NIS Q260-1

DMS-100 Family

OSSAIN API

User's Guide

Release 7.0 DRAFT 03.01 April 2000

NORTEL NETWORKS™

DMS-100 Family

OSSAIN API

User's Guide

Publication number: Q260-1
Product release: Release 7.0
Document release: DRAFT 03.01
Date: April 2000

Publication history

April 2000

Version 03.01 Standard release of *OSSAIN API User's Guide*. This document applies to the following produect software suite:

- OSSAIN LET0012 DMS switch software
- Open Automated Protocol (OAP) version 7.0
- OSSAIN Application Programmer's Interface (API) version 7.0

September 1998

Version 02.01 Standard release of *OSSAIN API User's Guide*. This document applies to the following product software suite:

- OSSAIN LET0010 DMS switch software
- Open Automated Protocol (OAP) version 5.0
- OSSAIN Application Programmer's Interface (API) version 5.0

November 1997

Version 01.01 Preliminary beta release of *OSSAIN API User's Guide*. This document applies to the following product software suite:

- OSSAIN TOPS09 DMS switch software
- Open Automated Protocol (OAP) version 4.0
- OSSAIN Application Programmer's Interface (API) version 4.0

Contents

About this document

The *OSSAIN API User's Guide* accompanies the Application Programmer's Interface (API) software for the Operator Services System Advanced Intelligent Network (OSSAIN) product. The book provides the reader with an understanding of the OSSAIN API framework. It describes how to build and configure an application using the API.

The *OSSAIN API User's Guide* is intended for software developers and managers who plan to develop service node (SN) applications using the OSSAIN API. Readers should be familiar with the Open Automated Protocol (OAP) specification and OSSAIN functionality before reading this book. Descriptions of OAP and OSSAIN are contained in the two Nortel documents in Table 1. Refer to the chapters listed for specific information.

Table 1 Nortel OSSAIN documents

Document name and number	Type of information	Chapter content
OSSAIN Open Automated Protocol Specification, Q235-1	Functionality and overview of software	Chapter 1: OAP Introduction Chapter 2: OSSAIN Overview Chapter 3: Protocol Description Chapter 12: Message Flows Chapter 13: Functionality Overview
	High-level	Chapter 4: Message Format Chapter 5: Data Format Chapter 6: OAP Error Handling
	Reference	Chapter 7: Call Processing Operations Chapter 8: Non-Call Processing Operations Chapter 9: Data Block Descriptions Chapter 10: Field Definitions Chapter 11: Error ID Values Appendix B: Message Set

Table 1 Nortel OSSAIN documents

Document name and number	Type of information	Chapter content
OSSAIN User's Guide, 297-8403-901	Functionality and overview of software	Chapter 1: OSSAIN product overview Chapter 2: OSSAIN software functionality
	High-level	Chapter 3: OSAC call processing Chapter 10: OSSAIN maintenance
	Reference	Chapter 7: OSSAIN data schema Chapter 9: OSSAIN billing Appendix A: Data communications

OSSAIN API code reference documentation

The *OSSAIN API Code Reference Guide*, Q260-2, contains reference information on the API C++ software code. This book shows tree diagrams and describes usage, methods, and attributes for all the classes in the OSSAIN API software.

Chapters in this book

Following is a summary of each chapter.

Chapter 1: OSSAIN API product overview

This chapter provides an introduction to the OSSAIN network and the API software.

Chapter 2: Guidelines for SN developers

This chapter provides guidelines for application developers, including information on basic OAP message flows. It also discusses guidelines for testing SN applications.

Chapter 3: OSSAIN API functionality

This chapter describes the components of the OSSAIN API and how they work together. It discusses the API software layers, classes, and objects, and the flow of control in application processing.

Chapter 4: OAP interface

This chapter describes how to use the protocol interface layer of the API.

Chapter 5: Building a basic application

This chapter provides step-by-step instructions on how to build a basic application using the node infrastructure layer of the API.

Chapter 6: Advanced application topics

This chapter discusses advanced application topics such as integrating an existing application with the API.

Chapter 7: Sample SN applications

This chapter illustrates a sample SN application built using all the components of the OSSAIN API.

Chapter 8: Additional API components

This chapter describes additional components of the API. It includes details on the communications classes and utilities.

Chapter 9: Configuration and administration

This chapter discusses the file format of the configuration data and provides an example of a configuration file. The chapter also provides engineering considerations and administration information.

Appendix: OSSAIN OAP Simulator

This chapter describes how to use the OSSAIN OAP Simulator tool.

Appendix: OAP Corba Server

This chapter describes the OAP Corba Server.

List of terms

This chapter lists OSSAIN API terms and definitions.

How OSSAIN API code is represented in this book

The OSSAIN API software is written in C++ code. The names for types, enumerated types, classes, methods, attributes, and files conform to the coding conventions described in this section. Throughout this book, code references appear in 10-point Courier font, shown as follows: sample code.

Note: To avoid symbol conflicts with the application developer's code, any global API identifiers are prefixed with uppercase OA_.

Types

Types are prefixed with uppercase OA_ and appear in all uppercase letters, as in this example:

OA_UWORD

Enumerated types

Elements within enumerated types are prefixed with OA_ and appear in all uppercase letters, as in this example:

```
enum OA_OPID
{ . . .
   OA_OPID_SESSION_BEGIN_INFORM
 . . .
}
```

Classes

Classes are prefixed with OA_ and appear with an uppercase letter beginning each word (no spaces between words), as in this example:

```
OA_SessionPool
```

Methods

Methods appear with a lowercase letter beginning the first word, followed by an uppercase letter beginning subsequent words and ending with parentheses, as in this example:

```
OA_RCODE addSession( OA_AbstractSession* sesn )
```

Note: In this document, references to method names include a parentheses, but may not show a return type or required parameter, as in this example:

```
getStatus()
```

Attributes

Attributes appear with a lowercase letter beginning the first word, followed by a capital letter beginning subsequent words, as in this example:

```
OA_UWORD poolId
```

Filenames

Filenames appear as one to eight alphanumeric characters, as in these examples:

```
oper.h
```

```
oper.c
```

```
oper.mak
```

After installation of the API software, all API header files reside in a single directory. Header files are prefixed with lowercase oa when referred to in an include directive, as in this example:

```
#include “oa/oper.h”
```

Chapter 1: OSSAIN API product overview

OSSAIN allows network-based services to be created and delivered without corresponding DMS switch development. This functionality is achieved by giving the control of calls to an external computer, the service node (SN). The SN directs the switch processing of calls through a standard interface, the Open Automated Protocol (OAP). With OSSAIN, many types of services can be created which include, but are not limited to, applications that use operators or the operator services environment.

The OSSAIN API software minimizes the effort to create applications that use OAP and reside on an SN. The API has the following characteristics that make it easy to use:

- It is portable to a broad range of computing platforms.
- It contains an object library that models key OSSAIN concepts.
- It provides full OAP support, including protocol encoding and decoding, encryption algorithms, and multiple versions of OAP.
- It provides basic OSSAIN maintenance, which can be easily extended for application-specific maintenance requirements.

This chapter gives an *overview* of the OSSAIN API product, including a background of the OSSAIN network and an introduction to the OSSAIN API software. The last section in this chapter provides a *road map* to detailed OSSAIN API information in this book.

OSSAIN network

This section focuses on the following OSSAIN background information:

- AIN and OSSAIN call models
- a simple OSSAIN topology
- a simple centralized OSSAIN (known as OSAC) topology
- additional components in an OSSAIN network

AIN and OSSAIN call models

The standard AIN call model is based on query/response interactions between the switch and the service control point (SCP). The SCP can also direct the call to an intelligent peripheral for voice processing. At various points in the call, the switch determines whether to send a query to the SCP. If the switch sends a query, the call triggers to the SCP. When the switch receives a response, it assumes control of the call and processes the response. Processing the response may require the switch to change its point in call.

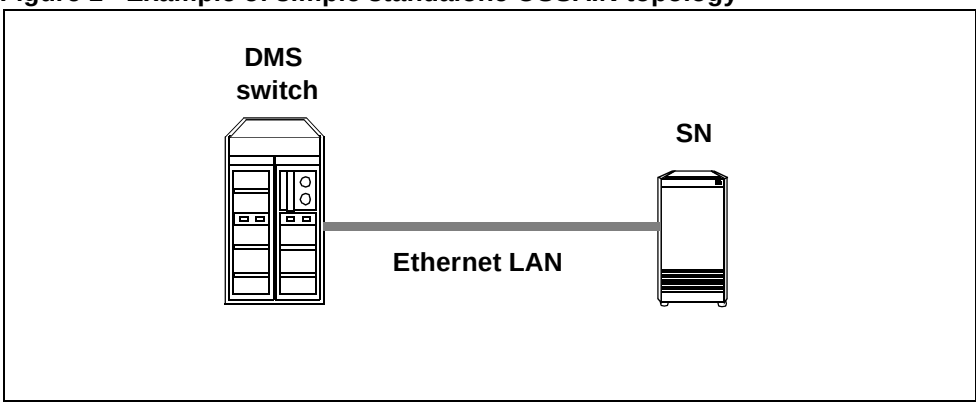
The OSSAIN call model is fundamentally different from that of standard AIN. With OSSAIN, when a call triggers to an SN, the SN has *complete control* of the call. The messaging is not query/response oriented, but rather, request/response oriented. After processing a request, the switch typically does not assume control of the call. It performs the request and responds to the SN, and the SN can then send another request. The SN can send a broad range of requests to the DMS switch. One example is a request for the switch to make a voice connection for voice processing.

Simple standalone OSSAIN topology

Standalone OSSAIN is an architecture in which call processing control is distributed among a single DMS switch and one or more SNs. The switch is responsible for distributing calls, making voice connections, and maintaining the SN. The SN relies on the switch for call processing. The switch and SN have data communication with each other over a LAN using standard Ethernet technology.

Figure 1 shows an example of a simple standalone OSSAIN topology.

Figure 1 Example of simple standalone OSSAIN topology



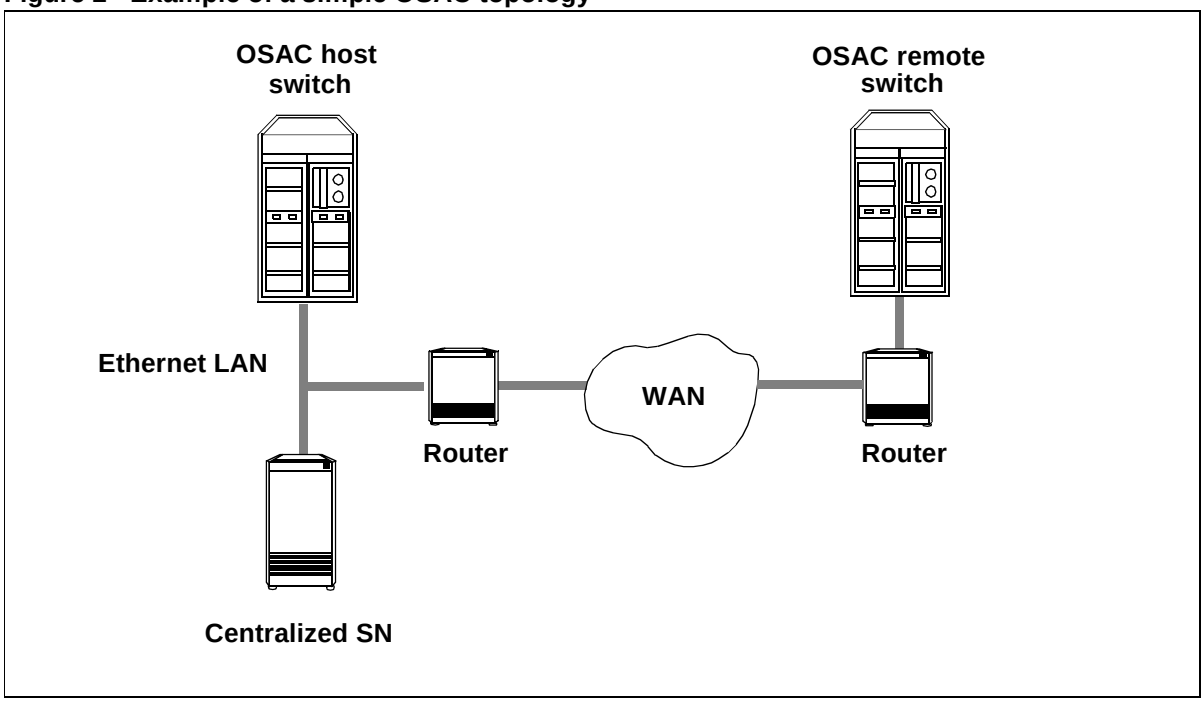
Simple OSAC topology

OSAC is an architecture in which call processing control is distributed among more than one switch and one or more *centralized* SNs. OSAC allows the services offered by a centralized SN to be shared by multiple switches. The switches are set up in a host and remote configuration. The OSAC host switch is responsible for distributing calls, making voice connections, and maintaining the centralized SN.

The OSAC host switch and centralized SN communicate with each other over an Ethernet LAN. WAN technology is used to span longer distances between the OSAC host switch and the OSAC remote switch. The OSAC remote switch can exchange call processing messages with the centralized SN over the WAN and the LAN.

Figure 2 shows an example of a simple OSAC topology.

Figure 2 Example of a simple OSAC topology



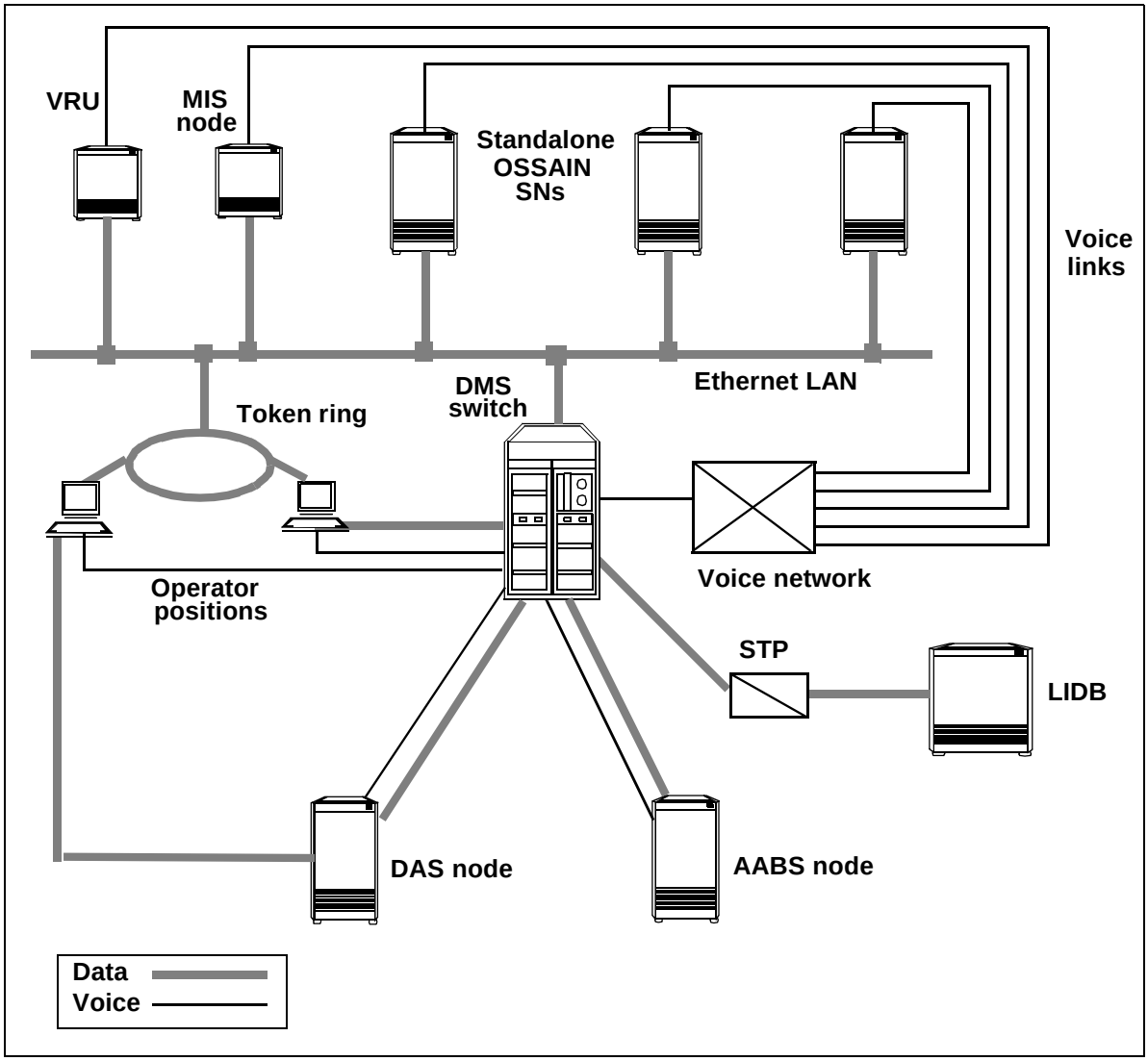
Additional OSSAIN components

The following components are additional and optional in an OSSAIN network:

- voice links
- token ring
- operator positions
- voice response unit (VRU)
- management information system (MIS) node
- directory assistance system (DAS) node
- Automated Alternate Billing Service (AABS) node
- signal transfer point (STP)
- line information database (LIDB)

Figure 3 shows an example of a standalone OSSAIN network with additional components.

