	Codes
Digital Equipment Corp.	0
IBM (International Business Machines) Corp.	1
Apple Computer, Inc.	2
Microsoft Corp.	3
Sun Microsystems, Inc.	4
AT & T Corp.	5
Hewlett-Packard Co.	6
The Santa Cruz Operation, Inc.	7
Cisco, Inc.	8
Alphaphatronix, Inc.	9

(continued on next page)

Table 6–6 (Cont.) COMPANY Codes

Company	Codes
Prime Computer Corp.	10
Data General Corp.	11
Virtual Microsystems, Inc.	12
Ricoh Corp.	13
Sony Corp.	14
Novell Corp.	15
Silicon Graphics, Inc.	16
Reserved	17–255

6.1.3 Stop Message

Figure 6–4 shows the Stop message format. Table 6–7 describes the message field.

Figure 6–4 Stop Message Format

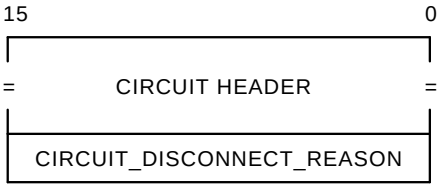


Table 6–7 Stop Message Field

Field	Length	Description
CIRCUIT_DISCONNECT_REASON	2 bytes unsigned	A zero value means that no reason is given. The currently defined reasons are as follows: 1 = Invalid response timer 2 = Circuit stop from user 3 = Protocol error 4 = No resources

6.2 Solicitation Layer Messages

Figure 6–5 shows the common format of Solicitation layer Advertisement, Solicit Request, and Solicit Response messages. Table 6–8 describes message fields.

