The DECnet/OSI for OpenVMS Version 5.5 Implementation

1 Abstract

The DECnet/OSI for OpenVMS version 5.5 product implements a functional Digital Network Architecture Phase V networking product on the OpenVMS system. This new software product ensures that all existing OpenVMS application programs utilizing published interfaces to DECnet-VAX Phase IV operate without modification over the new DECnet product. The components of DECnet/OSI for OpenVMS version 5.5 include the new interprocess communication interface. The design goals and implementation strategy were redefined for network management, the session control layer, and the transport layer. The configuration utility was structured into several files that are easy to read.

The DECnet Phase V networking software presented the DECnet-VAX development team with a major challenge. Although the Digital Network Architecture (DNA) has always corresponded to the lower layers of open systems interconnection (OSI), the Phase V architecture has substantial differences from Phase IV in many layers. For example, the session control layer now contains a global name service.[1]

DECnet Phase V also added new network management requirements for all layers. In most cases, the existing Phase IV code could not be adapted to the new architecture; it had to be redesigned and rewritten. This presented the engineers with the opportunity to restructure and improve the older pieces of code that have been continually modified and enhanced since the first release of DECnet-VAX. Due to the large installed customer base, however, it also presented a huge compatibility problem. We could not simply drop the old in favor of the new; we needed to ensure that the customers' DECnet-VAX applications would continue to be supported.

This paper gives an overview of the design of the base components in the new DECnet/OSI for OpenVMS version 5.5 product. It then presents details about the internals of the network management, session control, and transport layers. Finally, the new configuration tool designed for DECnet/OSI for OpenVMS version 5.5 is discussed. Unless otherwise noted in this paper, the term DECnet/OSI for OpenVMS refers to version 5.5 of the product.

2 High-level Design

Numerous goals were identified during the design phase of the base components for the DECnet/OSI for OpenVMS software. Foremost among these goals was to conform to the DNA Phase V architecture and to support image-level compatibility for existing Phase IV applications. Care was also taken

in the design to allow the product to be extensible to accommodate the ongoing work with industry standards.

The DECnet/OSI for OpenVMS Version 5.5 Implementation

Design Overview

The queue I/O request ($QIO) application programming interfaces (APIs) for the VAX OSI transport service and DECnet-VAX are already defined and widely used by network applications. To ensure that existing applications would continue to work, these interfaces were modified in a compatible fashion. As a result, not all of the capabilities of Phase V could be added to the existing APIs. A new API, the interprocess communication interface ($IPC), was developed to support all the functions defined in the Phase V session control layer. In addition, the $IPC interface was designed to allow for future capabilities.

The $QIO and $IPC interfaces interpret the application's requests and communicate them to the DNA session control layer through a kernel mode system interface called session services. In the initial release of DECnet /OSI for OpenVMS, the VAX OSI transport service joined its $QIO interface to the stack at the network layer. The first follow-on release will fully support this API. It will be rewritten to interface directly to the common OSI transport module.

DECnet/OSI for OpenVMS implements each layer of the Phase V architecture in separate modules. These modules require a well-defined interface to communicate. This is supplied by the new interrupt-driven VAX communication interface. This interface defines the rules used by cooperating VAX communication modules to exchange information. The upper VAX communication modules consume a set of services, and the lower modules provide services. The lower VAX communication modules define the explicit messages and commands that are passed between the modules. This definition is then referred to as the lower layer's VAX communication interface. For example, the transport layer provides a service to the session control layer. Transport is the lower module, and session is the upper. The rules for how the interface works are defined by the VAX communication interface itself, but the commands and services supplied by the transport layer are defined by that layer. As a result, the interface between the session and transport layers is referred to as the transport VAX communication interface.

To comply with the new Enterprise Management Architecture (EMA), each of the modules supplies one or more manageable entities to network management. This is accomplished by the EMA agent (EMAA) management facility. EMAA supplies both an entity interface to the individual modules and an EMAA interface to the network. This interface is discussed further in the Network Management section.

Implementation of the Modules

Each DECnet/OSI for OpenVMS base component is implemented in one of three ways. The most prominent method is through OpenVMS executive loadable

images. These loadable images are all placed in the SYS$LOADABLE_IMAGES system directory during installation and loaded as part of the NET$STARTUP procedure, which the OpenVMS system runs during a system boot.

The two $QIO interfaces must operate within the OpenVMS I/O subsystem. As a result, they are both coded as device drivers and loaded during NET$STARTUP by the SYSGEN utility. Once started, they can create a VAX communication interface port to the appropriate modules to process their network requests.

Figure 1 shows the components of the DECnet/OSI for OpenVMS product and their logical relationship to each other.

The third way a component can be implemented is as a standard OpenVMS image or shareable image. These images include NET$ACP.EXE, which is started as a system process by NET$STARTUP, and NCL.EXE, which is the utility that supplies the network control language (NCL) interface to users. Other images, such as NET$MIRROR.EXE, are started by the network software in a separate process when a network request is received for the application.