Figure 3 Example of a standalone OSSAIN network with additional components



Data connections in standalone OSSAIN

The switch connects to the LAN through an Ethernet interface unit (EIU), which is provisioned on either a Link Peripheral Processor (LPP) or a Fiber Link Interface Shelf (FLIS). SNs communicate with each other over the LAN.

Voice connections in standalone OSSAIN

Basic T1 or E1 trunks provide the voice links between the switch and the SNs.

Open Automated Protocol

OAP is the protocol used for communication between the DMS switch and an SN. The OAP operation set includes a series of request and inform operations with their associated responses, if any. Each request operation is defined with a success response and an error response.

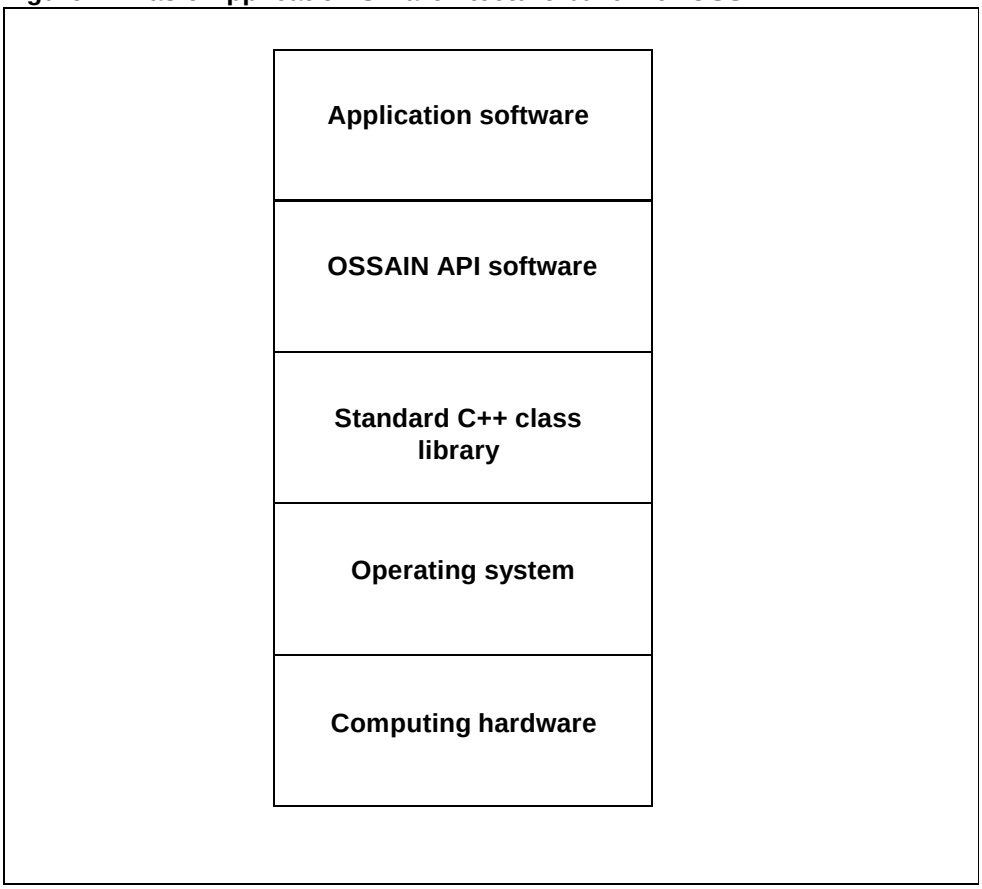
The switch uses OAP to present a call to an SN, inform an SN of call-related events, and perform basic maintenance of an SN. The SN uses OAP to request specific actions from the switch while retaining control of a call.

Note: Refer to Chapter 2: "Guidelines for SN developers" for more information OAP messaging. For complete details on OAP, please refer to *OSSAIN Open Automated Protocol Specification*, Q235-1.

SN architecture

Figure 4 shows the architecture of an SN that is built with the OSSAIN API.

Figure 4 Basic Application SN architecture built with OSSAIN API



The next paragraphs give a brief explanation of each SN component.

Application software

The third-party developer provides the application software.

OSSAIN API software

The OSSAIN API software consists of a set of C++ classes that incorporate basic concepts from the OSSAIN switch software. Ready-made objects provide the application with a high-level interface to the DMS switch. This guide describes the OSSAIN API software.

Note: The *OSSAIN API Code Reference Guide*, Q260-2, provides details on all the classes in the API software.

Standard C++ class library

The OSSAIN API is portable and is built on the ANSI standard C++ class library from the platform chosen by the user.

Operating system

The OSSAIN API can be used with commercially-available operating systems that support User Datagram Protocol (UDP) and Internet Protocol (IP).

Computing hardware

The hardware for the SN consists of a commercially-available platform that supports 32-bit processing and Ethernet connectivity.

OSSAIN API software

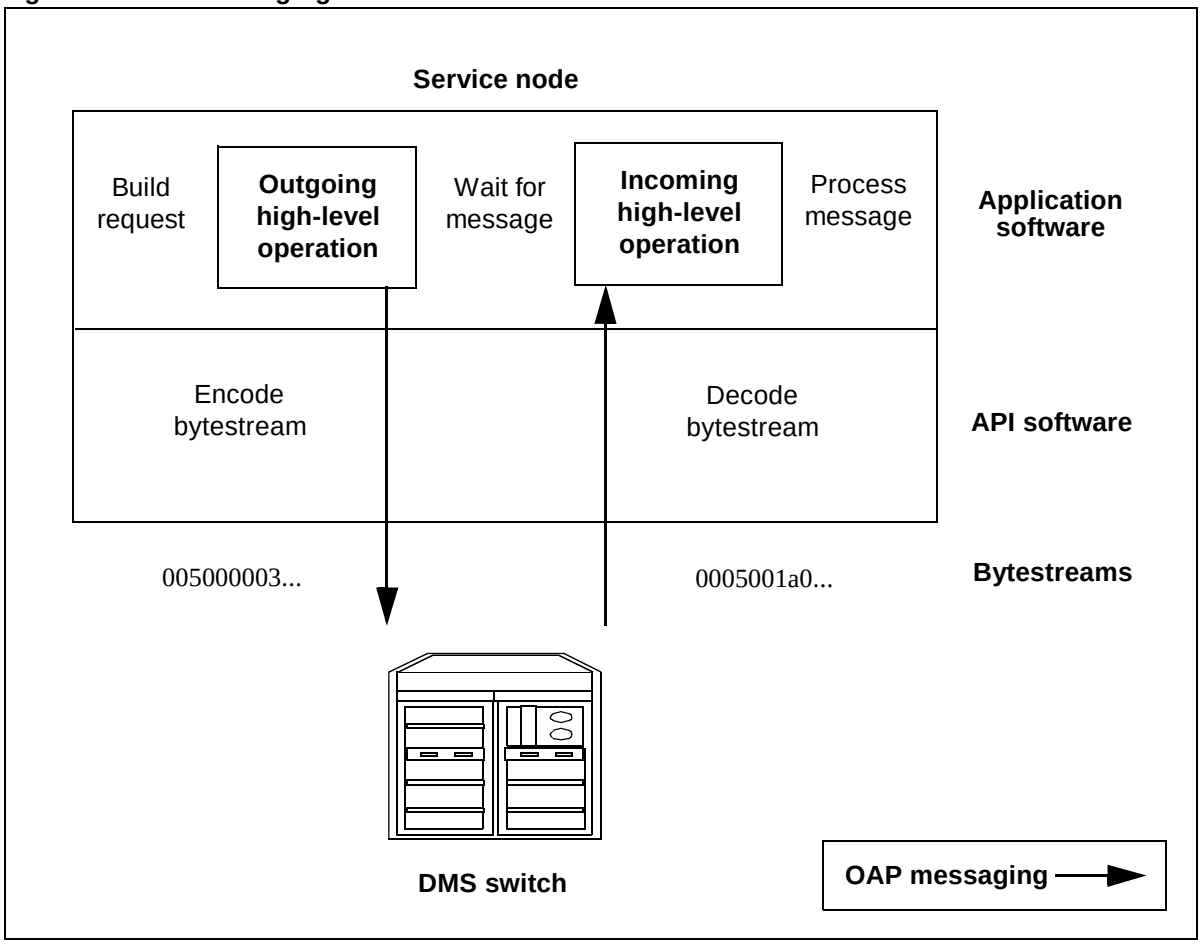
OSSAIN API software allows an SN application to communicate with one or more DMS switches. Data messages are exchanged using OAP.

The API software also allows an SN application to communicate with another SN in the network. Data messages can be exchanged using any protocol. Both types of communication are discussed in this section.

SN to switch communication

Figure 5 shows the high-level OAP messaging between the SN and the switch. When the SN application builds an outgoing high-level OAP operation (such as a request), the API converts it into a bytestream and sends it to the switch. After the API receives a message from the switch, it converts the bytestream into an incoming high-level OAP operation.

Figure 5 OAP messaging between SN and switch



SN to SN communication

The application can use the API to encode and decode OAP messages when communicating with another SN (such as a calling card database). If a protocol other than OAP is needed, the developer must provide the application with its own routines to encode and decode messages.

API layers

The OSSAIN API is organized into three software layers.

- The *base layer* provides low-level utilities for communication and datafill configuration, and the building blocks required by the other layers.
- The *protocol interface layer* provides the high-level interface to the OAP. The API uses the protocol interface layer to encode, decode, and validate OAP messages. This layer requires the base layer.

- The *node infrastructure layer* provides well-defined objects that correspond to the node, session pool, and session components in OSSAIN software. This layer also provides the structure and flow of control of API applications. This layer requires both the protocol interface layer and base layer.

Chapter 3: "OSSAIN API functionality" discusses each OSSAIN API layer in detail.

OSSAIN concepts in the API

OSSAIN introduces the basic concepts of *session, session pool,* and *node*. In OSSAIN software, these concepts are defined as follows:

- A session provides a service to a call. The DMS switch distributes a call to a session.
- A session pool is a group of sessions that provide the same type of service. Datafill at the DMS switch associates a session pool with a particular node.
- A node is a network component that participates in call processing.

Note: For detailed information on OSSAIN switch software, please refer to *OSSAIN User's Guide*, 297-8403-901.

These basic OSSAIN concepts are shared by the OSSAIN API software. An API object represents each OSSAIN concept. The next paragraphs contain a high-level definition. More information is in Chapter 3: "OSSAIN API functionality."

Session object

The API uses a session object to provide call processing.

Session pool object

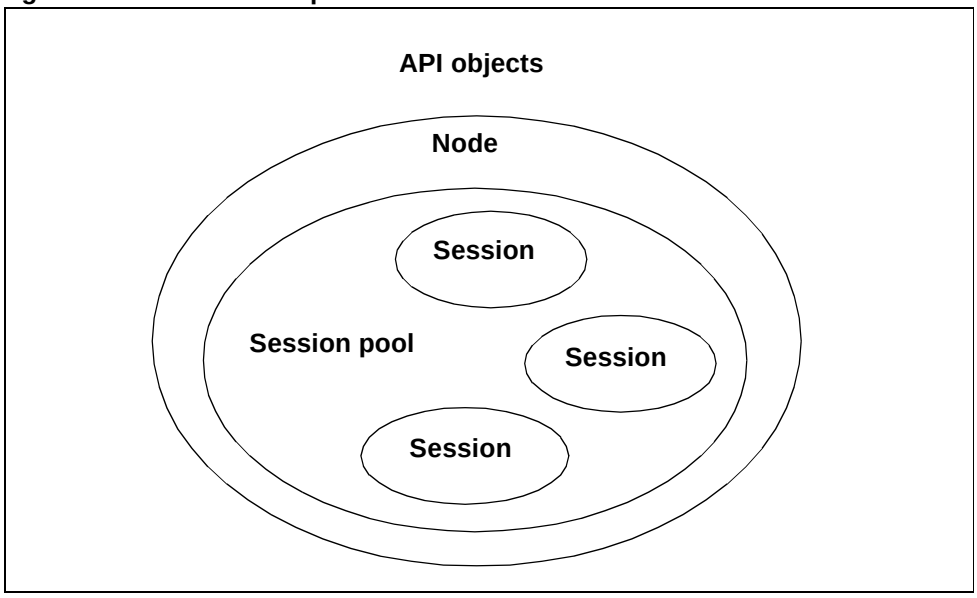
The API uses a session pool object to perform session pool maintenance and manage sessions. A session pool object contains up to 1023 session objects.

Node object

The API uses a node object to perform node maintenance and manage session pools. A node object contains one or more session pool objects.

Figure 6 shows a high-level view of the API objects for the node, session pool, and sessions.

Figure 6 OSSAIN concepts in the API



Additional components of the API software

The API software provides additional components that interact with OSSAIN switch processing. This section gives an overview of these components. Details are in Chapter 8: "Additional API components" and Chapter 9: "Configuration and administration."

OMs

The API collects operational measurements throughout the life of the application and displays them in real time. The API provides an OM utility to collect measurements specific to the API and the application.

Logs

The API provides a log utility to record events specific to the API and the application.

Audits and maintenance states

The API provides routines for default maintenance-handling. The routines respond to DMS switch audits and enforce in-service state and busy state behaviors.

Security

The API protocol interface layer can encrypt and decrypt sensitive data (such as calling card numbers) in certain OAP fields.

Configuration utility

Some OSSAIN datafill needs to be coordinated in both the switch and the SN. The API provides a configuration utility to datafill the SN. The developer is responsible for creating a configuration file used by this utility. This file contains data that matches the switch-defined datafill.

OAP Corba Server

New in release 7.0 of the OSSAIN API is the OAP Corba Server application. The OAP Corba Server is an OSSAIN API based application that provides a CORBA based interface for other applications to use to exchange OAP messages with a DMS. The OAP Corba Server is one of the common components of the Interactive Call Server (ICS) platform. Alternatively, the OAP Corba Server can be configured to run without other ICS components. The OAP Corba Server is fully described in Appendix B of this user's guide.

OSSAIN API information road map

The following road map is a guide to the location of specific information in the *OSSAIN API User's Guide*.

- Chapter 2 provides guidelines for SN application developers, including information on basic OAP message flows. It also discusses guidelines for testing SN applications.
- Chapter 3 describes the components of the API, including the software layers, classes, and objects, and the flow of control in application processing.
- Chapter 4 describes how to interface with OAP using the protocol interface layer of the API.
- Chapter 5 describes how to build an application using the node infrastructure layer of the API. It provides detailed instructions and code examples.
- Chapter 6 discusses advanced application topics such as integrating an existing application with the API.
- Chapter 7 illustrates a sample SN applications.
- Chapter 8 describes the components of the base layer. It includes details on the communications classes and utilities provided by the API.
- Chapter 9 discusses the file format of the configuration data and provides an example of a configuration file. It also includes engineering and administration considerations.
- Appendix A describes how to use the OSSAIN Simulator tool.
- Appendix B describes the OAP Server application.

Chapter 2: Guidelines for SN developers

Before beginning to design an SN application, developers need to understand the Open Automated Protocol (OAP) message flows for call processing and maintenance. This chapter focuses on the following information:

- OAP message types
- OAP conversations
- SN application call flow
- OAP call processing message flow
- OAP maintenance message flow

The last sections discuss OSSAIN voice links and guidelines for testing SN applications.

OAP message types

The following OAP message types are exchanged between the SN and the switch.

- *Inform* messages update the message receiver of an SN maintenance event or call event (such as a subscriber party going on hook), or request an action that does not require a response. An inform message is not responded to unless there is a protocol violation.
- *Request* messages request that a specific operation be performed by the message receiver (such as connecting a voice link). The receiver responds to the request with either a success response message or an error response message.
- *Success response* messages indicate that the requested operation was successfully performed by the receiver of the request operation (for example, a voice connection was successful). A success response message is not responded to unless there is a protocol violation.

- *Error response* messages indicate that the operation request was attempted, but failed. An example of an error response is when a voice connection fails because the voice channel sent with the message is invalid. A set of possible errors is defined for each operation request message. An error response message is not responded to unless there is a protocol violation.
- *Reject* messages indicate that there is a protocol violation with a message or that the message could not be parsed. If the message is a request or inform type, an attempt is *not* made to perform the operation. A set of protocol violations is defined for OAP.