Figure 6–5 Format of Advertisement and Solicit Messages

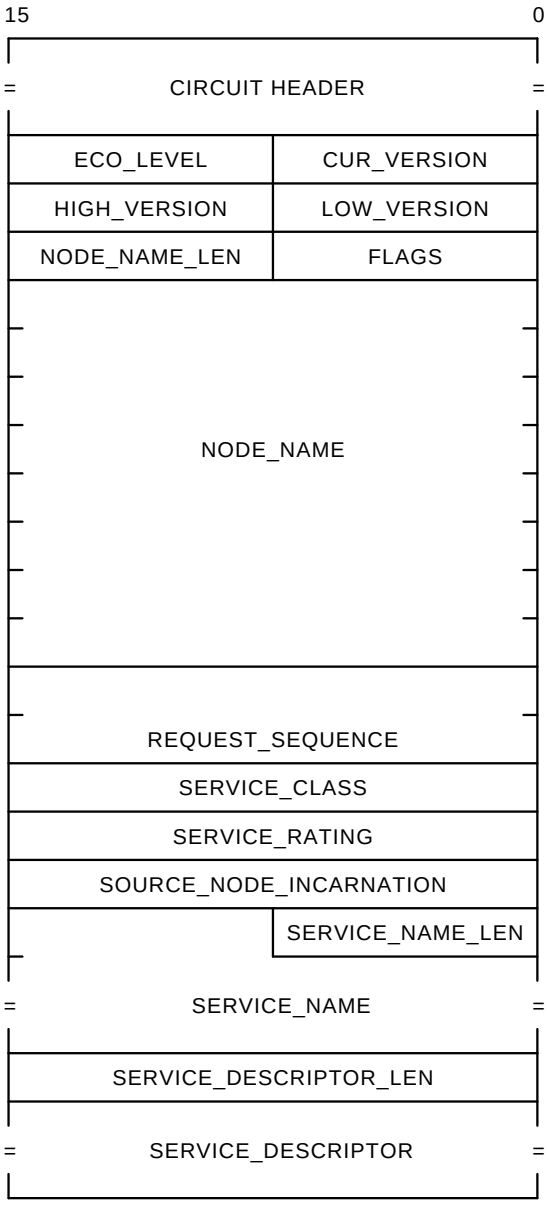


Table 6–8 Advertisement and Solicit Message Fields

Field	Length	Description
CUR_VERSION	1 byte unsigned	Protocol version of this message. This node supports message exchanges using this protocol version.
ECO_LEVEL	1 byte unsigned	ECO of this message and this protocol version.
LOW_VERSION	1 byte unsigned	Lowest version supported by this node.
HIGH_VERSION	1 byte unsigned	Highest version supported by this node. When attempting to create a circuit (construct a Start message), choose a protocol version in the range defined by LOW_VERSION and HIGH_VERSION.
FLAGS	1 byte unsigned	Bitfield describing the transmitter transport type and flags. This field is required in all Advertisement and Solicit messages. Transports use the field to determine whether the transmitting node is maintained in the topology. For example, because a client-only node has the CLIENT_FLAG bit set, the node ignores messages from other client-only nodes. Both the CLIENT_FLAG and SERVER_FLAG bits can be set. Table 6–9 lists flags and settings for this field.
NODE_NAME_LEN	1 byte unsigned	Length in bytes of the transport node name in the range 1 to 16. This field is required for all Advertisement messages.
NODE_NAME	16 bytes	Name associated with the transmitting transport. This field is required for all Advertisement and Solicit messages.
REQUEST_SEQUENCE	4 bytes	Reserved for use by the transmitter. This field must be copied to any Solicit Response message. During path maintenance operations, the LASTport protocol uses this field as the Solicit Identifier.

(continued on next page)

Table 6–8 (Cont.) Advertisement and Solicit Message Fields

Field	Length	Description
SERVICE_CLASS	2 bytes unsigned	A unique identifier indicating the service type. Clients and servers are registered with the transport and are assigned Service Class values. If SERVICE_CLASS is zero, a Solicit Response message is required. Currently defined service classes are the following: 1 Virtual Disk Service 2 Primitive Distributed Queueing Service 3 LANsess File Service 4 Virtual Tape Service
SERVICE_RATING	2 bytes	Field used by the server system application to indicate quality of service.
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	Length in bytes of Service Name. This field is valid for all directory service messages. If SERVICE_NAME_LEN is zero, the message is interpreted as a topology message. If the SERVICE_NAME_LEN is not zero, it is interpreted as service related.
SERVICE_NAME †	SERVICE_NAME_LEN bytes	Service Name of interest. SERVICE_CLASS users are free to construct the Service Name to suit their applications.
SERVICE_DESCRIPTOR_LEN	2 bytes unsigned	Length in bytes of the following SERVICE_DESCRIPTOR field.
SERVICE_DESCRIPTOR †	SERVICE_DESCRIPTOR_LEN bytes	Data associated with the SERVICE_NAME field.

†Because the length of the SERVICE_NAME and SERVICE_DESCRIPTOR fields is limited by the LAST_DATAGRAM_SIZE, the Solicit message length cannot exceed the LAST_DATAGRAM_SIZE.

Maximum Solicit message length = CIRCUIT_HEADER (16 bytes) + SERVICE_NAME_LEN + SERVICE_DESCRIPTOR_LEN + remaining message fields (35 bytes).

When transmitting Solicit Request and Solicit Response messages, system applications must take into account the LAN topology to achieve complete connectivity. Bridges between different LAN interconnects (for example, Ethernet, FDDI, LocalTalk™) might not be capable of forwarding Solicit messages. System applications might want to limit the size of these messages.

Table 6–9 FLAGS Field Settings

Flag	Setting	Description
CLIENT_FLAG	Bit 0 = 1	CLIENT_FLAG set if transport type is client.
SERVER_FLAG	Bit 1 = 1	SERVER_FLAG set if transport type is server.
PURGE_ALL_PATHS_FLAG	Bit 2 = 1	All known path data is purged for the SOURCE_NODE_ID node except the path on which the message arrived.
TAZIOR	Bits 3–7	Transmit as zero and ignore on receipt.

6.3 Association Layer Messages

Association layer messages, also called Run messages, are used to identify transactions. The following message subtypes are defined:

- Connect Request
- Connect Response
- Data Request
- Data Response
- Resync Response
- Disconnect Request
- Disconnect Response

These are described in Sections 6.3.2 through 6.3.8. Section 6.3.1 describes the Run message header.

6.3.1 Run Message Header

Run messages are transmitted after completion of a Start/Stack message exchange and continue until a Stop message deletes the CSB. The MSG_TYPE field in Run messages is set to 0 by default.

Run messages include the common header (CIRCUIT HEADER and RUN MESSAGE HEADER) shown in Figure 6–6. Table 6–10 describes the message fields. Table 6–11 describes the mode indicators for Run messages.