Implementation of the Base Image

The base image, SYS$NETWORK_SERVICES.EXE, has been present on all OpenVMS systems since version 5.4. The OpenVMS system loads this executive image early in the boot cycle. The default file shipped with OpenVMS is a stub that simply sets a system cell during initialization to indicate that the older Phase IV code is loaded. This system cell can then be interrogated through an OpenVMS system service or from a Digital Command Language (DCL) command line to determine which version of the DECnet software is loaded.

When the DECnet/OSI for OpenVMS product is installed, the base image is replaced with the Phase V version. The new image sets the system cell to indicate that Phase V is loaded. It provides a host of common services, including EMAA, to the remaining system components. It also contains the code used to implement the Phase V node agent required by EMA on each node. Each of the remaining DECnet/OSI for OpenVMS components makes use of the base image by vectoring through a system cell to the desired function.

Network Item Lists

The DECnet/OSI for OpenVMS modules pass large amounts of data between themselves. This exchange requires an efficient means to encode and move the data. Conversions are expensive operations; therefore a decision was made to use the same structure for all the interfaces within the base components. The structure chosen, a network item list, is a simple length /tag/value arrangement in which the tags are defined in a common area between sharing modules. Network item lists are very easily extended as new functions are added to the software. Since they contain no absolute addresses, they are also position independent. This has the advantage of making it easy to copy or move them when necessary.

Network item lists are used between all VAX communication modules, by EMAA, and by the session services interface. They are also presented to user-written applications through the $IPC interface, thus allowing the interface to be expanded as more protocols and standards are implemented in the DECnet network.

3 Network Management

This section discusses the DECnet/OSI for OpenVMS network management design and network management functions implemented in Phase V.

Network Management Design

The key to Phase V network management design is the EMA Entity Model, which defines the standard management structure, syntax, and interface to be used by each manageable object. The DECnet/OSI for OpenVMS EMA framework is built on this model and defines the components required for a system manager to perform actions on managed objects, both locally and across a network. The EMA framework consists of the following components.

- A director interface, through which user commands called directives are issued
- A management protocol module that carries directives to the node where the object to be managed resides
- An agent that decodes the directive into specific actions and passes that information to the managed object
- An entity, the object to be managed

For a full understanding of the DECnet/OSI for OpenVMS network management implementation, the reader should first understand the EMA model. Details on the EMA model can be found in the paper on management architecture in this issue.[2]

In the DECnet/OSI for OpenVMS network management design, the components and their division of function generally follow the EMA framework. There are, however, a few exceptions. Figure 2 shows the DECnet/OSI for OpenVMS components that implement the EMA model and other Phase V management functions.

The NCL utility provides the EMA director function. The NCL image processes user commands into management directives. It also displays the responses that are returned.

The common management information protocol (CMIP) requester library routines provide part of the management protocol module functions. These include encoding a management directive into CMIP, transmitting it to the designated node, and receiving the response. The CMIP requester routines are implemented as part of NCL, not as a separate management protocol module.

A CMIP listener server process, CML.EXE, provides the remainder of the management protocol module function. It receives a management directive and passes it to the agent. When the agent returns a response, CML transmits the response to the originating node.

The DECnet/OSI for OpenVMS EMA agent, EMAA, accepts management directives from CML, dispatches them to the requested entity, and returns responses to CML. EMAA also extends this concept by actually performing the management directives in some cases.

Entities are not strictly a part of network management. They do, however, receive management directives from EMAA in DECnet/OSI for OpenVMS. They must be able to carry out the directives and return the results of the operation to EMAA.

In DECnet Phase V, an event is the occurrence of an architecturally defined normal or abnormal condition. Events detected by entities are posted to an event dispatcher, which passes them to a local or remote event sink. If remote, a CMIP event protocol is used. In DECnet/OSI for OpenVMS, the event dispatcher image, NET$EVENT_DISPATCHER.EXE, implements the event dispatching and event sink functions.

The data dictionary is a binary compilation of architecturally defined codes for all known Phase V management entities, the manageable attributes of each entity, and the actions that can be performed. It also contains information necessary to encode this information into Abstract Syntax Notation Number 1 (ASN.1), required for the CMIP protocol.

Finally, there is the maintenance operations protocol (MOP). Although MOP is not an EMA component, it is a component of DNA. It performs low-level network operations such as down-line loading and up-line dumping.

Network Management Implementation

The most visible differences between DECnet Phase IV and DECnet Phase V arise from adherence to the EMA architecture. This section discusses the replacement functions implemented in Phase V.

The NCL Utility The network control program has been replaced in Phase V with the NCL utility. NCL provides a highly structured management syntax that maps directly to the EMA specifications for each compliant entity. In an NCL command, the hierarchy of entities from the node entity to the subentity being managed must be specified. For example, the following command shows the local area network (LAN) address attribute of a routing circuit adjacency entity.

```
NCL> Show Node DEC:.zko.Ilium -
Routing Circuit lan-0 Adjacency -
rtg$0002 LAN Address
```

The command contains the node entity name, DEC:.zko.Ilium; the module entity within the node, routing; the name of the circuit subentity of

routing, lan-0; the name of the adjacency subentity of circuit, rtg$0002; and finally the attribute name.