Note: For complete details on OAP, please refer to *OSSAIN Open Automated Protocol Specification*, Q235-1.

OAP conversations

A *conversation* refers to a sequence of OAP messages between the SN and switch. A call processing conversation begins when a call is associated with a session. It ends when the session is released from the call. Call processing conversations can be initiated by either the switch or the SN. Likewise, they can be ended by either the switch or the SN.

A node maintenance processing conversation begins when an SN is brought into service by the switch. It ends when the SN is taken out of service by either the switch or the SN. Only one node maintenance conversation at a time is active for the node.

Likewise, a session pool maintenance processing conversation begins when a session pool is brought into service by the switch. It ends when the session pool is made busy by either the switch or the SN. Only one session pool maintenance conversation at a time is active for a session pool.

Determining the call flow of the application

Determining the call flow is central to SN application design. With OSSAIN, the SN application controls the call flow; it directs the DMS switch and other network components to service the call.

Determining the call flow consists of the following steps:

- drawing a network diagram
- determining the success path
- determining the call handling for errors

Drawing a network diagram

The network diagram should show all the application components. These include telephony switches, one or more SNs, data connections, and (optionally) voice trunks. The DMS switch with OSSAIN software and the SN with the OSSAIN API and application software are required components.

Additional components of the network diagram may include other switches (such as end office, DMS-250, or Meridian), the caller, other subscribers, and any service agents for human interaction or backup.

Optional network components include databases and voice response units (VRU). These nodes can already be present in the network (such as a directory assistance database) or they can be introduced into the network by the new service (such as a VRU that plays announcements to the caller or analyzes their keypad and voice selections).

Determining the success path

The order of events for the success path (or paths) of the application can include any of the following functions:

- initiating a call by the subscriber or at the SN
- establishing data connections and sending data
- segregating call traffic based on call information (such as the called number or the selected language of the caller)
- establishing and releasing voice connections
- querying existing databases (such as LIDB or DA), or databases that are introduced into the network for the new service
- playing announcements or collecting subscriber information
- establishing a broadcast voice connection (for example, for playing music on hold to several subscribers)
- connecting a second subscriber to the call or conferencing a third subscriber
- connecting a customer service agent or operator for subscriber interaction
- determining or validating billing for the service
- transferring the call and a 50-byte block of application-definable data to another node in the network
- notifying the SN application of call events such as subscriber on hook or off hook, flash, octothorpe (#), or DTMF digits throughout the call
- appending predefined or custom billing information to the billing record

Note: The application success flow is the basis of the software state machine of the OSSAIN API application. The call and network functions in the previous list correspond to OAP operations defined in the OAP Specification. To become familiar with specific OAP operations, refer to “Call Processing Operations” in *OSSAIN Open Automated Protocol Specification*, Q235-1.

Determining the call handling for error paths

While failures should be rare for a robust application, the application must be prepared for errors at any point in the call flow. An important part of the application design is to determine a reasonable error treatment.

For example, consider the following questions:

- How should the call be handled if there is a failure to obtain a voice connection to a VRU? Should the call be routed to an operator or customer service agent? Should ringing be provided to the subscriber?
- How should the call be handled if there is a lost OAP message while conferencing a third party? Should the application retry once or twice? Should the application retry at all?

Note: The possible error cases and their handling are also incorporated into the software state machine for the application. Success and failure responses for request operations are listed in *OSSAIN Open Automated Protocol Specification*, Q235-1.

OAP call processing message flow

The basic message flow between a switch and an SN controling the call can be divided into the following three areas:

- beginning a session
- servicing a call
- ending a session

Beginning a session

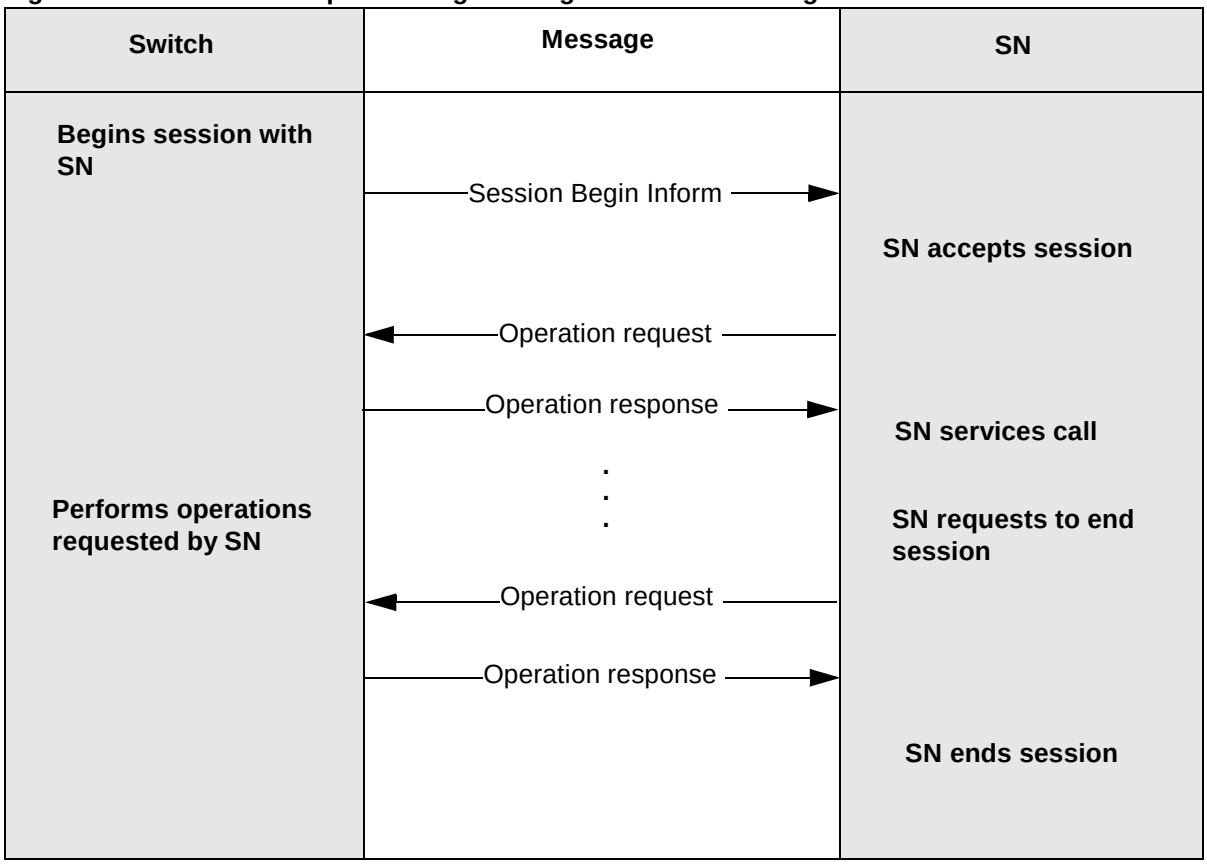
A session between the switch and SN is either originated by the switch or initiated by the SN. When a switch determines that an OAP conversation is required, it selects a session to an SN that provides the service. The switch sends a Session Begin Inform operation message to the SN.

Note: For details on how the switch selects a session, please refer to routing and queuing information in the *OSSAIN User's Guide*, 297-8403-901.

The SN can choose to either accept the call or not accept the call. If the SN accepts the call, it begins by providing the service to the call, which may result in sending requests to the switch to perform required operations. If the SN does not accept the call, it releases itself from the call.

Figure 7 shows an example of a basic OAP call processing message flow for a switch-originated call.

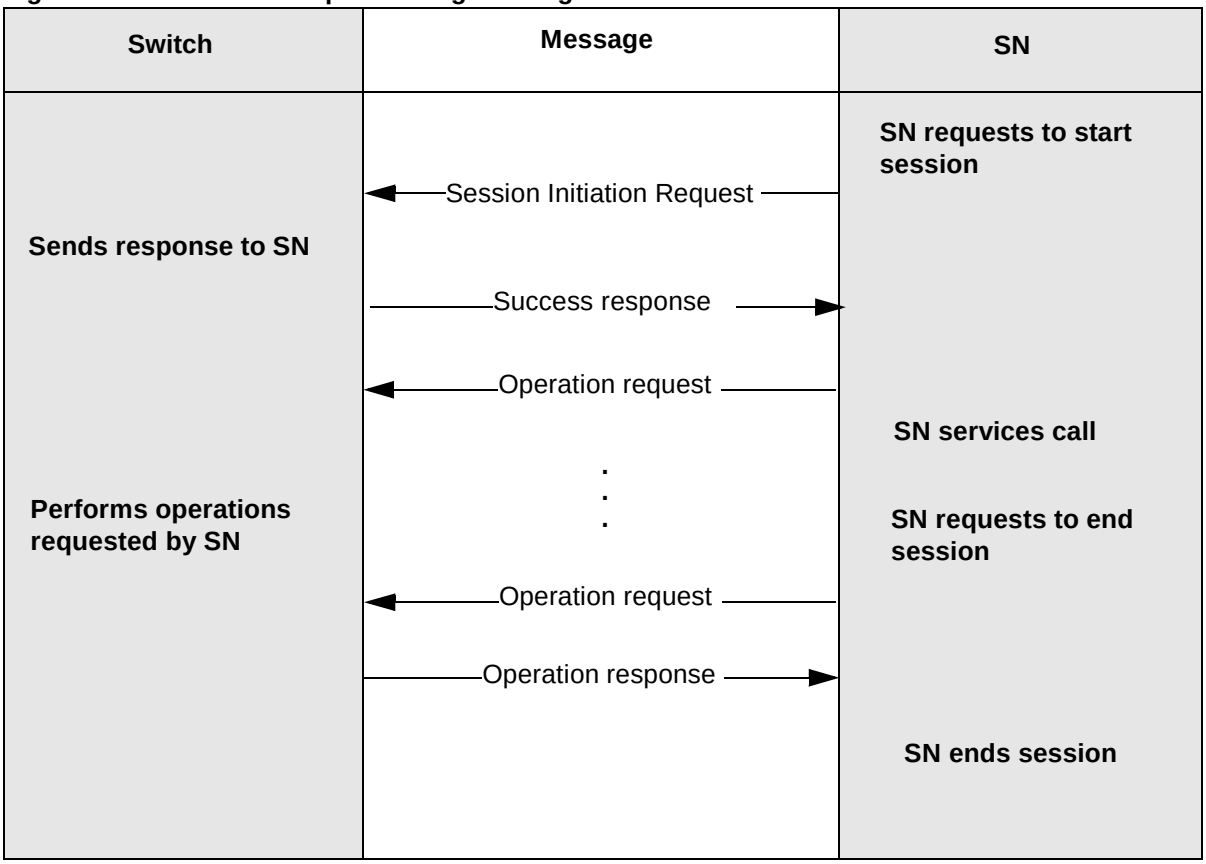
Figure 7 Basic OAP call processing message flow—switch-origination



An SN can initiate a session to the switch by selecting a session and sending a Session Initiation Request operation message to the switch. The switch returns a response message indicating whether or not the switch can begin the session.

Figure 8 shows an example of a basic OAP call processing message flow for an SN-initiated call.

Figure 8 Basic OAP call processing message flow—SN-initiation



Servicing a call

The SN controls the call while the session is active. The SN can perform any application-specific tasks it needs, such as querying a database, playing a prompt, and collecting digits. The SN can send requests to the switch to perform tasks, such as establishing a voice connection, connecting a subscriber to a call, and transferring a call.

Once the switch receives a request, it attempts to perform the operation. If the switch can successfully perform the operation, it returns a success response message to the SN. If the switch fails to perform the operation, it returns an error response message. The appropriate error reason appears in the Error ID field in the operation header of the error response message.

If the switch is unable to parse the message or detects a protocol violation, it may send a reject message to the SN. The switch may send an inform message at any time during the call to indicate important new events, such as an on hook, off hook, or flash.

Ending a session

The session between the SN and switch ends by request of the SN or by the switch. Once the SN has completed its tasks for the call, it releases itself from the call.

A common way for the SN to release itself is to *float* the call. Float is a process where the SN releases itself from the call before or after it establishes a connection between the parties. Another way for the SN to release itself is to transfer the call to an operator or an automated system, such as another SN or a TOPS automated system.

Likewise, the switch can send an inform message to the SN that indicates the end of the call, or indicates the SN was released from a simultaneous connection.

Additional call flow considerations

This section discusses additional call flow considerations.

Triggers

An OSSAIN trigger is an event that causes a call to be redirected to an SN, operator, or automated system. When a call triggers to an SN, the SN receives a Session Begin Inform message from the switch and takes control of the call.

Triggers also can be set up to inform but not give control to the SN. A Trigger Event Inform message sent to the SN does not give control of the call to the SN. Instead, this inform message reports an event about a call from which the SN has already been *released*. The SN should not attempt to make any new requests to the switch regarding that call.

Positive assertion

The SN may receive a Session Begin Inform message for a session that the SN considers to be active with an existing call. In this event, the SN should consider the first call to be ended and respond to the Session Begin Inform as it would to a new call.

Timeouts

The SN should monitor request messages that have timed-out while waiting for a response. The application should determine when it is appropriate to resend the request or perform some other action.

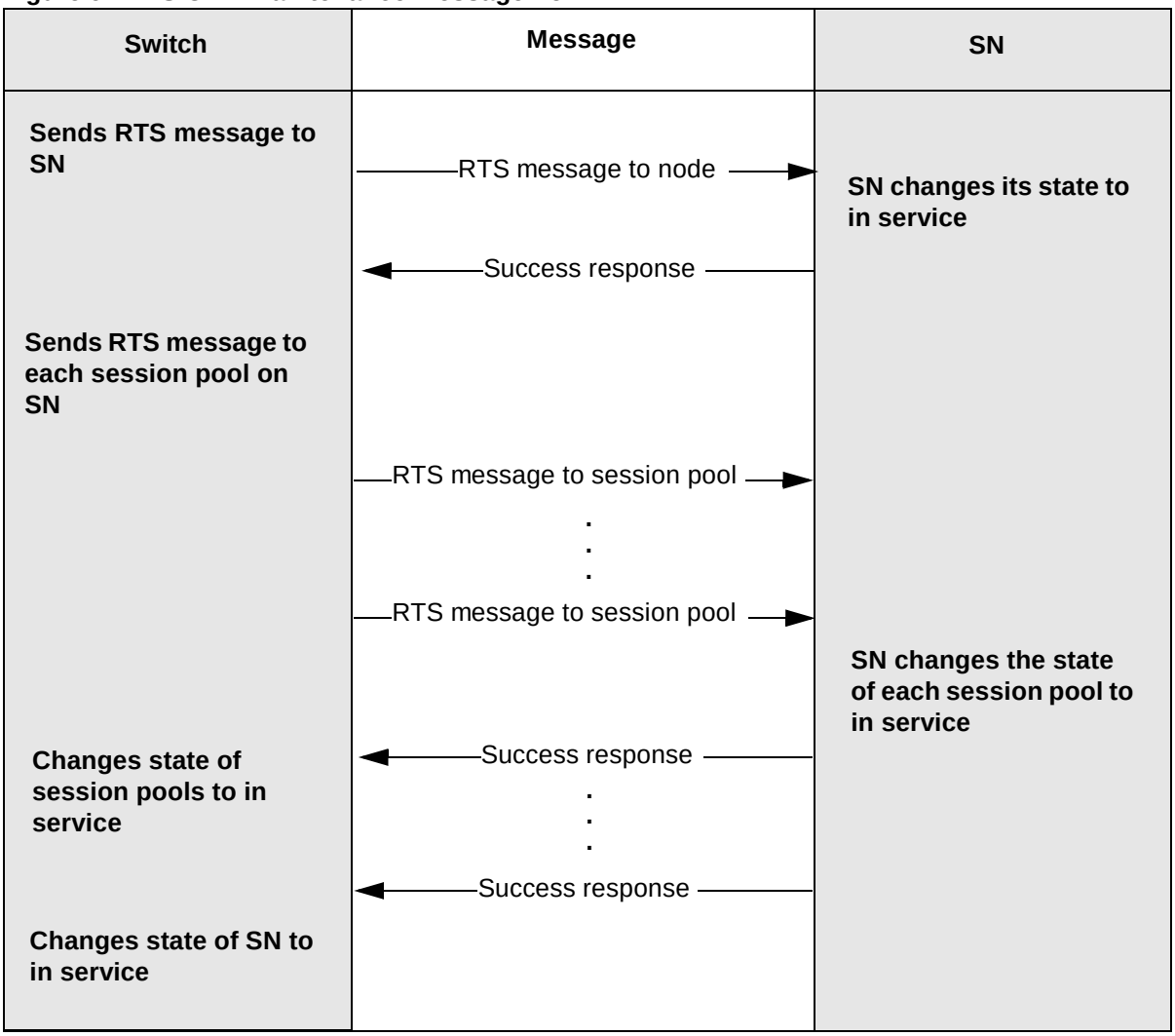
OAP maintenance message flow

SNs respond to requests from the switch for maintenance activities. SNs are responsible for the following tasks:

- responding to switch-originated maintenance requests regarding the node, such as a return to service (RTS) request for the node
- responding to switch-originated maintenance requests regarding a session pool on the node, such as an RTS request for the session pool
- responding to node and session pool audits to ensure that communications exist with the switch

Figure 9 shows an example message flow when the SN is manually returned to service at the switch.