Figure 6–6 Run Message Header Format

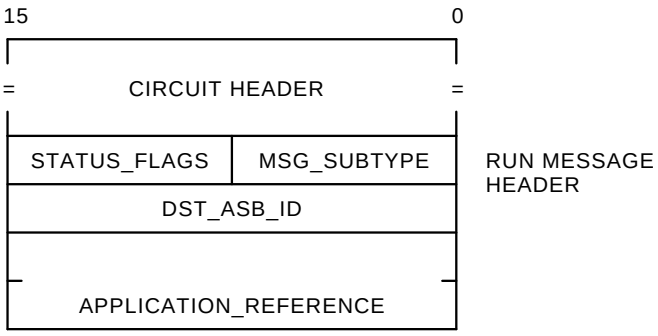


Table 6–10 Run Message Fields

Field	Length	Description
MSG_SUBTYPE	1 byte unsigned	The value for this message type. The following message subtypes are defined: 0 = Data Request 1 = Data Response 2 = Connect Request 3 = Connect Response 4 = Reserved 5 = Resync Response 6 = Disconnect Request 7 = Disconnect Response
STATUS_FLAGS	8 bits	Association modifiers are as follows: Bits 0–1 (2 bit unsigned integer) is the mode indicator. Mode indicators are assigned as described in Table 6–11. Bits 3–7 = TAZIOR.
DST_ASB_ID	2 bytes unsigned	An index to the remote ASB. This value must be zero for a connect request message, and must not be zero for any other Run message.
APPLICATION_REFERENCE	4 bytes	This field, also called the Client Handle, is supplied by the client system application when a request is made. The server returns this field in response Run headers, allowing the client to match a particular response with the application request. If the response does not match the request, the client ignores the response.

Table 6–11 Mode Indicator for Run Messages

Mode	Mode Name	Description
Mode 0	Idempotent_mode	The server can discard transaction responses immediately, without waiting for acknowledgment.

6.3.2 Connect Request Message

Figure 6–7 shows the format of the Connect Request message. Table 6–12 describes the message fields.

Figure 6–7 Connect Request Message Format

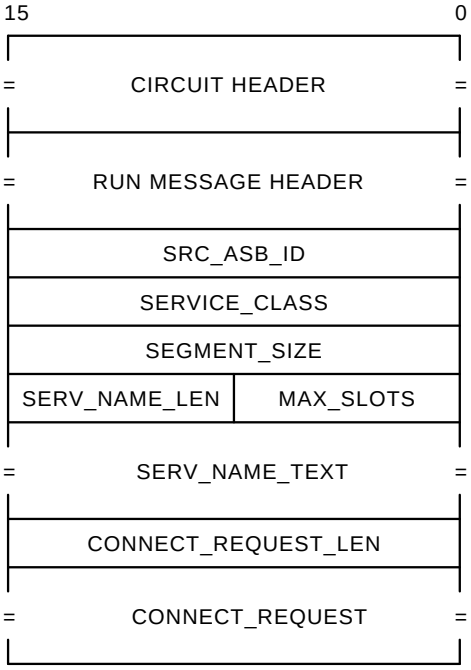


Table 6–12 Connect Request Message Fields

Field	Length	Description
SRC_ASB_ID	2 bytes unsigned	The handle of the client ASB.
SERVICE_CLASS	2 bytes unsigned	A value representing the type of service being connected.
SEGMENT_SIZE	2 bytes unsigned	Segment size used to fragment transaction requests. Represents the amount of user data transferred in a Data Request/Data Response message. See Section 6.3.2.1.
MAX_SLOTS	1 byte unsigned	Maximum number of transactions that a client can pipeline. See Section 6.3.2.2.
SERV_NAME_LEN	1 byte unsigned	Number of characters in the SERVICE_NAME_TEXT field.
SERV_NAME_TEXT	SERV_NAME_LEN bytes	Destination Service Name used in forming the connection.
CONNECT_REQUEST_LEN	2 bytes unsigned	Length in bytes of CONNECT_ REQUEST field.
CONNECT_REQUEST	CONNECT_REQUEST_LEN bytes	CONNECT_REQUEST_LEN bytes of application-specific data. The total length of the Connect Request message is limited by the LAST_ DATAGRAM_SIZE. The maximum message length is computed as for Solicit messages. (See Table 6–8.)

6.3.2.1 Segment Size Computation

A segment size cannot be larger but can be smaller than the size computed using the following formula:

SS = DS - RH - 7

Elements are as follows:
SS = Segment size
DS = Datagram size (from Start message)
RH = Run request header size (8 bytes)
7 = Allowance for data checksumming and rounding

The client specifies its segment size in the Connect Request message. The server can return a smaller value in the Connect Response message. Transactions are segmented using the negotiated segment size.

6.3.2.2 Maximum Slots Computation

The MAX_SLOTS value is the maximum number of transactions that this client concurrently supports. The client specifies its MAX_SLOTS value in the Connect Request message. The server can return a smaller value in the Connect Response message. The value associated with the Connect Response message is the maximum number of concurrent transactions available for this association.

6.3.3 Connect Response Message

Figure 6–8 shows the format of the Connect Response message. Table 6–13 describes the message fields.