To issue management commands from a DECnet/OSI for OpenVMS system, a user invokes the NCL utility. NCL parses commands into fragments called tokens, containing ASCII strings. It uses the data dictionary to translate these into management codes for directives, entities, and attributes. NCL then constructs a network item list from this information and invokes the CMIP requester send function.

CMIP requester functions are implemented as a set of library routines that are linked with the NCL utility. Underneath this caller interface, the CMIP routines establish a connection over DNA session control to the destination node's CMIP listener. The directive is then encoded into a CMIP message and passed to the destination.

NCL now posts the first CMIP requester receive call. More than one receive call may be needed to obtain all the response data. As soon as a partial response is available, the receive function decodes the CMIP messages into network item lists and passes them back to NCL. NCL translates these into displayable text and values and directs the output to the user's terminal or a log file. If the partial response is not complete, NCL then loops and issues another call to the CMIP requester receive function.

The CMIP requester functions are optimized for the local node case. If the destination node is specified as "0" (the local node), the CMIP requester functions interface directly to the EMAA interface, skipping the CMIP encoding, decoding, and the round trip across the network.

The CMIP Listener The CMIP listener is implemented as a server process, similar to the Phase IV network management listener. When an incoming connection request for CML is received, a process is created to run the CML image. The CML image utilizes the DNA session control interface to accept the connection and receive the CMIP encoded directive. It then uses the data dictionary to decode the message into a network item list. EMAA is then invoked to process the directive and return any required response from the entity. Once CML has received all portions of the response from EMAA, encoded them into CMIP, and transmitted them back to the requesting node, the CML image terminates.

EMAA, the EMA Agent The management structure imposed by EMA contains common directives that must be supported by all entities. A design goal for EMAA was to provide a common management facility with support for common operations such as show or set. EMAA can perform these functions against an entity's management data structures, thereby freeing each entity from separately implementing them and simplifying the entity's code requirements. This approach was successfully implemented, though at the cost of a more complex agent implementation and a set of registration macro instructions colloquially known as the "macros from hell."

The above interface between EMAA and the entities is known as the full interface. Not all development groups' coding entities were interested in this approach; thus, EMAA also provides a basic interface. An entity specifies which interface to use during its initialization when it registers with EMAA. For an entity that uses the basic interface, EMAA

simply passes the directive information to the designated entity and expects response data returned.

The choice of interface must be made by the module-level entity. If the entity uses the full interface, it must register its management structure, including all subentities and attributes, with EMAA. For these entities, EMAA processes the network item list passed by CML. It creates a data structure for each subentity instance, specifying the attributes, any values supplied, and the actions to be performed. EMAA passes this to the designated entity, which uses tables set up during initialization to call the appropriate action routine for the directive. By default, these action routines are set up as callbacks into EMAA itself, thereby allowing EMAA to perform the task. With either the basic or the full interface, a separate response is required for each subentity instance specified by a directive. EMAA calls CML iteratively through a coroutine call to pass response data back to CML.

The Event Dispatcher Phase IV event logging allowed events to be sent to a sink on one node. In Phase V, the event dispatcher supports multiple sinks that can be local or on any number of remote nodes. Event filtering can be applied on the outbound streams of events, filtering events before they are transmitted to a sink. This provides a mechanism to direct different types of events to different sinks.

An event sink is the destination for an event message. A node can have multiple sinks, each accepting events from any number of remote nodes. Event filtering can be applied to the inbound streams of events at the event sink. An event message that passes is sent to the sink, which uses the data dictionary to format it into ASCII character strings. It is then output to the sink client, which may be a console, printer, or file.

An optimization is used when an event is generated on a node and the destination sink is on the same node. In this case, the event bypasses the outbound stream and is queued directly to the event sink. The DECnet/OSI for OpenVMS product, in the default configuration for a local node, defines one outbound stream directed to a sink on the local node and defines the console as the sink client.

An event relay provides compatibility with Phase IV nodes. This important function permits a Phase V event sink to log messages from Phase IV or Phase V DECnet systems. Event relay is a session control application that listens for DECnet Phase IV event messages. It encapsulates each Phase IV event message in a Phase V event message and posts it to the event dispatcher, using the same service that other DECnet/OSI for OpenVMS entities use to post events.

Maintenance Operations Protocol The

NET$MOP process is the DECnet/OSI for OpenVMS implementation of the DNA maintenance operations protocol. MOP uses the services of the local and wide area data link device drivers to perform low-level network operations. MOP can down-line load an operating system image to a VMScluster satellite node and respond to remote requests from a network device to down-line

load or up-line dump an image. MOP also supports management directives that allow a system manager to load or boot a remote device, monitor system identification messages, perform data link loopback tests, or open a terminal I/O communications channel to a device's console program.