Figure 9 RTS OAP maintenance message flow



Protocol version negotiation

The SN is responsible for updating the switch with the maximum protocol version the SN supports. When the switch sends a maintenance request message at a protocol version *higher* than the SN can support, the SN responds with a Protocol Violation Error (PVE) reject message that includes the maximum protocol version it does support. The switch then performs protocol version negotiation and sends another maintenance request message at a version supported by the SN. The SN sends a success response to the switch.

When the switch sends a maintenance request message at a protocol version *lower* than the maximum but still supported by the SN, the SN adjusts and responds using the lower protocol version.

Note: For complete details on OAP protocol version negotiation, please refer to "Protocol Description" in *OSSAIN Open Automated Protocol Specification*, Q235-1.

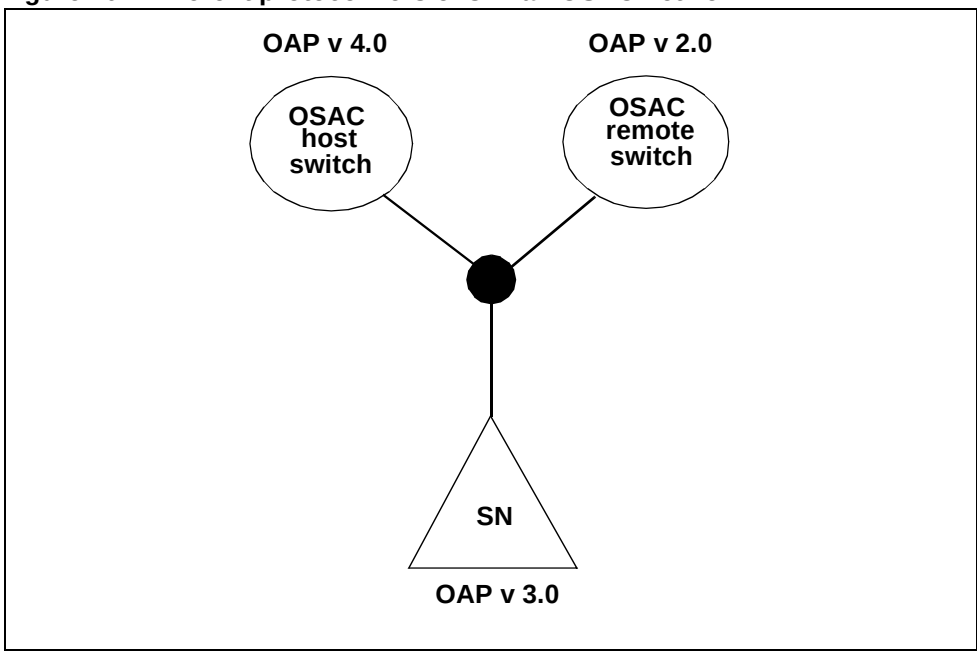
Centralized OSSAIN (OSAC) network

OSAC host switches (and standalone host switches) determine the optimum OAP protocol version for call processing with a node during a maintenance conversation. OSAC remote switches, however, do not have a true maintenance conversation with centralized SNs. As a result, the rules for determining the optimum call processing protocol version are different for a remote switch than for a host switch.

The SN is responsible for storing the optimum protocol version for each OSAC switch it communicates with. Once the optimum protocol version is determined, it should be used until the OAP conversation is over.

In an OSAC network, the optimum protocol version potentially can be different for each switch the SN communicates with. Refer to Figure 10 for an illustration.

Figure 10 Different protocol versions in an OSAC network



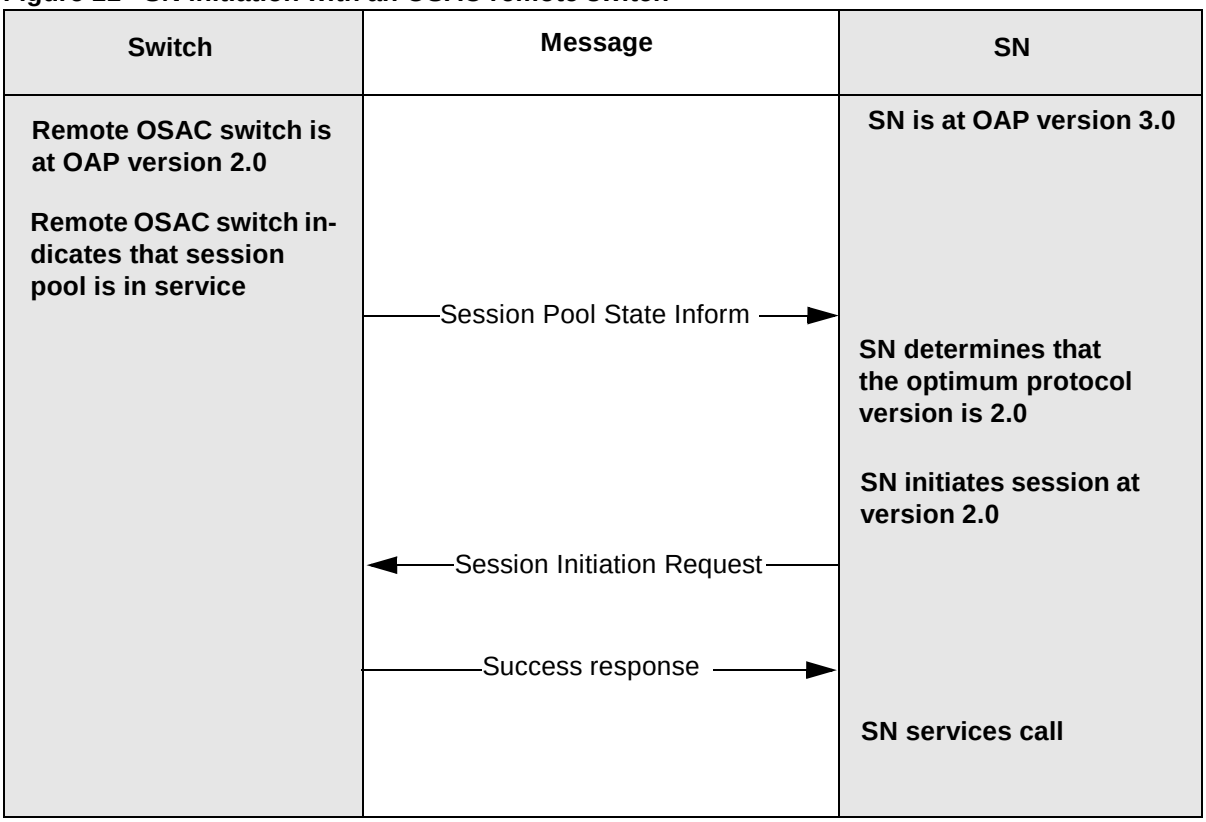
In the OSAC network shown in Figure 10, a single session pool can communicate with multiple switches. However, an individual session communicates with only one switch for a particular call.

Before an SN can initiate a session with an OSAC remote switch, the SN must receive an OAP Session Pool State Inform message. This message indicates that the session pool is in service with respect to the OSAC remote switch. Until the OSAC remote switch sends this inform message, it ignores any request from the SN to initiate a session. This inform message also is used to determine the optimum protocol version for *all* sessions in the session pool with that OSAC remote switch.

Note: For details on OSAC call processing, refer to *OSSAIN User's Guide*, 297-8403-901.

Figure 11 shows the OAP messages in an SN initiation with an OSAC remote switch. In the example, the OSAC remote switch is at OAP version 2.0 and the SN is at OAP version 3.0.

Figure 11 SN initiation with an OSAC remote switch



SN maintenance requests

The SN also can make maintenance requests for the following tasks:

- generating logs at the switch on a node basis or a session pool basis
- generating an alarm at the switch on a node basis or a session pool basis
- changing the number of sessions available for call processing
- changing the maintenance state of the SN to busy

Real-time considerations

The SN application must respond to maintenance requests from the switch within ten seconds. The only exceptions to this are the RTS and test requests, which have a time limit set in the DMS switch table OANODINV (OSSAIN Node Inventory).

OSSAIN voice links

As of OAP version 4.0 there are two main categories of voice links:

- two-way, non-broadcast voice links, referred to as **standard** voice links.
- one-way, **broadcast** voice links.

Note: All voice links prior to version 4.0 were of the two-way, non-broadcast type.

In addition, as of OAP 6.0 standard voice link operation can be further categorized by their method of selection:

SN requested/SN selected - explicit Voice Connect Request sent by SN, selection of voice circuit by SN from its own pool of voice links.

Each service node that is selecting standard voice links is associated in switch datafill with its own set of voice channels. This set of voice channels is considered as a resource that is 'owned' by a particular Service Node, at least as far as the switch is concerned. For standard SN-selected voice links, an SN is only allowed to use its own voice link resources.

When a service node requests a voice connection for a particular session, it must specify a logical voice channel number. The node ID of the SN, along with the logical voice channel number, are mapped to an actual voice circuit at the switch using table OAVLMAP. The switch then connects the voice circuit to the call and sends back a success response.

Note: Voice link resource owning is only a concept that the switch uses to map a logical voice channel to a voice link circuit. The DMS makes no assumptions about how voice circuits are terminated in the network. Voice circuits (either standard or broadcast) can be connected to the service node itself, to another service node, or to any other external system.

SN requested/switch selected - explicit Voice Connect Request sent by SN, selection of voice circuit by switch from a group of voice links associated with the SN and function.

When the SN sends in a request for a voice link but does not specify a logical channel number, the switch will select the voice link. The voice circuit is selected from a trunk group associated with a particular node and function in table SNVLGRP. The switch selects an available member, connects it to the call and informs the SN of its selection in the Voice Connect Success Response.

Switch auto voice link selection - automatic voice link selection and connection of voice circuit by the switch at the start of the session.

In this case, the call arrives at the SN with a voice link already attached. Functions can be datafilled in the switch to automatically select and connect a voice link to the call prior to starting the session with the SN. The voice circuit is selected from a trunk group associated with the node and function in table SNVLGRP. The switch selects an available member, connects it to the call and informs the SN of its selection in the first message of the session.

Broadcast voice links are limited to the following method of selection (OAP 4.0 and above):

SN requested/SN selected - explicit Voice Connect Request sent by SN, selection of voice circuit by SN from its *own* pool of voice links or from *another* SN's pool of voice links.

For broadcast connections, an SN is allowed to request the voice channel resources of another Service Node. Whenever a Service Node requests a broadcast voice connection for a particular call, it must specify a node ID (indicating the voice link resource owner) as well as the logical voice channel number. This is true whether it is using its own or another SN's voice links. By allowing Service Nodes to share broadcast voice link resources, common recordings such as those that play music or provide a branding announcement can be used by several different nodes.

For broadcast voice links, the received node ID (voice link resource owner) and logical voice channel are mapped to the actual voice circuit in table OAVLMAP. The switch then connects the voice circuit to the call and sends back a success response.

Switch/Service Node parallel functionality/datafill

It is important that the switch and the Service Node agree on a voice link's functionality with regards to the following:

- *the type of voice link* (broadcast or standard). Due to switch network configuration constraints, individual voice links can function as only one type or the other. In the switch this is determined by the BCST field in table OAVLMAP.
- *the method of selection* (SN or switch). SN selected voice links must be datafilled in table OAVLMAP, while switch-selected voice links must have the trunk group associated with the service node and function defined in table SNVLGRP.
- *the point at which the voice path is cut through on broadcast voice links.* The switch can allow speech transmission from the voice link to be cut through immediately (controlled by setting the CUTTHRU field in table OAVLMAP to IMMEDIATE) or it can wait for a signal from the voice trunk (a momentary offhook) before cutting through speech (controlled by setting the CUTTHRU field to HKCHG).

- *the maximum number of connections that can be made to a single broadcast voice link.* Again, because of hardware limitations in the switch network, the total amount of broadcasting connections must be factored into the engineering of the switch. For this reason the number of broadcast connections is stored on a per voice link basis in field MAXCONNS in table OAVLMAP. This value can range up to 1023 for trunks using the IMMEDIATE method. For trunks using HKCHG, the maximum allowable is 255.
- *who controls the release of a broadcast voice link*. The option used is dependent on information received in the OAP Voice Connect Request and can change from call to call for an individual broadcast voice link. Two options exist:
  - — the Service Node can request the release
  - — the switch can release the voice link independently of the Service Node.

Standard voice link operation

When voice processing is required to collect information from a party involved in a call at a Service Node, the call is connected to an individual SN voice link. In this situation, a unique, two-way Network connection is necessary in order for the SN to obtain any voice or DTMF input from the call participants. Standard voice links are used for this purpose.

Note: The exact speech path status (one-way, two-way, zero-way) of each port in an OSSAIN call is controlled by the Service Node.

For SN-selected voice links, the switch will set up a two-way connection when mapping of the logical channel and node ID result in a voice link that has the BCST filed set to N. For switch-selected voice links, the switch will setup a two-way connection after selecting an idle voice circuit from a group of circuits associated with the node and function.

Release of a standard voice link during a session is under the control of the Service Node. An exception to this rule is if the switch detects a problem on the circuit and has to release it. In that case, the SN will receive a Voice Release Inform message.

Signalling on standard voice links

Information signalled on standard voice links is limited to the ‘Available/Not Available’ status of the voice circuit. This information is transmitted as either an onhook (available) or an offhook (not available). If either side goes offhook the voice link is no longer available for use by call processing. If the offhook is from the SN the switch puts the voice link into the RMB (Remote Make Bsy) state. It remains RMB until an onhook is detected from the SN. Likewise, a voice link that is offhook in the switch must remain unused by the SN until the switch sends an onhook. If the SN attempts to select a trunk in a busy state, a switch log will be generated and the failure will be indicated to the Service Node.

NOTE: An offhook of longer than 40 millisecond duration from the SN while a voice link is involved in a call results in the call being taken down as well as the voice link being put into the RMB state.

Broadcast voice link operation

In some cases it may only be necessary for the SN to play a recording that is not unique to that particular call, such as a branding announcement or music, without collecting any information from any party. While this type of call could be handled using a standard two-way, non-broadcast voice link, the fact that the direction of the information is one-way only (transmitted from the voice link to the call participants) and is somewhat generic lends itself to being more of a broadcasted type of transmission. In such a situation, voice link broadcasting may be used to connect not just an individual call, but several calls simultaneously to the same voice link. The switch network connection is a one-way broadcast transmission from the SN voice link to multiple calls. This results in a significant savings in voice link facilities between the switch and the Service Node.

When a Service Node requests a connection to a broadcast voice link, the DMS maps the given voice channel (from the Voice Channel DB) and node ID (from the Broadcast DB) to a physical trunk group and member number using table OAVLMAP. If the voice channel's BCST field is set to Y and connection to the trunk does not exceed the maximum number of simultaneous connections allowed for the voice link (field MAXCONS), a one-way broadcast network connection is made.

If a non-zero number is received in the Broadcast Cycle Count field of the Broadcast DB, the switch will keep track of the number of cycles played and release the voice link once that count is reached. The SN is informed of the release by a Voice Release Inform message.

If zero is received for the count, the switch will not count cycles and it is up to the Service Node to release the voice link with a Voice Release Request message.

While the DMS Network connection is made when the voice link is selected, the enabling of the transmission of Pulse-Code Modulated (PCM) speech samples from the broadcasting voice link occurs in one of two possible modes:

- immediate PCM cut-through when the connection is made. This is similar to the way a standard voice link operates.
- PCM cut-through disabled until the DMS detects an offhook of a least 10 milliseconds on the voice link.

The mode used is determined by datafill of field CUTTHRU in table OAVLMAP. The rationale for providing two modes of connecting is to allow for different types of broadcasting announcements. In some cases, such as for playback of music, it is unimportant as to when the call participants actually begin hearing the recording. For announcements of this type, immediate voice cut-through may be specified by setting field CUTTHRU to IMMED.

On the other hand, for a branding announcement it is usually important that the recording not be heard unless it is at the beginning of the recording. For such recordings, the voice transmission path should not be cut thru until a change of state on the circuit is received, indicating the beginning of the recording. This functionality is obtained by setting the CUTTHRU field to HKCHG.

For broadcast voice links the period of time that the PCM stream, as well as the voice link itself, remains connected is determined at the time of the request to connect a voice link. The SN can specify one of the following:

- the DMS is to disable the PCM stream and remove the voice link from the call after a certain number of cycles have been played, without waiting for instruction from the SN. In this case, the DMS will inform the SN that the specified number of cycles has been played and that the voice link has been removed from the call.
- An explicit request from the SN is required to remove the voice link from the call. The DMS will not monitor the number of cycles played, nor will it disconnect PCM from the voice link until the voice link is removed by a request from the SN.