Figure 6–8 Connect Response Message Format

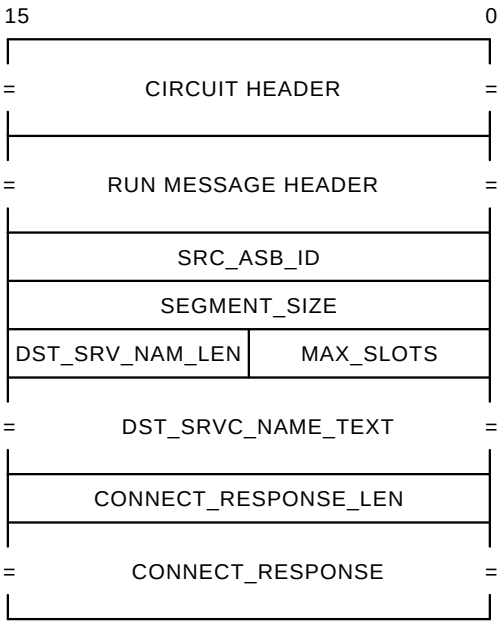


Table 6–13 Connect Response Message Fields

Field	Length	Description
SRC_ASB_ID	2 bytes unsigned	The handle of the server ASB.
SEGMENT_SIZE	2 bytes unsigned	Segment size used to fragment transaction requests. Represents the amount of user data transferred in a Data Request/Data Response message. See Section 6.3.2.1.
MAX_SLOTS	1 byte unsigned	Maximum number of transactions that the server can pipeline (support simultaneously). See Section 6.3.2.2.
DST_SERVICE_NAME_LEN	1 byte unsigned	Number of characters in the DST_SERVICE_NAME_TEXT field.
DST_SERVICE_NAME_TEXT	SERVICE_NAME_LEN bytes	Destination service name used in forming the connection.
CONNECT_RESPONSE_LEN	2 bytes unsigned	Length in bytes of CONNECT_ RESPONSE field.

(continued on next page)

Table 6–13 (Cont.) Connect Response Message Fields

Field	Length	Description
CONNECT_RESPONSE	CONNECT_RESPONSE_LEN bytes	Application data. The total length of the Connect Response message is limited by the LAST_DATAGRAM_SIZE. The maximum message length is computed as for Solicit messages. (See Table 6–8.)

6.3.4 Data Request Message

Figure 6–9 shows the format of the Data Request message. Table 6–14 describes the message fields.

Figure 6–9 Data Request Message Format

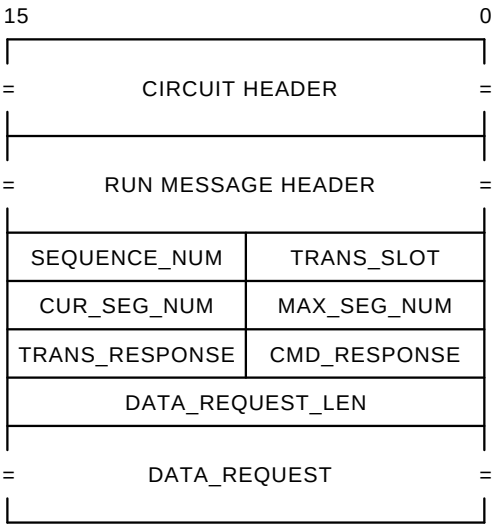


Table 6–14 Data Request Message Fields

Field	Length	Description
TRANS_SLOT	1 byte	Multiple transactions can be pipelined through an association. This value provides an index to the current transaction slot for this message. Cannot be zero.
SEQUENCE_NUM	1 byte †	Unique sequence number associated with this message for this transaction ID. This field is incremented for each new transaction within a slot. Can be zero.
MAX_SEG_NUM	1 byte unsigned	Used to identify the number of segments in this transaction request. Cannot be zero.
CUR_SEG_NUM	1 byte unsigned	Used to identify this segment number. Cannot be zero.
CMD_RESPONSE	1 byte unsigned	The value in seconds specified by the client, which is an initial time limit for receipt of a request message acknowledgment or transaction response message. This value is required to determine whether the request message is lost. This value is referenced as the AsnTransactionResponseShortTimer. (See Section 2.1.4.)

†In the next version of the architecture, length will be 4 bytes.

(continued on next page)

Table 6–14 (Cont.) Data Request Message Fields

Field	Length	Description
TRANS_RESPONSE	1 byte unsigned	The value in seconds specified by the client, which is subsequently used as a time limit for completion of the transaction and can be reset by the server using Resync Response messages. This value is referenced as the AsnTransactionResponseLongTimer. (See Section 2.1.4.)
DATA_REQUEST_LEN	2 bytes unsigned	Length in bytes of the next field.
DATA_REQUEST	DATA_REQUEST_LEN bytes	User-supplied data.