The primary design goal of the MOP implementation was to respond quickly and with low system overhead to remote requests from devices to down-line load an image. In some network configurations, a power failure and restoration can cause hundreds of devices to request a down-line load at the same time. The Phase IV implementation was known to have difficulty handling this, so the new implementation of MOP was designed for multithreaded operation. This means there is only one MOP process per node, and it processes multiple concurrent operations by creating a separate thread for each management directive, program request, or dump request received. Moreover, all management data required to service MOP requests is contained in MOP-specific management data structures, designed to be searched quickly. When a request is received, MOP can promptly ascertain whether the required information to service the request is available and make a response.

4 Session Control Implementation

The design of the DECnet/OSI for OpenVMS session control layer is based on goals defined by both the session control architecture and the DECnet user community. These goals include

- Compatibility. The DECnet-VAX product has a large customer base with major investments in DNA applications. The session control layer supports these applications without requiring a relink of the object code.

- Performance. Transmit and receive operations across the network must be as efficient as possible. Minimal overhead is introduced by the session control layer in making each transport protocol available to applications.

- Extensible. The session control layer design allows for future additions to the architecture.

- New features. The session control layer takes full advantage of the new naming and addressing capabilities of Phase V DNA.

- Improved management. The session control layer complies with EMA, allowing it to be managed from anywhere throughout the network.

Session Control Design

The session control layer is divided into several logical components, $QIO, IPC, NETACP, common services, and network management. $QIO and $IPC provide the APIs required to communicate across the network. $QIO is fully compatible with all Phase IV DECnet-VAX applications; however, it does not allow access to the full set of features available in DECnet/OSI for OpenVMS. These new features, and any future additions, are available only through the new $IPC interface.

The two APIs are consumers of session control services provided by the common services component. This component provides all the network functions defined in Phase V to the APIs above it. In order to do this, the common services component makes use of both the NET$ACP and network management portions of the session control layer.

Figure 3 shows the session layer components and their relationships to each other.

Session Control APIs

DECnet Phase IV restricted node names to six characters in length. In DECnet-VAX the $QIO interface was the only means by which an application could make calls to the session control layer. This interface also enforced the six-character name limit. With the advent of Phase V, this restriction no longer applies. It is possible for a node running Phase V to be unreachable by a Phase IV-style six-character node name. As a consequence, the $QIO interface was extended to allow full name representations of a node.

The $IPC interface is a new interface that incorporates all the functions of the $QIO interface, along with extensions made to the session control architecture. This item-list-driven interface provides a cleaner, more extensible interface and allows for easy conversion of $QIO applications. The $QIO interface uses a network control block (NCB) and a network function block (NFB) to hold data. This data is easily mapped to items in a network item list. Also, the function codes used by $QIO can be easily mapped to $IPC function codes. As new requirements arise, supported items can be added to the list without impacting the existing values.

The $IPC interface also supplies some new features not available in $QIO. Phase V DNA uses the Digital Distributed Name Service (DECdns) to store information about nodes and applications in a global namespace. Once an application declares itself in the global namespace, $IPC enables session control to maintain its address attribute. This address attribute contains all the information necessary to define where the application resides on the network. $IPC can then be used by the client side of an application

to connect to a server through a single global name, instead of using a node name and application name pair. This enables the client side of an application to communicate with its server without knowing where the server currently resides.

$IPC supports a new means of accessing a node by its address. In Phase IV, addresses were limited to 63 areas with 1,023 nodes in each area. The address of each node could be represented with a 16-bit integer. The $QIO interface supports a form of node name in which the 16-bit address is converted into the ASCII representation of the decimal equivalent. This is not sufficient to address all Phase V nodes, so a new function called "connect-by-address tower" is available through $IPC. The address tower is discussed further in the Common Services Component section.

Yet another feature of $IPC is the ability to translate a node's address into the name of the node as registered in the global namespace. In Phase IV the address-to-name translation was a management function. Furthermore, the translation was local to the node on which it was performed.

Session Control Network Management

The session control layer makes use of the full EMAA entity interface to support all entities defined by the session control architecture. These include the session control entity itself, as well as the application, transport service, port, and tower maintenance subentities. Each of these entities contains timers, flags, and other control information used by the session control layer during its operation. They also contain counters for the events generated by the session control layer.

The application subentity is of special interest. This entity is the equivalent of the Phase IV object database. It allows the system manager to register an application with session control to make it available for incoming connections. This entity is also used to control the operation of the application and select the types of connections that can be sent or received by it.

Common Services Component

The common services component is the hub for session control. It is responsible for performing tasks that are not specific to the $IPC or $QIO interfaces. These tasks include managing transport connections on behalf of session control users, mapping from a DECdns object name to addresses, selecting communication protocols supported by both the local and remote end systems, maintaining the protocol and address information corresponding to local objects in the namespace, and activating (or creating) processes to service incoming connect requests.

The NET$ACP process is used to provide the common services component with process context. The NET$ACP image itself is nothing more than a set of queues and an idle loop. When the session control layer is loaded, it creates user-mode and kernel-mode tasks. A queue is assigned for each task, and the NET$ACP process attaches to the task when it is started.