Note: A cycle is considered to be 1 complete playing of a repetitive recording.

This latter method of releasing voice links is the way it works for **all** *standard* voice links.

Signalling on broadcast voice links

Signalling is used on broadcast voice links to convey not only an ‘Available/ Not Available’ status but also to signal the beginning of an announcement playback cycle.

Like the standard voice link, an idle voice link (one not used by any calls) is considered out of service if either the switch or the SN transmits an offhook; however, unlike standard voice links, an offhook received by the switch *during* a call will not result in the call being taken down, nor will the trunk go RMB at that time. Offhooks received from the SN during a call can be used to indicate the start of a new cycle, therefore they are either ignored (if the CUTTHRU type is IMMED) or are used to enable PCM (if CUTTHRU = HKCHG) and keep track of cycles (if CYCLE COUNT field in OAP message is other than 0). The offhook cannot be interpreted to mean ‘Not Available’ for broadcast voice links while at least one call is using the voice link. If the offhook still exists once the voice link is no longer in use by any calls it will at that time be put RMB.

Note: Refer to the *OSSAIN Open Automated Protocol Specification*, Q235-1 for a full description of OSSAIN voice links.

Application testing guidelines

This section gives information to help developers plan for testing their applications.

Platform considerations

The OSSAIN API is designed to be easily portable to a broad range of computing platforms. Hardware assumptions were minimized and the invocation of operating-specific routines were reduced. However, in some instances, platform-dependent procedure calls are required (for example, using sockets in the Unix environment).

The OSSAIN API was originally developed and tested on a platform with the following specifications:

- Hewlett Packard (HP) 712 workstation with 32-bit processor
- HP-UX A.09.05 operating system
- HP C++ A.03.72 compiler

The OSSAIN API has been upgraded to run on a platform with the following specifications.

- Hewlett Packard (HP) B1000 workstation.
- HP-UX 10.20 operating system
- HP C++ A.03.72 compiler

The OSSAIN API has been ported to a platform with the following specifications:

- PC with Intel Pentium processor
- Microsoft Windows NT 4.0 operating system
- Microsoft Visual C++ 6.0 compiler

Due to intrinsic differences in platforms, developers may need to modify and recompile the OSSAIN API software when using other hardware, operating systems, or compilers.

Nortel Vendor Validation Lab

To ensure software robustness, Nortel recommends that all applications built using the OSSAIN API software be verified in the Nortel Vendor Validation Lab (VVL) before deploying in an operating company. The VVL performs software and hardware validation for third-party vendors who provide services that work with DMS switches and functionality. Users should contact Nortel Directory and Operator Services Marketing to schedule time in the VVL.

Chapter 3: OSSAIN API functionality

This chapter describes the components of the OSSAIN API and how they work together. It discusses the three API software layers (protocol interface, node infrastructure, and base) and shows the flow of control in application processing.

Protocol interface layer

The protocol interface (PI) layer provides the high-level interface to the Open Automated Protocol (OAP). The PI layer stores the protocol specification, which includes OAP syntax and rules. The API uses the PI layer to examine the content of incoming OAP operations and populate the fields of outgoing OAP operations.

Note: For complete information on OAP, please refer to *OSSAIN Open Automated Protocol Specification*, Q235-1.

Operation class

The PI layer contains the operation class, which consists of a class header, an operation header, and zero or more data blocks. Figure 12 shows the OAP operation object.

Figure 12 OAP operation object

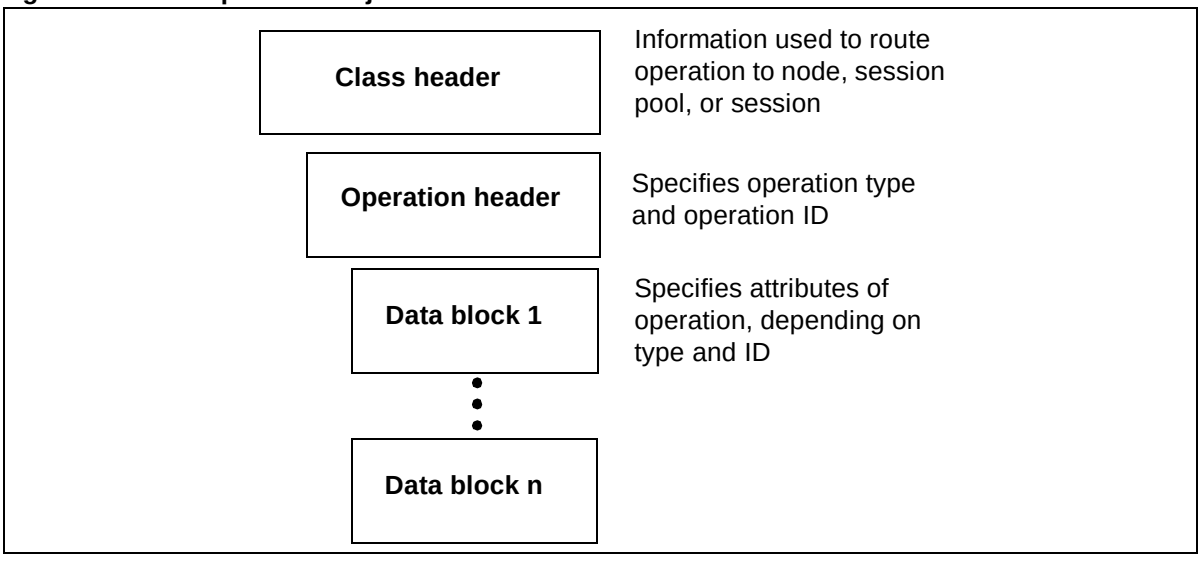
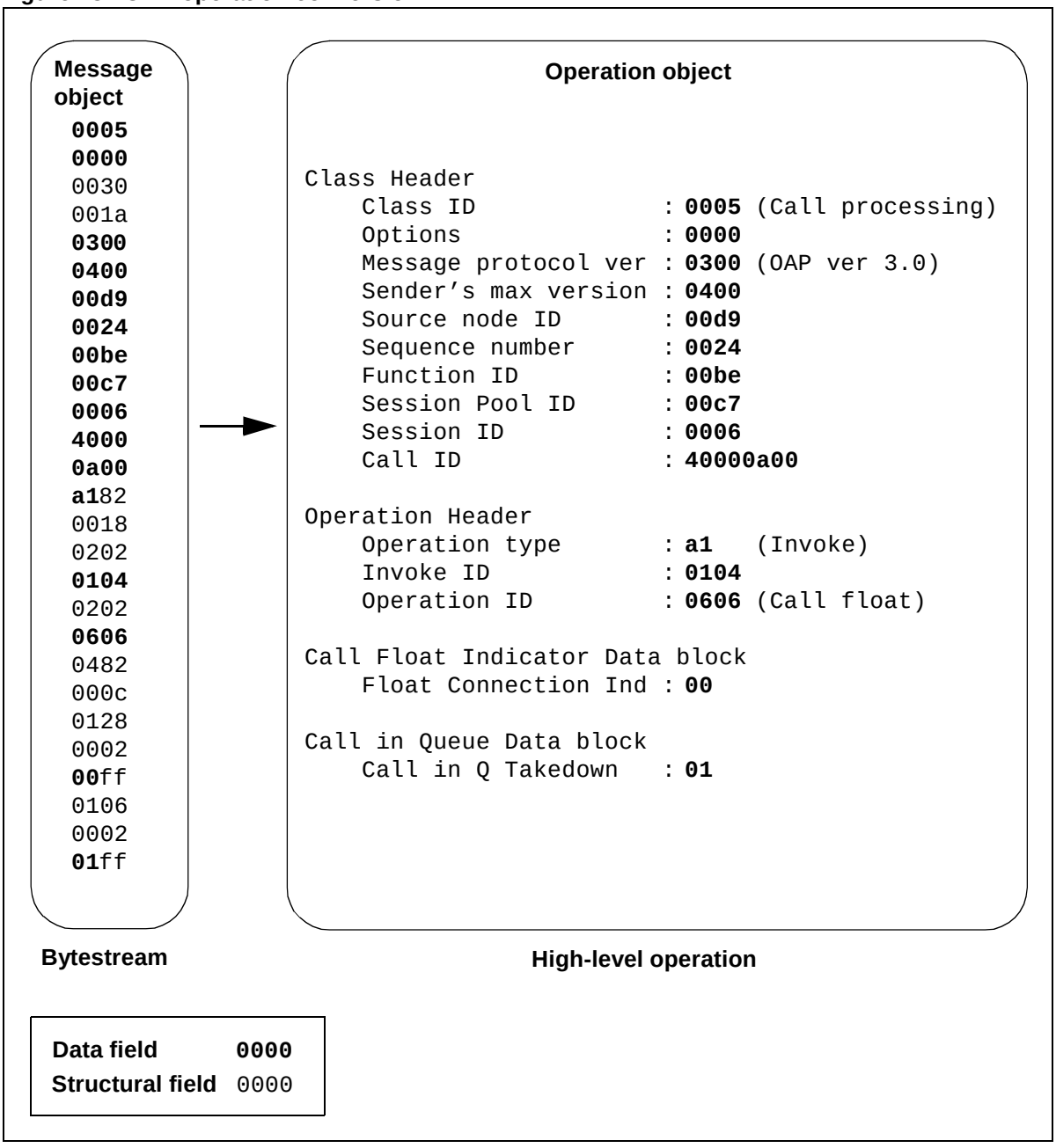


Figure 13 shows an example of a high-level representation of an OAP operation that the PI layer converted from the incoming bytestream sent by the switch. Data fields that are present in the high-level operation are shown in bold. The other fields are structural fields used in parsing the message, such as the length, offset, and byte count.

Figure 13 OAP operation conversion



Refer to Chapter 4: "OAP interface," for details on the PI layer.

Node infrastructure layer

The OAP API's node infrastructure classes model the important entities in the OSSAIN network. The NI includes node, session pool and session classes that parallel the entities provisioned on the DMS TOPS switch. A node contains one or more session pools, and a pool contains one or more sessions.

Beyond these classes and their obvious relationships, the API includes a mechanism to receive, process and send events handled by or produced by a node, session pool or session. All these objects accept and process instances of an abstract event class, OA_AbstractEvent.

Each event source can contribute its own variations on the abstract event class to the stream of events being processed. The variation is handled within the node, session pool or session by a context object. The context is a framework containing protocol-specific components that take care of processing the events in the context of the protocol to which the event subclass belongs.

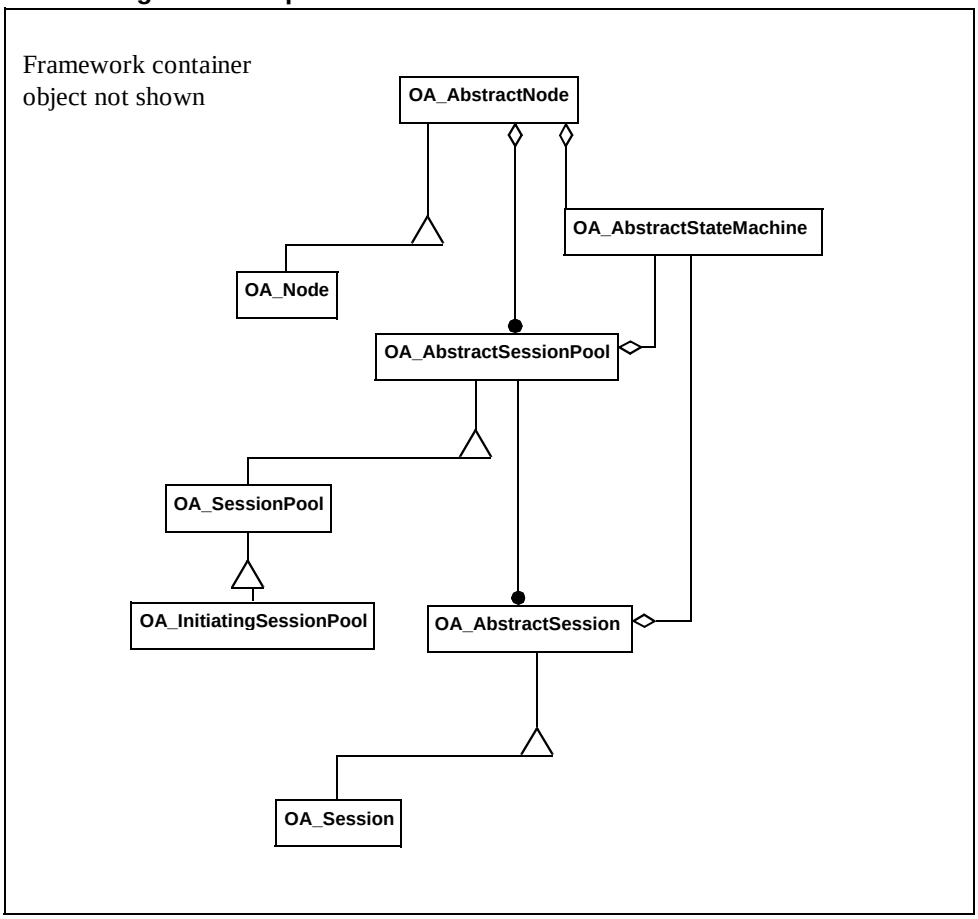
The NI objects described in this section therefore include:

- The node, pool and session hierarchy;
- The abstract event and event subclasses; and,
- The context object and its protocol-specific components.

Node, Pool and Session Objects

The object model in Figure 14 "Implementation of Node Infrastructure" shows the OAP API architecture for the node, session pool and session. The objects shown are implementation classes. All these objects are instantiated within a framework container object, which includes the entry point for the start of event processing.

Figure 14 Implementation of Node Infrastructure



A node, pool or session contains a state machine, which in turn contains a pointer to the object's current state. The state machine for the node and session pool are tailored to provide processing for node and session pool maintenance events. The state machine for a session is application-dependent.

The object hierarchy can be seen as a pyramid, with the node at the top, and session pools and sessions fanning out toward the bottom. When an event is received, the node looks at it first. If it is not intended for the node, the event is passed to the next layer down, the session pool named in the event. If the event's ultimate destination is not the session pool, then it must be for a session contained within the pool, and the pool routes the event downward.

Once at the destination, the event is handled by the state machine contained within the receiver.

Single and Multi-threaded Frameworks

Applications built with the OAP API can be single threaded or multi-threaded. In the single threaded model, all event sources are polled by the framework and then processed to completion in turn.

Note: Multi-threaded frameworks are only supported in the Windows NT environment.

In the multi-threaded model, the framework containing the object hierarchy spawns a number of threads, each of which corresponds to an event source. When an event is received in a thread, the thread locks the entire object hierarchy and then processes its event through the hierarchy by calling a process method on the node object.

The multi-threaded model used here is "coarse-grained." We lock the entire node infrastructure, even though processing of an event takes place primarily on a single node, pool or session object and its components. The rationale for this design is that it is simpler and therefore less prone to error than attempting concurrent access to the NI. In addition, the impact on performance of finer-grained locking is hard to judge, since finer-grained locking introduces more places to wait on mutex objects.

Event Flows Supported

Event processing in the API framework is divided into a sequence of preprocessing, processing and postprocessing of the input event. These steps take place in the state machine contained by the node, session pool or session handling the event. (See OA_StateMachine::process method for the implementation of the event handling algorithms.)

Each event processing flow can be run in any thread. Generally, a thread would lock the framework, execute the event processing logic and then release the framework lock so another thread can seize it. A maintenance task or event is an internal event that basically allows one API component to instruct another to take an action. Handling of maintenance tasks is nested within in the handling of external events. The supported algorithms for event processing are listed in bullets below.

Event flows:

- A single external input event resulting in zero, one or more output events. (Preprocess, process, postprocess.)
- A single external input event resulting in zero, one or more output events, with a single maintenance event also produced. The maintenance event will take precedence and will be processed in the postprocessing step before the output events are sent externally. (Preprocess, process, postprocess with maintenance processing first.)
- A positive assertion event will be supported. In this case, the application can interpret an incoming event as a positive assertion event that causes
  - — A change of state; and,
  - — Reprocessing of the incoming event in this new state.

  In this instance, the reprocessing will take place within the processing step. The event will thus be preprocessed, processed twice, and then the

postprocessing will occur. (Preprocess, process, process again, postprocess.)