6.3.5 Data Response Message

Figure 6–10 shows the format of the Data Response message. Table 6–15 describes the message fields.

Figure 6–10 Data Response Message Format

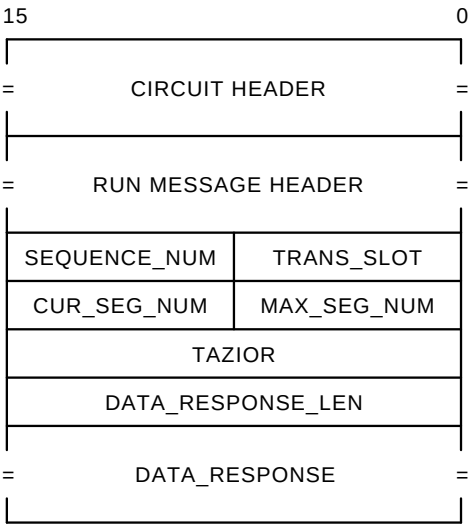


Table 6–15 Data Response Message Fields

Field	Length	Description
TRANS_SLOT	1 byte	This value provides an index to the current transaction ID for this message. Cannot be zero.
SEQUENCE_NUM	1 byte †	Unique sequence number associated with this message for this transaction ID. This field is incremented for each unique transaction. Value is copied from the Data Request message. Can be zero.
MAX_SEG_NUM	1 byte unsigned	Used to identify the number of segments in this transaction response. Cannot be zero.
CUR_SEG_NUM	1 byte unsigned	Used to identify this segment number. Cannot be zero.
TAZIOR	2 bytes	Transmit as zero and ignore on receipt.
DATA_RESPONSE_LEN	2 bytes unsigned	Length in bytes of the next field.
DATA_RESPONSE	DATA_RESPONSE_LEN bytes	User-supplied data.

†In the next version of the architecture, length will be 4 bytes.

6.3.6 Resync Response Message

Figure 6–11 shows the format of the Resync Response message. Table 6–16 describes the message fields.

Figure 6–11 Resync Response Message Format

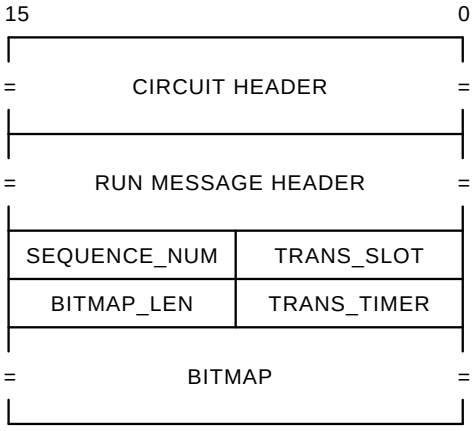


Table 6–16 Resync Response Message Fields

Field	Length	Description
TRANS_SLOT	1 byte	Multiple associations can be pipelined through a association. This value provides an index to the current transaction ID for this message. Cannot be zero.
SEQUENCE_NUM	1 byte †	Unique sequence number associated with this message for this transaction ID. This field is incremented for each unique transaction. Can be zero.
Call Phil Wells re next two fields		
TRANS_TIMER	1 byte unsigned	An unsigned byte. The units are 10 millisecond intervals.
BITMAP_LEN	1 byte signed	A byte indicating the length of the following field.
BITMAP	variable length	A bitmap of transaction segments.

†In the next version of the architecture, length will be 4 bytes.

6.3.7 Disconnect Request Message

Figure 6–12 shows the format of the Disconnect Request message. Table 6–17 describes the message fields.

Figure 6–12 Disconnect Request Message Format

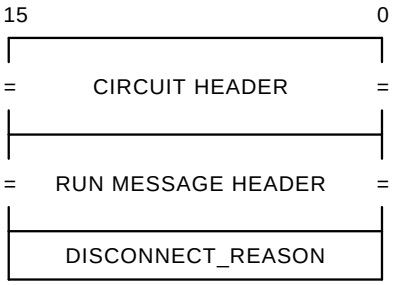


Table 6–17 Disconnect Request Message Fields

Field	Length	Description
DISCONNECT_REASON	2 bytes unsigned	Disconnect reason. A reason of 0 is undefined. The defined reasons are as follows: 1 = Invalid message format received 2 = Invalid local connection identification received 3 = Invalid remote connection identification received 4 = Association_halt event received from client 5 = No progress being made 6 = Time limit expired 7 = Retransmit limit reached 8 = No resources 9 = Service Name has changed

6.3.8 Disconnect Response Message

Figure 6–13 shows the format of the Disconnect Request message. Table 6–18 describes the message fields.