When the session component needs to execute in the context of a process and not on the interrupt stack, it builds a work queue entry, queues it to the appropriate task queue, and wakes up the NET$ACP. The NET$ACP finds the address of the desired routine in the work queue entry and executes it. This routine can be located anywhere that is addressable by the process,

but it is usually contained within the session control loadable image. The common services component makes use of the NET$ACP for reading files, creating network processes, and making calls to the DECdns clerk. It also makes use of the process for functions that require large amounts of memory. By performing these tasks in the NET$ACP process, session control is able to use process virtual memory even though it is implemented as an executive loadable image.

The tower set data structure plays a key role in session control. A tower set consists of one or more towers. Each tower represents a protocol stack and is composed of three or more floors, as shown in Figure 4. The lowest floors in the tower correspond to the DNA routing, transport, and session control layers; they are used to identify protocol and associated address information to be used at that layer. When viewed as a whole, the tower set describes a combination of protocols supported on a node. The session control layer on every DECnet/OSI for OpenVMS system not only uses this information to communicate with remote nodes, but is also responsible for building a tower set to represent that local system. Once built, this tower set is placed in the namespace as the attribute for the node.

The session control interfaces allow the user to specify a node in many ways. A node can be specified as a Phase IV-style node name, a Phase IV-style address, a DECdns full name, or a tower set. The three forms of name representations are mapped to the corresponding tower set by making calls to the DECdns clerk to obtain the node's tower set attribute. Once the tower set is in hand, it can be used to communicate with the session control layer on the remote node.

The tower set for a remote node and the tower set for the local node are used in conjunction to determine if both nodes support a common tower. If a common tower is found, session control attempts to establish a connection to the remote node using that tower. If the connection fails, the comparison continues. If another matching tower is found, the connection attempt is repeated. This continues until the connection is established or the tower sets are exhausted.

Use of DECdns

The session control layer uses DECdns objects for all global naming. These objects are used in two different ways: they can represent a node or a global application. A node object is a global representation of a node in a DECdns namespace. Each node object contains attributes that identify the location of a node. Foremost in this list of attributes is the DNA$Towers attribute. The DNA$Towers attribute contains the tower set for the node and is written automatically by the session control layer when DECnet/OSI for OpenVMS is configured and started. Once created, this attribute is updated by session control to reflect the current supported towers for the node.

When the session control layer builds the tower set for the DECdns node object, it creates a tower for each combination of supported protocols and network addresses on the node. If the node supports two transports and three network addresses, the tower set is generated with six towers. It always places the CML application protocol floor on top of the session control floor. The address information for the session control floor is then set to address the CML application. The transport address information is set to address DNA session control, and the routing information of each tower in the set is set to one of the supported network addresses for the node.

The node object DNA$Towers attribute contains data that completely describes the node. Since session control supports node addresses and Phase IV-style node names, soft links are created in the namespace to map from a Phase V network service access point (NSAP) or a Phase IV-style node name (node synonym) to the node object. These links can then be used by the session control layer as alternate paths to the node object.

An application object is a global representation of an application. The DNA$Towers attribute of this object contains a set of address towers used to address the application. The routing and transport floors for each tower in this set are used in the same manner as for the node object. The address information in the session floor, however, addresses the application, not CML. Once set, the information in this tower set is not maintained unless the application issues a register object call through the $IPC interface. If this is done, session control maintains the tower in the same manner as it does for the node object.

5 Transport Implementation

The DECnet/OSI for OpenVMS product supports two transport protocols: the open systems interconnection transport protocol (OSI TP) and the network service protocol (NSP). Each transport protocol, or group of logically associated protocols, is bundled as a separate but equivalent VAX communication module. This approach accomplishes many goals. The more notable ones include

- Isolating each module as a pure transport engine
- Defining and enforcing a common transport user interface to all transports
- Providing extensible constructs for future transport protocols, i.e., providing a set of transport service libraries
- Eliminating previous duplication in adjacent layers (session and network routing layers)

- Providing backward compatibility with existing Phase IV transport protocol engines (NETDRIVER/NSP and VAX OSI transport service)

Transport Layer Design

A transport VAX communication module has two components, a protocol engine and the transport service libraries. The service libraries are common code between modules and are linked together with each engine to form an executive loadable image. The three elements of DECnet/OSI for OpenVMS transport, the NSP protocol engine, the OSI protocol engine, and the transport service libraries, are linked into two images. Figure 5 shows the relationship of these elements.

The specific functions provided by a transport engine depend on the protocol. The generic role of NSP and the OSI transport layer is to provide a reliable, sequential, connection-oriented service for use by a session control layer. The design provides a common transport interface to both NSP and the OSI transport layer. This enables NSP and OSI transport (class 4) to be used interchangeably as a DNA transport. As future transport protocols are developed, they can be easily added into this design.

The DECnet/OSI for OpenVMS transport design places common functions in the service libraries for use by any protocol engine that needs them. Any functions that are not specific to a protocol are performed in these libraries. Separating these functions enables new protocols to be implemented more quickly and allows operating-system-specific details to be hidden from the engines.

The NSP transport VAX communication module operates only in the DNA stack and supports only DNA session control. Due to an essentially unchanged wire protocol, NSP is completely compatible with Phase IV implementations.