Related to the event processing algorithms in the state machine is the behavior of other objects with respect to events. Some important points include:

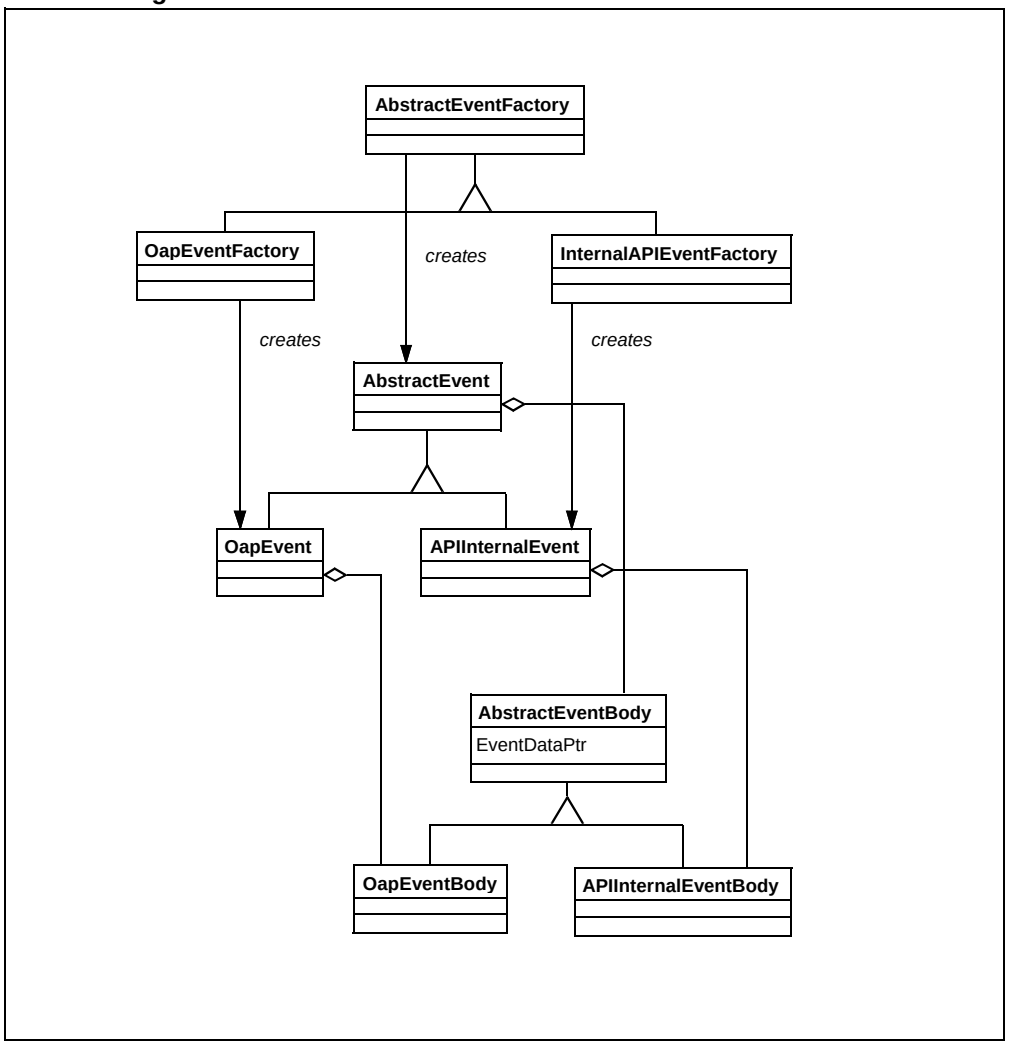
- In the multi-threaded model, the context object can hold a single output event in thread-specific store across event processing cycles before sending the event. This mechanism allows a thread that may be communicating asynchronously with another process to build an output event in steps as it receives information from the other process.
- Protocol-specific processing (namely, for OAP) in the context object will place an outgoing request event on a timer list. If a response is not received before timer expiration, the outgoing event will be marked as timed out and will be returned to the sending object. (Preprocess, process and postprocess of time-out like any other event.)
- A protocol-specific subclass contained in the context object can validate contents of incoming events, and can groom the incoming events to fill in event fields that are protocol-specific. This validation and any response to invalid events occurs during incoming event preprocessing. (Preprocess, send outgoing event if incoming event is invalid, OR, preprocess, process and postprocess if event valid.)

Event Classes

Events are expected to be generated by both OAP and non-OAP sources. An event hierarchy must reflect this variety. The structure of the event classes includes an abstract event and protocol-specific event subclasses. In addition, each protocol defines an instance of an event factory, which produces abstract events of the correct subclass for the protocol.

The object diagram below shows the OAP and Internal event subclasses.

Figure 15 Event Class Hierarch



In order to simplify memory management for events, which are continually created and destroyed, events are reference counted. The pattern described here is the handle-body pattern.

An abstract event is a data handle. It is bound to a body, which is the actual event object. The handle passes through all accessor methods to the actual body object that it is bound to. The body can be shared by multiple data handles and is reference counted to allow correct cleanup of events that are shared by multiple threads.

The body can contain a reference to event data. The event body carries an inuse flag for this reference that is protected by the same mutex used to protect the body's reference count. Event data can be set into a body only once and only before the reference count is incremented beyond 1.

Any handle that is deleted invalidates its contained event body, even though it will not delete the event body if it is held by another thread. Thus the first delete causes the event to be invalidated. It is the user's responsibility to check for event validity.

This implementation of the handle-body pattern is customized in that:

- In general, handles don't always have to be bound to bodies.
- The deletion of a handle doesn't have to mean that the body is invalid.
- The use of the reference lock to protect the inuse boolean is a customization.

Call Context Classes

The Node, Session Pool and Session objects each contain a state machine and a context.

In practice the context object contains almost all of the state information for event processing. The state machine contains a number of states (including any user-defined states) that express the behavior of the service node. All data that persists across events is contained in the context.

The context is responsible for tracking:

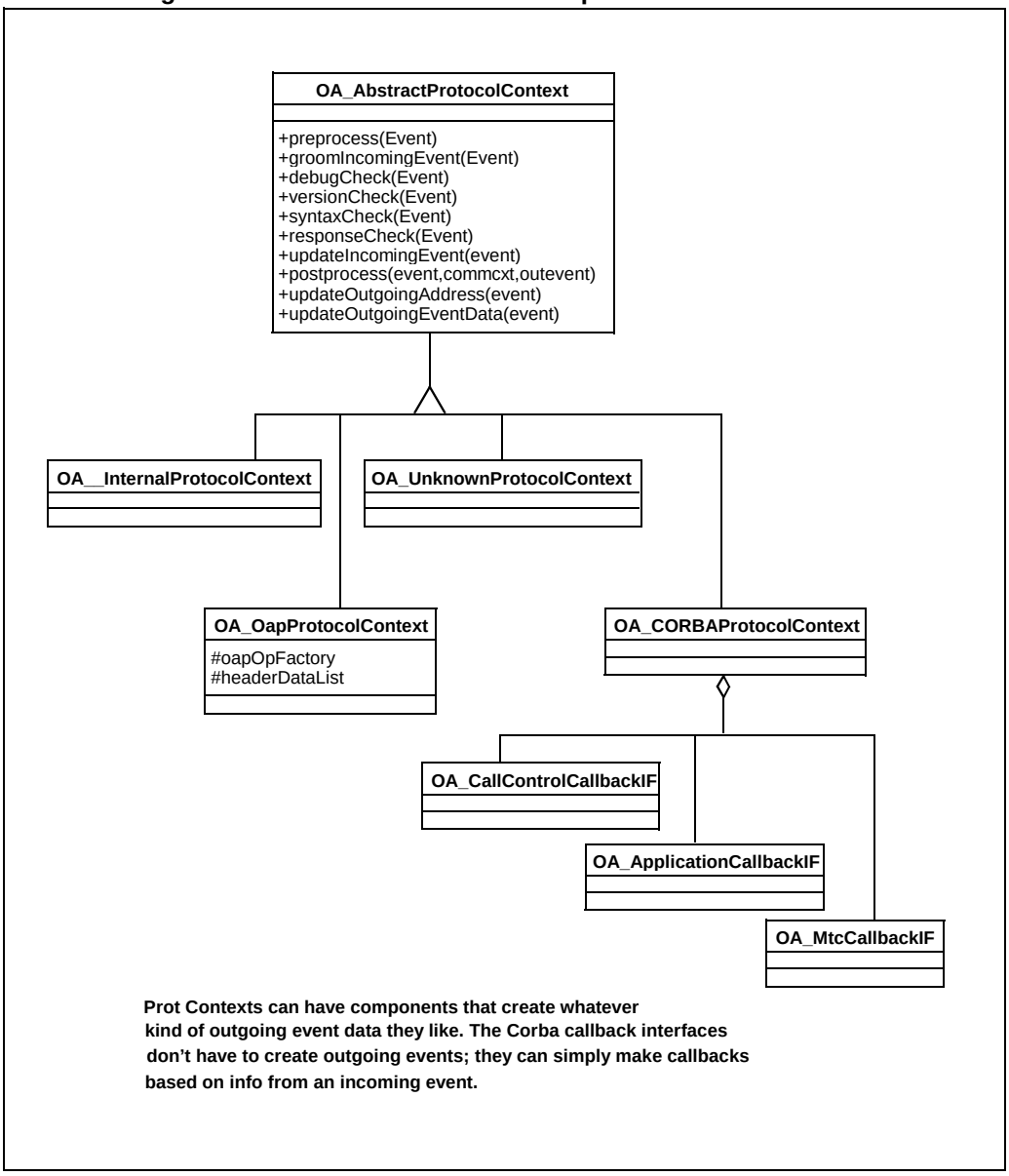
- OAP Conversation state for each external peer that the Node, Session Pool or Session is talking to;
- Addresses of all possible external peers;
- Outgoing events that have been sent and expect a response within a time-out period;
- Current in-service status and current OAP version for external peers;
- Any user-defined data that an application needs to keep across events; and,
- The current incoming event being processed and any outgoing event being prepared.

Due to its many responsibilities the context is divided into discreet objects with well-defined missions. The context class is a framework for objects that have a pre-defined responsibility. These components are varied according to event protocol, so that a context instance actually contains a set of components for each mission, one per protocol.

When a state machine processes an event, the context belonging to the node, pool or session is passed in to the state machine. The event is processed using the context object to perform protocol-specific steps, and using the context object to store call state across event-processing cycles.

The object model below shows how one major component of the context class, the protocol context, is subclassed to provide protocol-specific behavior.

Figure 16 The Context and Its Components



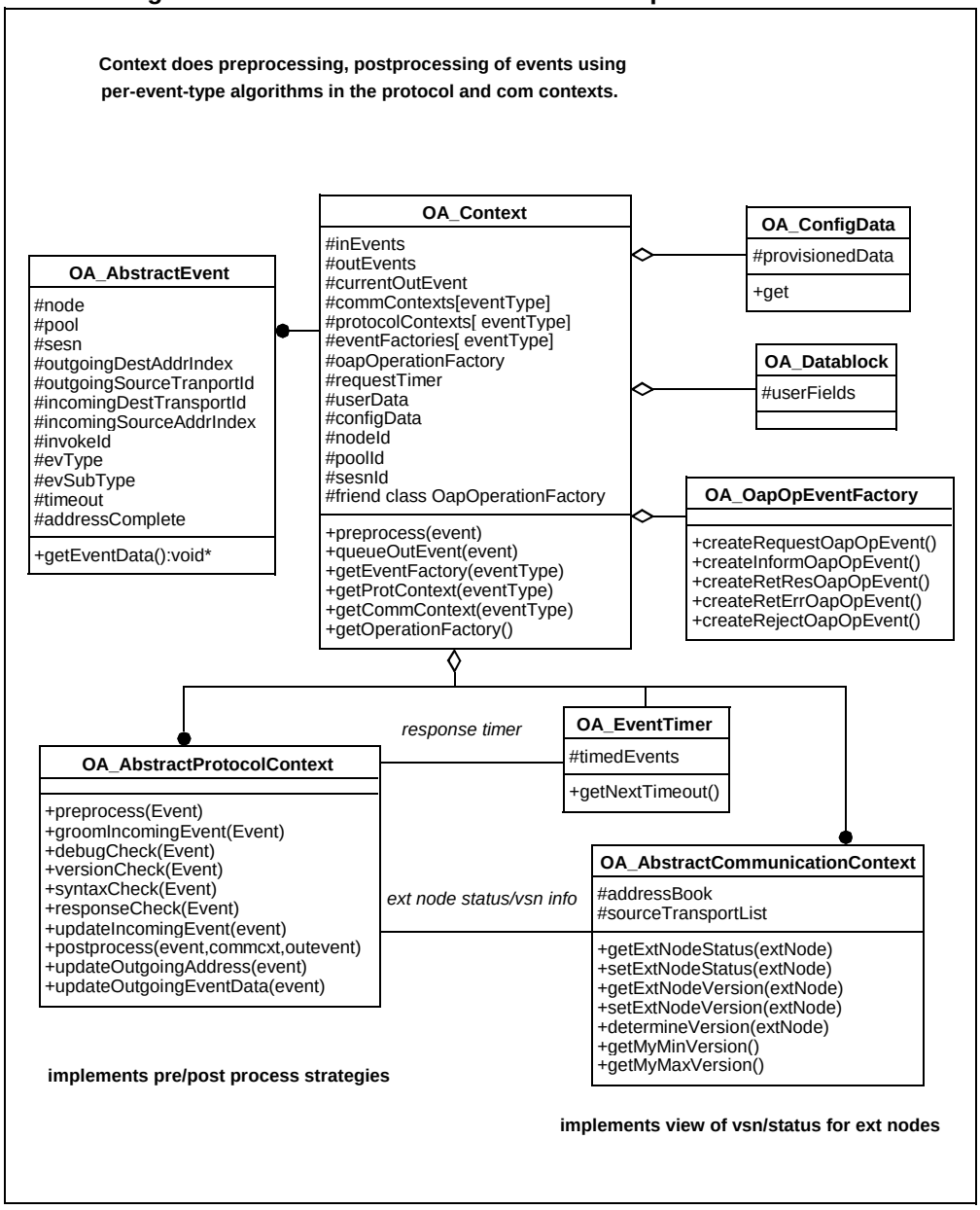
A subclass inherits and can override pre- and post-processing portions of event processing in the protocol component. For example, the OAP protocol context overrides updateOutgoingEventData to place a sequence number and invoke Id in the outgoing message.

The internal protocol context is mostly default routines used to handle internal maintenance tasks; the unknown protocol context is used to catch event types that are not implemented.

The Context Class and Its Subcomponents

The object model below shows the context class and its components. It illustrates how the context is divided into components that each have a distinct responsibility.

Figure 17 The Context Class and Subcomponents



The responsibilities of each component are listed in the following table.

Class	Description
OA_Context	The context provides a framework for event processing and storage of state information.
OA_AbstractProtocol Context	The protocol context implements strategies for pre- and post-processing of events. Subclasses of the protocol context can handle events in ways specific to the protocol.

Class	Description
OA_AbstractCommunicationContext	The communication context implements the protocol's view of its external peers. In general, a peer is said to have a version and a status (eg, in service or busy). In practice, these concepts are used only by OAP, but in theory another protocol could store its view in the communication context.
OA_EventTimer	The event timer provides a facility for timing abstract events. An event processing thread can place an event handle (reference) on the timer queue, and a timer thread will then remove the event and process it if its time limit expires.
OA_ConfigData	This object contains configuration data that is read in upon startup. It can be accessed then by any portion of the code that has access to the context.
OA_Datablock	The OA_Datablock class comes from the OAP Protocol, but is reused here to simply provide a flexible container for a user application to store information across event processing cycles.
OA_OapOpEventFactory	The OA_OapOpEventFactory violates the idea that the context is non-protocol-specific. Since much of the processing in the NI involves OAP messaging, however, it is convenient to allow the context to carry this factory as a means for code to create OAP events that contain OAP operations.
OA_AbstractEvent	The abstract event class is shown here associated with the context because the context can hold a list of all previous incoming events for a particular protocol. This event list allows code to retrieve previous operations to provide clients with data at any time from a previous incoming operation.

Base layer

The base layer includes low-level utilities for communication and datafill configuration, and building blocks required by the other layers. This chapter gives an overview of the base layer. Refer to Chapter 8: " Additional API components" for details.

Communication classes

The communications classes include a UDP and TCP transport class, an address book class (to store IP addresses of external nodes), and an IP address class (to store and format IP addresses).

Configuration utility

The configuration utility reads configuration parameters from a file. The parameters are used to create objects on the SN to match datafill on the switch. This utility also initializes logs.

Note: Chapter 9: "Configuration and administration" provides details on the configuration utility.

Other components of the base layer

The base layer contains the following other components:

- a log utility, which writes log reports to a file or display. The user can separate logs into alarm, trace, debug, and software error (swerr) types.
- an OM utility, which records operational measurements that are specific to the SN
- data structures, which include binary trees and linked lists to store data
- message classes, which store bytestreams
- a parser utility, which reads and tokenizes an input string or file according to a user-defined specification

Chapter 4: OAP interface

The Open Automated Protocol (OAP) is the interface between the DMS switch and an SN. OAP allows an adjunct processor to control the switch. This chapter describes the OSSAIN API protocol interface (PI), which is a collection of classes that allows an SN developer to use OAP without having to implement protocol details in code. This chapter discusses how developers can use the PI layer to examine the content of incoming OAP operations and populate the fields of outgoing OAP operations.