Figure 6–13 Disconnect Response Message Format

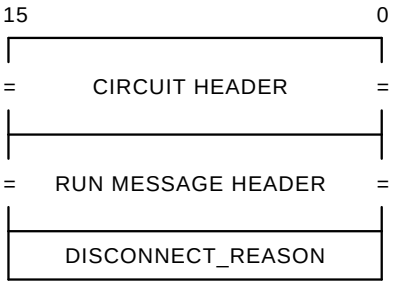


Table 6–18 Disconnect Response Message Fields

Field	Length	Description
DISCONNECT_REASON	2 bytes unsigned	Disconnect reason. A reason of 0 is undefined. The defined reasons for disconnects are as follows: 1 = Invalid message format received 2 = Invalid local connection identification received 3 = Invalid remote connection identification received 4 = Association_halt event received from client 5 = No progress being made 6 = Time limit expired 7 = Retransmit limit reached 8 = No resources 9 = Service name has changed 10 = Server rejects connect

A

Checksumming Algorithm

This appendix describes the checksumming algorithm that the LASTport protocol uses to detect errors. The checksumming flag contained in the Circuit message header controls checksumming for all LASTport messages.

The option to require checksumming under circuit control is negotiated at circuit startup in the Start and Stack messages. If either end of the circuit requires checksumming, both must generate checksummed messages and verify the received message checksum. This mechanism protects user data when communication equipment is known to be faulty.

The algorithm shown in Example A–1 shows a correct VMS checksum implementation that checks for groups of eight longwords and is designed to take advantage of certain VAX processor pre-fetch instructions. The $DISPATCH macro is commonly used to generate a CASE*x* instruction. Note that because checksum calculation for large packets can be a lengthy process, implementations must take hardware capabilities into account.

Example A–1 VAX Checksum Code

```
        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_LEN/4)/8
```

(continued on next page)