The OSI transport VAX communication module implements the International Organization for Standardization (ISO) 8073 classes 0, 2, and 4 protocols. It can operate on a pure OSI stack in a multivendor environment. The OSI transport is also completely compatible with the Phase IV VAX OSI transport service implementation and operates on the DNA stack supporting DNA session control.

Transport Engines The transport VAX communication modules provide a transport connection (logical link) service to the session layer. The connection management is designed to ensure that data on each logical link is handled independently from data on other logical links. Data belonging to different transport connections is never mixed, nor does a blockage of data flow on one connection prevent data from being handled on another.

The transport VAX communication modules are state table driven. Each transport engine uses a state/event matrix to determine the address of an appropriate action routine to execute for any state/event combination. As a transport connection changes state, it keeps a histogram of state

transitions and events processed.

Service Libraries The following functions are common to many protocols and are implemented in the service libraries.

- Transfer of normal data and expedited data from transmit buffers to receive buffers
- Fragmentation of large messages into smaller messages for transmission and the reconstruction of the complete message from the received fragments
- Detection and recovery from loss, duplication, corruption, and misordering introduced by lower layers

The key transport service library is the data transfer library. This library gives a transport engine the capability to perform data segmentation and reassembly. Segmentation is the process of breaking a large user data message into multiple, smaller messages (segments) for transmission. Reassembly is the process of reconstructing a complete user data message from the received segments. To use the data transfer library, a protocol engine must provide a set of action routines. These action routines hold the protocol-specific logic to be applied to the data handling process.

Network Services Phase V provides two types of network services: connectionless (CLNS) and connection-oriented (CONS). CLNS offers a datagram facility, in which each message is routed to its destination independently of any other. CONS establishes logical connections in the network layer over which transport messages are then transmitted.

Transport running over CLNS has a flexible interface. It opens an association with the CLNS layer and is then able to solicit the CLNS layer to enter a transport protocol data unit (TPDU) into the network. When admission is granted, transport sends as many TPDUs as possible at that time. Incoming TPDUs are posted to transport as they are received by the CLNS layer. Both NSP and OSI transports run over the CLNS layer.

Transport running over CONS has a more rigid interface. Once a network connection is established with the CONS layer, each transport request has to be completed by the CONS layer. Thus transport, when running over CONS, is not able to transmit all its TPDUs at once. Each transmit must be completed back to transport before the next can commence. Also, if transport is to receive incoming TPDUs, a read must be posted to the CONS layer. The OSI transport runs over the CONS layer, but the NSP protocol was designed specifically for CLNS and does not operate over CONS.

Differences between Phase IV and Phase V Transport Protocol Engines

Flow control policy is an important difference between the VAX OSI transport service and the DECnet/OSI for OpenVMS implementation. The VAX OSI transport service implements a pessimistic policy that never allocates credit representing resources it does not have. The OSI transport protocol, on the other hand, implements a more optimistic policy that takes advantage of buffering available in the pipeline and the variance in data flow on

different transport connections. It makes the assumption that transport connections do not consume all allocated credit at the same time. Other enhancements to the OSI transport protocol include conformance to EMA network management, compliance with the most recent ISO specifications, and participation in DECnet/OSI for OpenVMS VMScluster Alias.

The two main differences between the Phase IV and Phase V NSP implementations are conformance to the EMA management model, and, once again, flow control. In Phase IV, NSP does not request flow control and uses an XON/XOFF mechanism. This results in large fluctuations in throughput. Phase V NSP has been enhanced to request segment flow control. This mechanism enables each side of a transport to determine when it can send data segments. Due to this difference in flow control policy, Phase V NSP throughput converges to a maximum value.

Future Direction of Transports

The DECnet/OSI for OpenVMS transport design provides a common transport user interface to all transports and places common functions in the transport service libraries. This approach provides extensibility; it allows future transports to be easily incorporated as they emerge in the industry. This common interface can also be used to provide an API that interfaces directly to a transport. DECnet/OSI for OpenVMS engineering is currently looking at providing such an API.

6 Configuration

Design on the new configuration tools was started by collecting user comments about the Phase IV tools and desirable features for the new tool. This data was collected from customer communication at DECUS, through internal notes files, and through internet news groups.

The first goal agreed upon was to create configuration files that are easy to read; inexperienced Phase V network managers may be required to read and understand these files. Next, the tool must be structured. The configuration is divided into several files with recognizable file names rather than one potentially unmanageable one. Each file contains the initialization commands needed to initialize one network entity. Finally, the configuration tool should be extensible. New commands, entities, or other information can easily be added to the configuration.

Configuration Design

The main configuration tool is a DCL command procedure (NET$CONFIGURE.COM). This procedure generates NCL script files, which are executed during network start-up, to initialize the network. In general, each script file initializes one entity within DECnet/OSI for OpenVMS. It is possible,

however, for scripts to contain information for numerous entities. For example, the NSP transport initialization script contains commands to create an instance of the session control transport service provider entity, which enables the session layer to use the protocol. The procedure

can extract information about the configuration by using the NET$CONVERT_ DATABASE utility to translate an existing Phase IV configuration contained in the Phase IV permanent databases. Alternatively, it can prompt the user for the information needed to allow basic operation of the node.