Overview of the PI layer

The PI classes offer the following capabilities to the SN application developer:

- They allow the developer's code to set the minimum number of fields in an OAP operation in order to build a new operation. The PI classes handle encoding of the operation into bytestream format.
- They handle decoding of an incoming bytestream into a high-level operation object. The developer's code can query the operation for the value of any field in the operation without knowledge of encoding formats, byte offsets, and other structural information.
- They include code to perform syntax validation on incoming or outgoing OAP operations. Developers can set the type of validation. Logs and return codes provide feedback to users in case of syntax violations.

PI classes

The PI layer consists of the following components:

- operations, data blocks, and fields
- identifiers
- messages
- protocol specification

This section discusses each component.

Operations, data blocks, and fields

The term *operation* refers to the high-level view of an OAP request or response. The PI classes allow the developer to set and query operation fields without having to worry about protocol details. *Data blocks* are components of operations, and *fields* are components of data blocks.

Operation, data block, and field classes hold values the developer can store and send across the network during OAP transactions. These components provide the formatting between the high-level representation of data used in the application code and the low-level representation sent on the network.

Identifiers

Developers using the PI software should include pre-defined enumerations and constant declaration lists in their programs. These enumeration members and constants label all the operations, data blocks, fields, error IDs, and reject IDs that the PI classes recognize.

A parent class, `OA_Identifier`, is declared in the PI header files. The following subclasses support manipulation of identifiers for operations, data blocks, and fields during SN transactions:

- `OA_OPID` for operations
- `OA_DBID` for data blocks
- `OA_FID` for fields
- `OA_ERRID` for error ID values in header field `OA_OH_ERROR_ID_VALUE`
- `OA_REJECTID` for reject ID values in header field `OA_OH_REJECT_REASON`

Applications that use operations, data blocks, fields, error IDs, or reject IDs should use these identifier types to represent values stored in the protocol specification.

The enumeration members and constants that can be used as instances of these types begin with the identifier type followed by the identifier-specific tag. For example, the identifier `OA_ERRID_CANNOT_ALLOCATE_CALL_ID` represents the error ID 0x700, which is discussed in the *OSSAIN Open Automated Protocol Specification* document as "Cannot allocate call id."

The *OSSAIN Open Automated Protocol Specification,* document (Q235-1) has complete information on the semantics of OAP. It explains the meanings of all operations, data blocks, and fields. In the specification document, operations, data blocks, and fields are represented with text labels.

Table 2 shows examples of PI identifiers for an operation, data block, and field. The PI identifiers attempt to match the documented label as closely as possible.

Table 2 Examples of text labels

OAP text label in OAP specification	PI identifier in OSSAIN API
Append AMA Module Request	`OA_OPID_APPEND_AMA_MODULE_REQUEST`
AMA Module DB	`OA_DBID_AMA_MODULE`
AMA Data	`OA_FID_AMA_DATA`

PI identifier enumerations are found in the following PI header files.

- oapopid.h - OAP operation identifiers
- oapdbid.h - OAP datablock identifiers
- oapfid.h - OAP field identifiers
- oaperrid.h - OAP error identifiers
- oaprejid.h - OAP reject identifiers

Exceptions to the PI identifier format are as follows:

- The `OA_DBID` for class header is `OA_CLASS_HEADER`; for operation header, it is `OA_OP_HEADER`.
- The `OA_FID` prefix for class header fields becomes `OA_CH`; for operation header fields, it becomes `OA_OH` (e.g. OA_CH_FUNCTIONID and OA_OH_OPERATION_ID_VALUE).

Messages

Message refers to a bytestream that is the encoded form of an OAP operation. The PI layer represents a message with the `OA_Message` class. The PI layer `OA_Operation` class takes a high-level operation and encodes it into message format, or decodes a message into high-level operation format.

Protocol specification

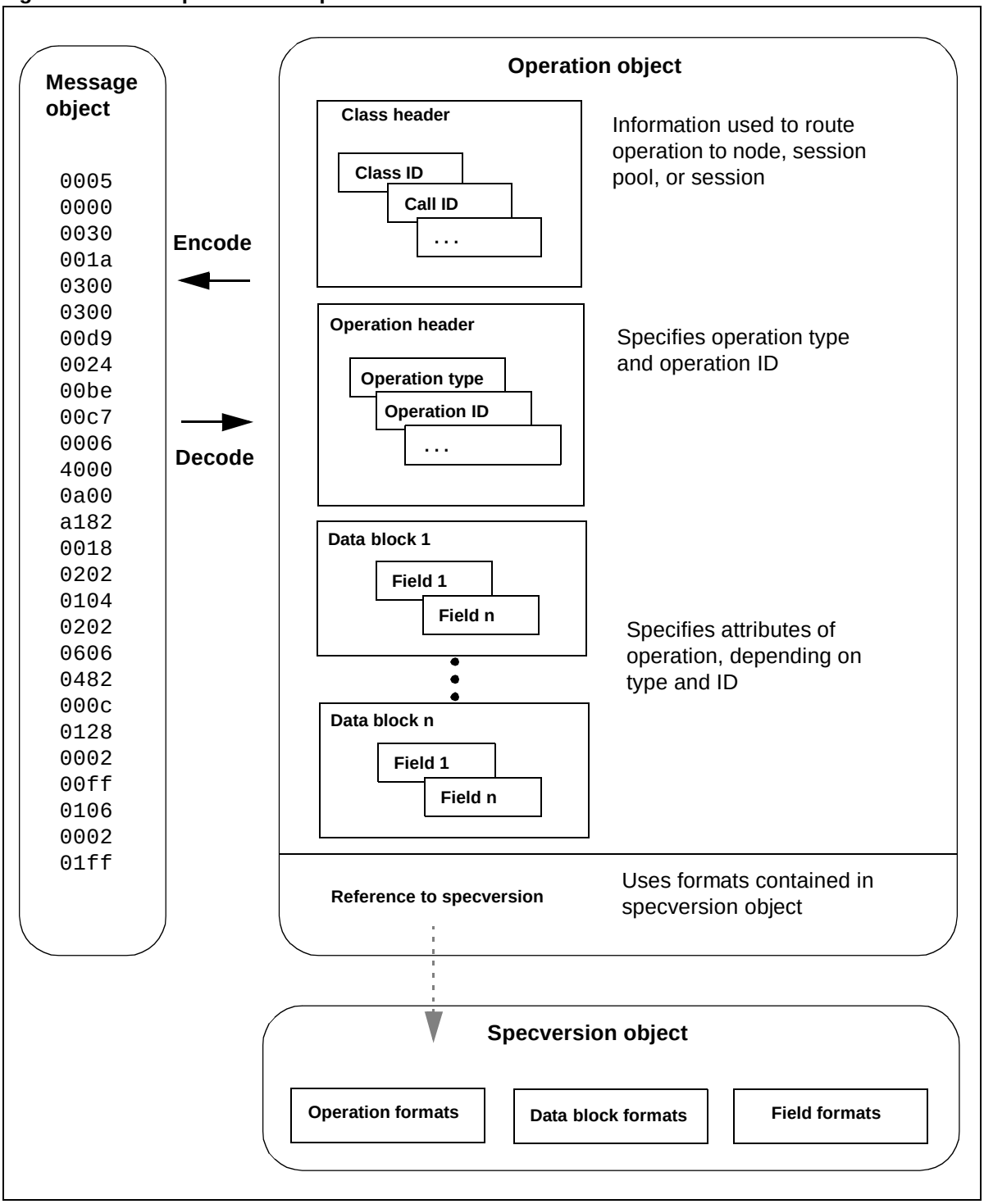
The PI layer contains classes that store specifications for the OAP protocol. The developer does not need to directly access the specification classes. The `OA_Operation` class maintains a reference to a specification version (specversion) object, which contains specification information for one version of the protocol.

The specification objects allow the developer to use operations, data blocks, and fields without knowledge of protocol details. For example, given a received bytestream encapsulated in an `OA_Operation` instance, the application can use `OA_Operation::getField()` to obtain a high-level representation of the field, without knowing details such as the field's offset within the bytestream or how it was encoded for sending on the network.

Relationships of PI components

Figure 18 shows components of an OAP operation. In the PI operation class, the class and operation headers are considered data blocks that contain fields.

Figure 18 OAP operation components



Decoding and encoding operations

The operation object provides the interface for converting OAP message byte streams into high-level operations and for converting high-level operations into byte streams.

Once an OAP message byte stream is retrieved, the buildOperation method of the operation object can be invoked to construct a high-level representation of the OAP message. A pointer reference to the message buffer and buffer size are provided as arguments to the buildOperation method.

High-level operations can be encoded into message byte streams using the getMessage method provided by the operation object. This method returns a pointer reference to the byte stream buffer representing the high level operation and the size of the buffer.

Figure 19 illustrates the use of the buildMessage and getMessage methods found on the operation object.

Figure 19 Decoding an encoding OAP message byte streams

```
OA_Operation* oper;
const OA_UBYTE *buffer              // buffer and bufferSize initialized
OA_UWORD bufferSize                 // elsewhere

oper = new OA_Operation( buffer, bufferSize );

oper->getMessage( &buffer, &bufferSize )
```

Using operations, data blocks, and fields

The operation object provides the interface for setting and getting fields from operations. The developer stores and retrieves data using numeric and char* types. The type used in actual storage of a field depends on the definition of the field, not on the format of the data the developer passes in.

For example, the developer could store a numeric field using a string value of "345." When the operation instance encodes a bytestream with this field value, the specification causes it to be formatted as an integer of the correct length, rather than as a null-terminated ASCII string.

Note: In such cases, the string data is always interpreted as ASCII chars representing hex digits.

To populate an outgoing operation with a field value, the developer must identify the data block and field within the data block where the value is to be stored. The API gives each data block a unique identifier. Field identifiers are unique within a particular data block. Each instance of a field value in a data block can be set only once; a second set overwrites the first value.

For convenience, if a field is set in a data block that does not already exist in the operation, that data block is transparently created in the operation and populated with the field value.

All fields have a default value that is encoded in the field when a message is produced, if the user does not set a value into the field explicitly. In many cases the default values are adequate, which means that the user only has to set values that are necessary in a given situation.

Tag fields

A unique OA_DBID exists for each potential instance of a data block in an operation. In most cases, when a data block is allowed to appear multiple times in an operation, the API uses a *tag field* value to create a set of unique data block identifiers. The tag field suffix distinguishes related data blocks. Table 3 shows examples of unique data block identifiers for the OA_DBID_LANGUAGE data block.

Table 3 Data block identifiers for OA_DBID_LANGUAGE data block

Data block	Tag value in party field	Unique data block identifier
OA_DBID_LANGUAGE	A_PARTY	OA_DBID_LANGUAGE_A_PARTY
OA_DBID_LANGUAGE	B_PARTY	OA_DBID_LANGUAGE_B_PARTY
OA_DBID_LANGUAGE	ALT_PARTY	OA_DBID_LANGUAGE_ALT_PARTY

Up to three instances of the language data block are allowed in an OAP Call Float Request operation. In the example, the language data block has three variants that are assigned the names OA_DBID_LANGUAGE, with a suffix of A_PARTY, B_PARTY, or ALT_PARTY. The developer should use the appropriate data block identifier for the party whose language is specified. The developer does not need to set the value of the distinguishing tag field, which in this example is the party field.

Multi-data blocks for MIS applications

For MIS applications, the OAP uses three data blocks to stream MIS event data from the switch to an SN. Because the same MIS event can occur multiple times, the data blocks in the stream may be identical except for the time at which the event occurred.

These *multi-data blocks* cannot be distinguished from each other using a tag field. The protocol specification of the API supports multi-data block types, and the PI layer classes support multiple instances of these data blocks in the same operation.

The following multi-data block IDs are used in the PI classes:

- OA_DBID_CALL_QUEUE_EVENT
- OA_DBID_NON_CALL_SESSION_EVENT
- OA_DBID_SESSION_EVENT

For MIS applications, developers can use optional arguments to `getField()` and `setField()` that indicate by an index which instance of a given MIS data block to manipulate. The `OA_Operation` class includes these optional index arguments in the parameter lists. And methods such as `getFirstDbComponent()` and `getNextDbComponent()` allow sequential access to the lists of event data blocks as an ordered set of data blocks.

Setting fields in outgoing operations

The header fields and structural fields of outgoing operations are set properly by the API. Structural fields include data block ID, data block byte count, padding fields, length fields, and tag fields. The application must set non-structural fields in data blocks of outgoing operations.

The application should use the `setField()` method of the operation object to set fields in operations. Figure 20 shows an example of setting data block fields in an outgoing operation.

Figure 20 Setting fields in an outgoing operation

```
OA_Operation* oper;
OA_UWORD networkServiceId = 500;
char* directoryNumber = "9195558372";
const OA_UBYTE *buffer;
OA_UWORD bufferSize;

OA_Operation::buildOperation((OA_Operation**)&oper,
                        OA_OPID_SESSION_INITIATION_REQUEST,
                        oapMsgVersion, oapMaxVersion);

//  Charge Status DB
//
oper->setField( OA_IS_BILLABLE_STATUS,
                OA_FID_BILLABLE_STATUS,
                OA_DBID_CHARGE_STATUS );

//  Network Service DB
//
oper->setField( networkServiceId,
                OA_FID_NETWORK_SERVICE_ID,
                OA_DBID_NETWORK_SERVICE );

//  Directory Number Update DB, A-Party
//
oper->setField( OA_UNKNOWN_LANGUAGE,
                OA_FID_LANGUAGE,
                OA_DBID_DIRECTORY_NUMBER_UPDATE_A_PARTY );

oper->setField( OA_DOMESTIC_LOCAL_DIRECTORY_NUMBER_TYPE,
                OA_FID_DIRECTORY_NUMBER_TYPE,
                OA_DBID_DIRECTORY_NUMBER_UPDATE_A_PARTY );

oper->setField( directoryNumber,
                OA_FID_DIRECTORY_NUMBER,
                OA_DBID_DIRECTORY_NUMBER_UPDATE_A_PARTY );(see Note)

// Retrieve message byte stream from operation
oper->getMessage( &buffer, &bufferSize );
```

Note: The directory number in this example is not a numeric value, so the digits are encoded as an array of BCD digits.

Getting fields from incoming operations

After an operation is constructed from an incoming bytestream, all header and data block fields can be retrieved using the `getField()` method of the operation object. For convenience, if the bytestream has not yet been decoded to high-level data when the application requests a field value, the decoding is done transparently.

Figure 21 shows an example of getting fields from an operation response.

Figure 21 Getting fields from an incoming operation

```
OA_AbstractOperation* oper;
OA_UWORD functionId = 0;
OA_UWORD allowedBillingSet = 0;
OA_UWORD phoneClassType = 0;
char* hotelRoomNumber = "";
const OA_UBYTE *buffer             // buffer and bufferSize
OA_UWORD bufferSize;               // elsewhere from incoming
                                   // OAP msg byte stream representing
                                   // session init success response.

// Build high-level operation from buffer
oper = new OA_Operation( buffer, bufferSize );

oper->getField( &functionId,
                OA_CH_FUNCTIONID,
                OA_CLASS_HEADER );

oper->getField( &allowedBillingSet,
                OA_FID_ALLOWED_BILLING_SET,
                OA_DBID_BILLING_INFORMATION );

oper->getField( &phoneClassType,
                OA_FID_PHONE_CLASS_TYPE,
                OA_DBID_ORIGINATING_STATION_INFORMATION );

if (( phoneClassType == OA_HOTEL_MOTEL_PHONE_CLASS_TYPE ) ||
    ( phoneClassType == OA_HOTEL_MOTEL_NO_CHARGE_PHONE_CLASS_TYPE ))
{
   oper->getField( &hotelRoomNumber,
                   OA_FID_HOTEL_ROOM_NUMBER,
                   OA_DBID_ORIGINATING_STATION_INFORMATION );
}
```

Field value constants

Numeric fields in OAP sometimes expect a limited range of values, each of which has some pre-determined meaning. For example, the code sample in Figure 21 includes constants for hotel/motel phone class type and hotel/motel no charge phone class type.

Constants declared in a header file (`oapfidvl.h`) in the PI layer include labels for all meaningful numeric values in OAP fields. The developer can include this header file to avoid having to know that the value of 5, for example, should be interpreted as hotel/motel phone class type when it appears in field `OA_FID_PHONE_CLASS_TYPE`. All meanings are the same as those listed in the *OSSAIN Open Automated Protocol Specification* (Q235-1) document.