Example A–1 (Cont.) VAX Checksum Code

```
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 circuit message header MESSAGE_LEN field, which is added to each successive 32-bit value for (MESSAGE_LEN+3)/4 32-bit integer values. The actual checksum itself is a 32-bit unsigned integer value located at offset MESSAGE_LEN + ((MESSAGE_LEN+3)/4).

Glossary

advertisement

A data multicast primitive message that allows LASTport server nodes to advertise node-specific **services** to their potential clients. The LASTport **Circuit layer** users Advertisement, Solicit Request, and Solicit Response messages to maintain connectivity.

application

See **system application**.

association

The active relationship between a client instance and a server instance for a particular **service** provided by a server instance. An association is required to perform transactions between a **client** and a **server**.

Association layer

A component of the LASTport protocol than manages connections and transactions between a **client** and a **server**.

checksum

A sum of digits or bits used to verify whether a number or an operation is valid.

circuit

The relationship between two nodes. Only a single circuit exists between any two nodes, regardless of the number of **associations**.

Circuit layer

A component of the LASTport protocol that defines and maintains the **circuit topology** and distributes messages between source and destination nodes.

circuit topology

The set of paths (logical links) that connect client and server nodes in a **local area network**. The circuit topology is mapped during the solicitation process. The topology can change depending on which paths and network adapters are available.

client

The software requesting a **service** of a **server**.

client request semantics

A set of rules that ensures execution of a **transaction** by a **server** at least once or causes an **association** to fail. The **client** specifies the retry interval and the retry count for the transaction.

client response semantics

A set of rules that ensures execution of a **transaction** by a **client** at least once. No execution at the **server** occurs after the **response** is delivered successfully.

client–server protocol

A protocol in which one **node** has priority over the other. For example, in the LASTport protocol, the **client** has priority over the **server**. The client initiates all requests for a **service**, and the **server** must respond to the client's request.

communication interface

A mechanism by which system components located on the various nodes in a network can interact through the exchange of messages conveying data and control information. Communication interfaces are usually implemented using communication **protocols**.

commutative transaction

A **transaction** that can be completed in any order without introducing errors in the result. The LASTport protocol requires that all concurrently executing transactions be commutative. This requirement allows the protocol to process multiple transactions concurrently, because transactions can complete across the session interface in the order in which responses arrive. No state need be maintained to preserve the order of transactions.

component

A constituent element of a **system**.

congestion control

The mechanisms that prevent catastrophic collapse of the distributed system (also known as "livelock") when instantaneous demand for service exceeds the available network capacity.

connection

The function that creates a new, agent–specific **association** between a **client** and a **server**. A connection is initiated by a client.

dally

A system application mechanism used to prevent flooding of the client's network adapter when multiple servers respond to a Solicit Request message.

datagram

A **packet** assembled into a network frame for transmission to remote nodes.

data link

An extended **local area network (LAN)** segment.

Data Link layer

A network protocol component that transmits datagrams between a **client** and a **server**. The Data Link layer assembles packets into network frames for transmission to remote nodes as datagrams.

data link port

The device that connects a **node** to a **data link**, also called an adapter. A data link port is associated with a single node. A node can have multiple data link ports.

data slot

A division of a LAT **packet** that contains the data from a LAT session.

data transfer service

A **service** that enables a **client** to send requests to a **server** and that enables servers to respond.

directory service

A **service** that locates a **server** for the **system application**. Directory services occur in the LASTport **Solicitation layer**.

disconnection

The act of terminating an **association**.

end-to-end guarantee

A guarantee ensuring that all transaction segments transmitted by a **client** arrive at a **server**, that the complete **transaction** is processed by the server, and that the results are returned to the client.

forward sequence number

A number defined for each **transaction slot** at the server as $n + 1$ to $n + 127$, where n is the sequence number of the LASTport transaction processed on that slot. This range covers half of the 256 possible sequence numbers. † Sequence numbers outside that range are assumed to be from previous transactions, and they are rejected at the server. To allow for the possibility of lost, and therefore, non-sequentially numbered requests, the server processes any requests in the set of forward sequence numbers.

frame

The primitive unit of data transfer between nodes on the **local area network**.

header

The control information prefixed in a message text, such as source or destination code or message type.

idempotent transaction

A **transaction** that can be repeated without changing the system state. The repeated transaction request has the "same strength" as the original request, assuming that the request is repeated in isolation from other requests.

idempotent procedure

A procedure that can use the LASTport protocol directly, such as Digital Equipment Corporation's Mass Storage Control Protocol (MSCP), access to read-only data in general, and name translation.

---

† In the next version of the architecture, the range will be $n + 1$ to $n + (2^{31} + 1)$.

interface

A point of interaction between two components or between a **system** and its environment. The LASTport protocol contains interfaces of the following types:

- User interfaces
- Communications interfaces
- Programming interfaces
- Data interfaces
- System interfaces

LAN

See **local area network**.

LAT protocol

A communications protocol used in a **local area network**. LAT is Digital's local area transport product.

layer

A set of software that is ordered partially with respect to dependency; that is higher layers can depend on lower layers, but lower layers cannot depend on higher layers.

The LASTport protocol includes three functional layers: the Circuit, Association, and Solicitation layers.

local area network (LAN)

Loosely, the set of nodes capable of communicating with any other node in the set across some data link segment. All nodes in the LAN must have at least one data link segment in common.

message

A **datagram** that is formatted with one of the architecturally defined LASTport protocol headers and identified in the local area network header as a LASTport protocol packet.

multicast

The ability to "address" a single **message** for receipt by any **data link port** that enables the multicast address. This capability supports the LASTport advertisement and solicit primitives.

Name Space

A collection of logically related names, such as LASTport/Disk **Service Names**, that identify objects accessible on a network.

Name Space Handler

A system application data structure that identifies a **Name Space**.

Network layer

A layer that provides user control of and access to operational parameters and counters in lower layers.

network topology

The physical arrangement and relationship of interconnected nodes in the network.

Name String

An architecturally specified name conveyed in a **Parameter Code**.

node