The first time NET$CONFIGURE is executed, all the questions, except for the node's full name and its Phase IV address, have default choices. If the defaults are chosen, the node operates properly once the network has started. When appropriate, NET$CONFIGURE also calls other configuration tools to configure the DECdns client and the Digital Distributed Time Service (DECdts), and to perform various network transition functions.

Once the initial configuration has been performed, customization of components is available. Subsequent execution of the NET$CONFIGURE procedure will present the user with a menu that allows specific subsections of the configuration to be done, for example, adding or deleting MOP clients or session control applications, changing the name of the node, or controlling the use of communications devices.

General help is available while running NET$CONFIGURE. If the user does not understand any individual query, responding with a "?" (question mark) provides a brief explanation.

The scripts created by NET$CONFIGURE are computed. A checksum is computed by NET$CONFIGURE for each script file, and it is stored in a database along with the answers entered for all other configuration questions. This allows the NET$CONFIGURE procedure to detect whether a script has been modified by an outside source. If this condition is detected, NET$CONFIGURE warns the user that user-specific changes made to the particular script may be lost.

If a user has modified the NCL scripts,
NET$CONFIGURE cannot guarantee that the information will be retained after future executions of the procedure. An attempt is made to maintain the changes across new versions. In all cases, previous scripts are renamed before the new scripts are generated. This allows the user to verify that customized change was transferred to the new script. If not, the saved version can be used to manually replace the change.

Node Configuration NET$CONFIGURE asks only one question that is directly related to the node entity. It asks for the node's DECdns full name and sets the node's name. Since the namespace nickname is a required component of the full name answer, it also allows the procedure to determine the namespace in which to configure DECdns.

The node synonym default is generated by using the first six characters of the last simple name of the node's full name. If the user entered the full name, USN:.Norfolk.Destroyer.Spruance.DD125, the synonym default would be

DD125. The user is free to change this information as long as the response is a legal Phase IV-style name. If present, the transition tools make use of this synonym when the node is registered in the DECdns namespace.

Data Link/Routing The NET$CONFIGURE procedure contains a table of all valid data link devices supported by DECnet/OSI for OpenVMS. When the data link/routing configuration module is called, the system configuration is scanned. Any valid devices found on the system are presented to the user for addition to the configuration. The only exceptions are asynchronous data link devices. The user must specifically request asynchronous support for these devices to be configured.

Configuration is mandatory for broadcast data link media since these devices are shareable and users other than DECnet/OSI for OpenVMS may request the device. For synchronous devices, the user has the choice to configure the device for use by DECnet/OSI for OpenVMS. If a device is configured, a choice between the Digital data communications message protocol (DDCMP) or high-level data link control (HDLC) as data link protocol must also be made.

Each data link device configured requires a name for the device and a name for the corresponding routing circuit. The defaults for these names are generated by using the protocol name, e.g., carrier sense multiple access-collision detection (CSMA-CD), HDLC, or DDCMP, along with a unit number. The user may override the default with any valid simple name. This allows the user to set the data link and routing circuit names to be more descriptive in their environment; for example,
HDLC_SYNC_TO_BOSTON for a data link and
CONNECTION_TO_BOSTON_DR500 for a routing circuit.

Transport/Session Control NET$CONFIGURE supports the NSP and OSI transports. The procedure configures both transports by default, but allows the user to select only one. Commands are generated in the start-up scripts to initialize both the transports and the session control transport service provider entity instances, which allow the session control layer to use them.

If OSI transport is configured, default templates are created to allow the installation verification procedures for the OSI applications to operate successfully. The user also has the option of creating specific connection option templates for use with OSI applications.

All default session control applications, e.g., file access listener (FAL), mail, or phone, are configured in the same way as they are with the DECnet-VAX Phase IV configuration tool. The user has the option to allow access to each application through a default account or not. The only queries made by the configuration tool are about the creation of the user account for the application.

DECdts Configuration The DECdts configuration is performed by a call to the DTSS$CONFIGURE procedure. DTSS$CONFIGURE prompts the user to choose between

universal coordinated time (UTC) or local time, which is UTC plus or minus the time-zone differential factor (TDF). If local time is chosen, then the procedure prompts for the continent and time zone on that continent to use. This information is needed to compute the TDF. The DTSS$CONFIGURE

tool creates its own NCL scripts. These scripts are not maintained by NET$CONFIGURE, and no checksums are computed or stored for them.

Configuration To configure DECdns, the network software must be in operation so that the DECdns software may use it. The NET$CONFIGURE procedure attempts to start the network once it has created the necessary scripts. Once the network has been started, the NET$CONFIGURE procedure calls DNS$CONFIGURE, passing it the node full name that was entered by the user. The full name contains the namespace nickname that the user wishes to use. DNS$CONFIGURE then uses the DECdns advertiser to listen on the broadcast media for a name server that is advertising the same namespace nickname. If a match is made, DECdns creates an initialization NCL script with the needed instructions to configure the DECdns clerk at the next system boot. It then tells the advertiser to configure against the same namespace.