A computer system on the **local area network**. The LASTport protocol requires only that a node be able to identify itself uniquely in space and time. A 48-bit identifier, such as the hardware address of a particular local area network controller, identifies the node in space, and a 16-bit value identifies a specific instance of the operating system.

orphan transaction

A complete transaction request or response that is no longer current but is still active in a **circuit**.

packet

The unit of data created by the LASTport **Circuit layer** and transmitted to the **Data Link layer**. A packet is inherently unreliable; that is, the local area network can neither guarantee reliable delivery of packets nor indicate delivery of any given packet.

performance

A measurement of the resources required to perform an operation. The most common measurement is the elapsed time needed to perform an operation. The elapsed time from the end of user input until output data is displayed is called response time. Consumption of other resources such as processor, communication, I/O, and memory resources might also be measured as part of performance.

Parameter Code

A system application code that conveys architecturally specified names called **Name Strings**.

Parameter List

A LASTport/Disk mechanism that allows extensions to message formats and **Name Spaces**.

path maintenance

The process of determining which paths remain viable between a **client** and a **server** after the solicitation process. A **circuit** is maintained as long as messages arrive on a path during a timed interval.

The **client** and **server** follow the same protocol for maintaining the **network topology**.

peer-to-peer transport protocol

A transport protocol such as DECnet NSP, in which data exchanges between nodes operate symmetrically. Usually, only one **transaction** can be conducted at a time, and one transaction must complete before another can start. The protocol used at both nodes is identical. By contrast, in a client–server protocol, one node has priority over the other.

platform

A combination of computer hardware and software that provides an environment for higher-level software.

protocol

A complete specification of a sequence of operations and messages needed to accomplish some specified processing or information exchange.

Examples: DECnet NSP protocol, LASTport protocol, LAT protocol, OSI stack, TCP/IP stack, X.400 protocol.

registration service

A **service** that allows the **system application** to interact with the LASTport protocol. Registration services make the Directory, Association, and Data Transfer services available to the system application. Registration services are not described in the LASTport architecture because they are implemented differently for each **system**.

reliability

The extent to which a **system** or part of a system yields the expected results on repeated trials. "Expected results" means correct output, without unintended side effects, meeting the performance specification. Reliability is measured by mean time between failures (MTBF).

request

The data that a **client** sends to a **server**.

request–response semantics

A set of rules that defines the relationship between a **client** and a **server**. The client sends a **request** to the server, and the server sends a **response** to the client.

response

The data returned by a **server** reacting to the request data supplied by a **client**.

segment

The unit of communication created by the LASTport **Association layer** and passed to the **Circuit layer** for transmission. Each segment has a **transaction identifier**.

segment number

A number that identifies the position of a **segment** in a **transaction**. The segment number is used to reassemble a transaction once the transaction packets reach the destination node.

sequence number

See **transaction sequence number**.

server

Software that provides a **service** to a **client**.

server name

A **Name String** that uniquely identifies a **server** on the **local area network**.

server request semantics

A set of rules that ensures execution of a **transaction** by a **server** exactly once.

server response semantics

A set of rules that requires a **server** to return the results of a **transaction** to a **client** exactly once.

service

A set of invocable routines that provides a complete, well-defined function. A service comprises a number of operations and can also maintain state.

The software that uses a service is called a **client**; the software providing a service is called a **server**. A given piece of software can be both a client and a server (for different services) at the same time. In a distributed system, client and server may be on different nodes of a network.

Examples: file services, naming services, virtual disk services.

Service Instance

A description of an instance of the requested **Service Name**.

Service Name

The name of a system application virtual disk. Multiple virtual disks or a single disk can be mapped to a single physical medium. Conversely, a single virtual disk can span multiple physical media.

Session layer

A layer that defines the system-dependent aspect of logical link communication. A logical link is a **circuit** on which information flows in two directions. Session control functions include name-to-address translation, process addressing, and, in some systems, process activation and access control.

slot

See **transaction slot**.

slot number

See **transaction slot number**.

solicitation

A function that allows a **client** to transmit data to all **servers** and receive response data from specific servers. A client builds a dynamic service binding capability, in which the decision to connect to a particular server can be deferred until a specific **service** is needed.

Solicitation layer

A component of the LASTport protocol that provides a combined service for directory operations, association management, and task naming.

subsystem

See **system**.

system

A set of **components** organized in a particular fashion and working together to achieve a certain outcome or objective. A system exists within an environment and interacts with objects in that environment.

Systems themselves can be composed to form even larger systems, sometimes referred to as macrosystems; the constituent systems are then known as **subsystems**.

system application

An application program that facilitates the development, management, or use of an information system.

Examples: language compilers, data structure viewers, backup and restore tools, file editors, system builders, command language interpreters, compound document editors, mail viewers.

transaction

The fundamental client–server communication paradigm supported by the LASTport protocol. A transaction comprises a two-step sequence that takes place within the context of a preexisting association. First, the client establishes the association and supplies request data to the server. The server then returns response data to the client in the context of the transaction that supplied the request data.

transaction identifier

An identifier that includes a **transaction slot number** and that is found in the **header** of every **segment** passed from the **Association layer** to the **Circuit layer**.

transaction sequence number

An 8-bit field (value 1 to 255) that, along with the transaction slot number, uniquely identifies a transaction. In the LASTport client Association layer, each transaction slot has a counter to record the current sequence number. On an initial transaction attempt, the client Association layer assigns the next available **transaction slot number** and transaction sequence number.

transaction slot

An active exchange in the context of its **association**. The number of slots supported by an association defines how many exchanges within that association can be active concurrently.

transaction slot number

A one-byte number that identifies a **slot**. The slot number identifies a **segment** as belonging to a particular **transaction**. All segments for one transaction carry the same slot number for both the **request** and the **response**.

Transport layer

A layer used to route a **datagram** to its destination. A Transport layer also provides **congestion control** and packet lifetime control.

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

W