If the namespace nickname cannot be matched, the user is given alternatives. First, a list of the current namespaces advertised on the broadcast media, along with the LOCAL: namespace is offered. LOCAL: is a special case used in lieu of the standard client-server configuration. The LOCAL namespace makes use of the client cache to store a small number of nodes locally.

If a choice is not made from the list, the user is queried to see if an attempt should be made to configure to a name server that may be located on a data link other than the broadcast media. If yes, then a valid address must be provided to the DNS$CONFIGURE tool so that it may connect to the name server on the remote node.

If no options are chosen at this point, a final choice of creating a name server on the local node is presented. Since DECnet/OSI for OpenVMS must configure the DECdns clerk, if the answer is still no, the procedure returns to the original list of known namespaces and starts again.

Transition Tools Once DECdns is configured, the transition tools are used to create the correct namespace directory configuration. If a new namespace has been created and selected for use, the tools populate the directories with the node information from the Phase IV DECnet database found on the system. Most often, the tools simply register the node with the DECdns name server along with the node synonym that was provided by the user during the node configuration portion of NET$CONFIGURE.

The transition tools also assist the user when renaming the node or changing from one namespace to another. They copy subdirectory information from the node's old DECdns directory to the new directory structure on the new namespace or within the same namespace, if the user only changed the node's name.

7 Summary

The DECnet/OSI for OpenVMS version 5.5 product implements all layers of the DNA Phase V architecture. This extends the OpenVMS system to a new degree of network access by supplying standard OSI protocols. The product also protects the large investment in network software that OpenVMS users currently hold. This is done by fully supporting the extensive selection of applications available for Phase IV DECnet-VAX. In addition, the design of DECnet/OSI for OpenVMS is structured in a way that will ease the introduction of new standards as they come available.

8 Acknowledgments

Throughout the course of this project, many people have helped in the design, implementation, and documentation of the product. We would like to thank all those people for all their help. We would also like to extend a special thanks to all members of the bobsled team. Without them, this product would never have come to be.

9 References

1. J. Harper, "Overview of Digital's Open Networking," Digital Technical Journal, vol. 5, no. 1 (Winter 1993, this issue).

2. M. Sylor, F. Dolan, and D. Shurtleff, "Network Management," Digital Technical Journal, vol. 5 no. 1 (Winter 1993, this issue).

10 Biographies

Lawrence Yetto Larry Yetto is currently a project and technical leader for the DECnet/OSI for OpenVMS Group. He joined Digital in 1981 and has held various positions in software engineering on development projects for VMS journaling, VMS utilities, and DECnet-VAX Phase IV. He also worked in the Project Services Center, Munich, and was the project leader for the OpenVMS version 5.0 field test. Prior to joining Digital, Larry worked as a systems programmer at Burroughs Corporation. He earned a B.A. in both math and computer science from the State University of New York at Potsdam.

Dorothy Noren Millbrandt Dotsie Millbrandt is a principal software engineer and a co-project leader for Common Network Management. Currently she is developing management components that will work across all the DECnet/OSI platforms: OpenVMS, OSF/1, and ULTRIX. Dotsie was the project leader for the MOP component and the trace facility and has worked on OSI transport and configuration software. Prior to this work, she was a project leader and microcode developer for DSB32 and KMV11 synchronous communications controllers in the CSS Network Systems Group.

The DECnet/OSI for OpenVMS Version 5.5 Implementation

Yanick Pouffary A principal software engineer, Yanick Pouffary is currently the transport technical leader in the DECnet/OSI for OpenVMS Group. She was the principal designer and developer of OSI transport and NSP transport protocol engines. Prior to this work, she developed the presentation layer for the VTX20, a videotext terminal. Before joining Digital in 1985, Yanick worked for the CODEX Corporation on a statistical multiplexer. Yanick earned a B.S. in computer science from the University of Nice, France, and an M.S. in computer science from the State University of New York at Stony Brook.

Daniel J. Ryan Jr. A principal software engineer in the DECnet/OSI for OpenVMS Group, Dan Ryan was responsible for the configuration and installation portion of the DECnet/OSI for OpenVMS product. Recently he was the team leader for the transport development effort. Currently he is investigating DECnet/OSI and TCP/IP integration as well as DECnet/OSI critical problems. Dan has 14 years of experience in data communications and has been with Digital since 1983. He was previously employed as a systems programmer and was a free-lance consultant on computer communication solutions.

David J. Sullivan David Sullivan is a senior software engineer and was the technical leader of the node agent and event dispatcher components for the DECnet/OSI for OpenVMS product. David also worked as an individual contributor on the design and implementation of the session control layer. He is currently working on a development effort to allow the DECnet/OSI product to run on Digital's AXP platforms. After joining Digital in 1987, he worked in the VAX/RPC Group where he was responsible for writing tests for the pidgin compiler. David holds a B.S.C.S. (1988) from Merrimack College.