Interpreting OAP operations

During a typical OAP message exchange between a service node and OSSAIN switch, the SN sends a request to the DMS and the DMS sends a response back to the SN indicating the success or failure of the request. This section describes how an application developer can utilize the PI layer to obtain the necessary information to interpret the OAP messages received by the SN application.

Note: The NI layer provides several convenience methods for performing these activities. Refer to the chapter on building a basic application for additional information.

Once a high level operation is built from an incoming OAP message using the operation object, the application analyzes the operation to determine the proper action to be taken. The first step is to retrieve the Operation/Response type of the OAP operation. The Operation/Response type indicates whether the operation is an invoke (inform or request), a return success, a return error, or a return reject. The OAP API provides enumerated types that represent the valid Operation/Response type values.

- Invoke operation (0xA1) - OA_OPTYPE_INVOKE
- Return result operation (0xA2) - OA_OPTYPE_RET_RES
- Return error operation (0xA3) - OA_OPTYPE_RET_ERR
- Return reject operation (0xA4) - OA_OPTYPE_REJECT

Figure 22 illustrates how to obtain the Operation/Response type from the operation object.

Figure 22 Obtaining Operation/Response type from high-level operation

```
OA_UWORD optype = OA_UNKNOWN_OPTYPE;

const OA_UBYTE *buffer              // buffer and bufferSize initialized
OA_UWORD bufferSize                 // elsewhere

OA_Operation* inOp = new OA_Operation( buffer, bufferSize );

inOp->getField( &optype, OA_OH_OPERATION_RESPONSE_TYPE, OA_OP_HEADER );
```

The `getField` method of the operation object is used to obtain the Operation/Response type from the operation object. The datablock identifier specified in the call to `getField` is `OA_OP_HEADER`. This specifies that the requested field is to be obtained from the operation header of the operation object. The exact field to be obtained is indicated by the field identifier `OA_OPERATION_RESPONSE_TYPE`.

Once the Operation/Response type is known, further analysis is done to determine the action to be taken. For invoke operations (inform or request), the operation header contains an operation identifier that specifies the inform or request operation being received. The operation identifier is obtained by calling the `getField` method of the operation object and specifying the field identifier `OA_OH_OPERATION_ID_VALUE` and datablock identifier `OA_OP_HEADER`. Figure 23 illustrates how to obtain the operation identifier from the high-level operation.

Figure 23 Obtaining operation identifier from high-level operation

```
OA_UWORD opId;
inOp->getField( &opId, OA_OH_OPERATION_ID_VALUE, OA_OP_HEADER );
```

Return success, return error, and return reject operations do not contain an operation identifier. They do contain an invoke identifier that can be used to match to outstanding requests that have been previously made. When an application constructs an OAP request operation (invoke operation), the application can set the invoke identifier of the operation to a unique value. When a response to a request is received, the invoke identifier of the response can be obtained and used to match against outstanding requests in order to determine the operation identifier of the request. Once again, the `getField` method of the operation object is invoked specifying the field identifier `OA_OH_INVOKE_ID_VALUE` and datablock identifier `OA_OP_HEADER`.

Figure 24 Obtaining invoke identifier from high-level operation

```
OA_UWORD invokeId
inOp->getField( &invokeId, OA_OH_INVOKE_ID_VALUE, OA_OP_HEADER );
```

Once the application has established the Operation/Response type of the message and the associated operation identifier, additional information can be obtained as required by the service node application. For example, if the operation received is a session begin inform, the application may obtain information about the call such as the call origination type or service type.

As documented in the OAP Specification, some operation datablocks are mandatory and some are optional. The getField method on the operation object has a return type of `OA_RCODE`. If the getField returns the value `OA_RC_SUCC`, then the requested information was successfully obtained from the operation. If the return value is `OA_RC_NOTAVAILABLE`, then the requested information could not be obtained. In such a case, it is likely that the requested information is optional.

Figure 25 provides an example of how an incoming operation could be processed by a SN application. In this example, a call is already in progress. The SN application has just made a voice connect request to the DMS and is waiting for a response. You will be shown how the PI layer can be used to analyze the incoming response to make decisions about the processing to be performed for the call.

Figure 25 Processing an incoming OAP operation

```
// In this example we are waiting for a response to a voice connect
// request.  The invoke id was stored in requestInvokeId

OA_AbstractOperation* inOp;
OA_UWORD optype = OA_UNKNOWN_OPTYPE;

const OA_UBYTE *buffer             // buffer and bufferSize
OA_UWORD bufferSize;               // elsewhere from incoming
                                   // OAP msg byte stream

// Build high-level operation from buffer
inOp = new OA_Operation( buffer, bufferSize );

if ( inOp->getField( &optype, OA_OH_OPERATION_RESPONSE_TYPE,
                     OA_OP_HEADER ) != OA_RC_SUCC )
{
   // Error
   return;
}

switch ( opType )
{
   case( OA_OPTYPE_INVOKE ):
      {
         // Handle the unexpected inform or request message
         OA_UWORD opId;
         if ( inOp->getField( &opId, OA_OH_OPERATION_ID_VALUE,
                              OA_OP_HEADER ) != OA_RC_SUCC)
         {
            // Error: There should always be an opId for invokes
            return;
         }
         switch ( opId )
         {
             case(OA_OPID_SESSION_BEGIN_INFORM):
                // Unexpected new call (Positive assertion).
                // Existing call has terminated.
                // Reset and handle new call
             case(OA_OPID_CONNECTION_STATUS_INFORM):
                // Party has gone on hook.  Verify hook status
                // and process accordingly
             case(OA_OPID_CALL_END_INFORM):
                // Call has been terminated at the switch.
                // Reset and wait for new call
             default:
                handleOther( inOp ); // Catch all method
         }
      }
```

Figure 26 Processing an incoming OAP operation (continued)

```
    case( OA_OPTYPE_RET_RES ):
       // We received a return response operation.
       // Retrieve the saved operation id based on th invoke id
       {
          OA_UWORD invokeId;
          OA_UWORD poolId;
          OA_UWORD sessionId;
          OA_UWORD opId;
          if ( inOp->getField( &invokeId, OA_OH_INVOKE_ID_VALUE,
                               OA_OP_HEADER ) != OA_RC_SUCC )
          {
             // Error: we should always be able to obtain an invoke id
             return;
          }

          // Obtain information about the outstanding request based
          // on the invoke identifier.  How outstanding requests
          // are saved is application specific.  For this example,
          // we assume a method exists where given an invoke id, session
          // pool id, and session id, the method returns an operation
          // id.  If no matching request is found, an error is returned.
          if ( inOp->getField( &poolId, OA_CLASS_HEADER,
                               OA_CH_SESSPOOLID) != OA_RC_SUCC )
          {
             // Error: we should always be able to obtain a pool id
             return;
          }
          if ( inOp->getField( &sessionId, OA_CLASS_HEADER,
                               OA_CH_SESSIONID) != OA_RC_SUCC )
          {
             // Error: we should always be able to obtain a session id
             return;
          }

          if ( getOutstaningRequest( poolId, sessionId,
                                     invokeId, &opId ) != TRUE )
          {
             // No outstanding request for pool/session/invokeId
             return;
          }
```

Figure 27 Processing an incoming OAP operation (continued)

```
         if ( opId != OA_OPID_VOICE_CONNECT_REQUEST )
         {
            // Some sort of state mis-match error.
            return;
         }

         // Assume voice link request specified switch was to pick
         // the voice link.  Obtain the voice link info from the
         // return result message.
         OA_UWORD trunkGroupId;
         OA_UWORD trunkMemberId;

         if ( inOp->getField( &trunkGroupId, OA_DBID_SN_VOICE_LINK,
                              OA_FID_TRUNK_GROUP_ID) != OA_RC_SUCC )
         {
            // Error: could not obtain trunk group id
            return;
         }
         if ( inOp->getField( &trunkMemberId, OA_DBID_SN_VOICE_LINK,
                              OA_FID_TRUNK_MEMBER_ID) != OA_RC_SUCC )
         {
            // Error: could not obtain trunk member id
            return;
         }

         // Save trunk group & trunk member id and continue call
         // processing.
       }

   case( OA_OPTYPE_RET_ERR ):
   case( OA_OPTYPE_REJECT ):
      // Handle error and reject case
      break;
   default:
      // Error: Unknown Operation/Response type
      return;
}
```

Operation syntax validation

The `OA_Operation` class includes the methods `validate()` and `validateResponse()`. These methods, when called on an operation instance, perform syntactic validation on the data contained in the operation. The methods return an `OA_RCODE` value indicating success or failure of the validation.

In general, validation is useful during development and testing of a service node application. It can ensure that an application is always formulating syntactically correct operations to send to the switch.

Note: To save real time, it is expected that after an application is debugged, validation of outgoing messages would be turned off.

Syntax validation is controlled using a four-bit set of validation flags. Each bit is represented by a constant declared in the PI layer. There are at least two ways to regulate validation, as follows:

- The application can place a validation flag value in a configuration file read in by the SN at the start of processing. (A configuration utility is included in the OSSAIN API base layer.) This value can range from no validation to full validation. This value is used in the NI classes to determine whether to validate incoming operations. Outgoing operations are validated only if a compiler directive defines `OA_DEBUG`.
- The application can call `validate()` or `validateResponse()` on instances of `OA_Operation` that it creates or that is passed from methods on NI classes. In this case, the application can set its own validation flags. These could be conditionally compiled, for example, to turn validation on or off using `OA_DEBUG` or another flag defined by the developer.

Types of validation

A set of validation flags controls the extent of the validation performed. The meaning of the flag bits is represented by the following constants in PI layer code:

- `OA_VAL_CLASS_HEADER`

  Field range validation is performed on fields in the class header. Missing or extra fields are detected. (Bit zero is on.)

- `OA_VAL_OP_HEADER`

  Field range validation is done on fields in the operation header. Missing or extra fields are detected. (Bit one is on.)

- `OA_VAL_DATABLOCKS`

  All data blocks in the operation have their fields validated. Missing or extra fields are detected. (Bit two is on.)

- `OA_VAL_OP_STRUCTURE`

  The operation is checked for extra or missing data blocks, and for data blocks that are mutually exclusive under OAP syntax. `OA_VAL_DATABLOCKS` must also be set for this check to be done. (Bit three is on.)

- `OA_FULL_VALIDATION`

  All four validations are performed on the operation. (All bits are on.)

- `OA_NO_VALIDATION`

  No validation is performed. (All bits are off.)

`validate()` and `validateResponse()` methods

The `OA_Operation` class includes two different validation methods, `validate()` and `validateResponse()`. The `validate()` method validates Invoke (OAP Inform or Request) operations. The `validateResponse()` method validates responses to operations. The reason for two different methods is that Invoke operations carry an operation identifier, and responses do not. So, the arguments to `validateResponse()` include an operation ID.

However, there is some overlap to operations that can be validated using these methods. Table 4 shows the usage of these methods.

Invoke, Return Error, and Reject operations can all be validated *without* an operation ID, while Return Result operations require the use of `validateResponse()` with an operation ID argument.

Table 4 Usage of validation methods

Operation type	Applicable validation routines
Invoke, Return Error, Reject	`validate()` or `validateResponse(OA_UNKNOWN_OPERATION)`or `validateResponse(OA_OPID_XYZ()` where `OA_OPID_XYZ` identifies the Invoke operation that either is being validated or that caused the Return Error or Reject response.
Return Result	`validateResponse(OA_OPID_XYZ()` where `OA_OPID_XYZ` identifies the Invoke operation that caused the Return Result to be sent.

Validation errors

Successful validation returns an `OA_RCODE` of `OA_RC_SUCC`. Validation errors can cause a large number of return code values. Error cases generally cause a log report explaining the error. For a complete list of return code values, refer to *OSSAIN API Code Reference Guide*, Q260-2.

Chapter 5: Building a basic application

This chapter provides step-by-step instructions on how to build a basic SN application using the components of the node infrastructure (NI) layer. After determining the possible call flows (see page 30), developers should follow these five steps:

1 Design a state model that handles the call flows.

2 Implement the call state model using the API state classes.

3 Extend the API maintenance classes to perform application-specific maintenance behavior.

4 Create a subclass of the OSSAIN API framework.

5 Implement the SN application as a stand-alone application.

Each step is described and illustrated in this chapter.

Note: For information on advanced application topics such as integrating another application with the API, refer to Chapter 6: "Advanced application topics."

Step 1—Designing a state model

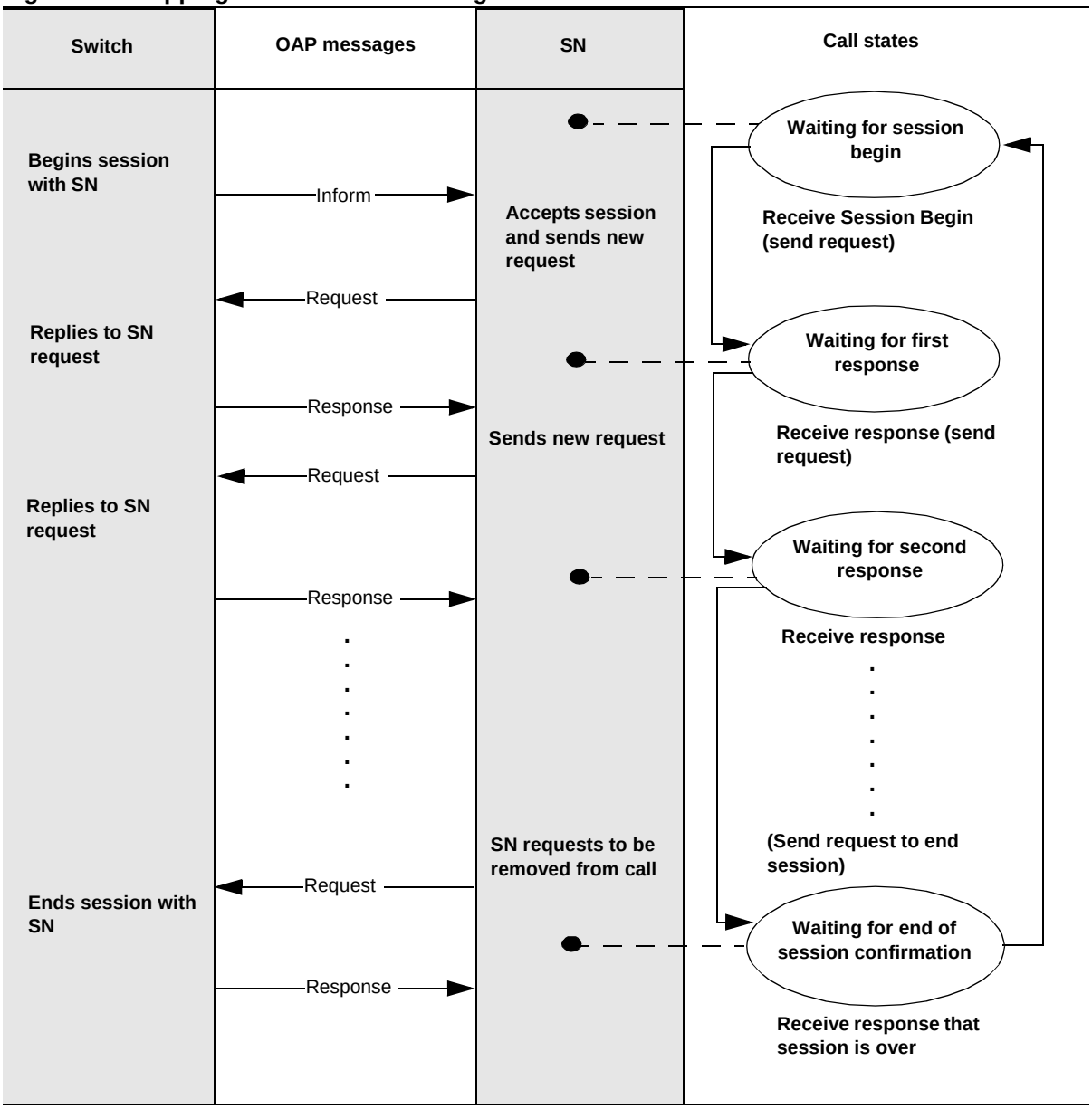
A state model defines a set of distinct points in time known as states. Each state recognizes a set of events. Based on the event, the state performs various tasks and transfers control to another state.

The state model for a call processing application typically contains an *initial state* that waits for the start of a new session. The incoming OAP message that starts the conversation is treated as an event. Based on the incoming message, the state performs application-specific tasks, creates any needed outgoing messages, and transfers control to the state that handles the next message in the call flow.

A state that represents a point during an active call typically sends a request and then transfers control to the state that expects its reply. States should recognize an event that represents the end of the conversation by transferring control back to the initial state.

Figure 28 shows the mapping between incoming OAP messages and call states for a subscriber-originated call.

Figure 28 Mapping between OAP messages and call states



Step 2—Implementing the call state model

To implement the state model, the API provides the OA_State class. Each state in the model is a subclass of OA_State.

The application developer must implement the following four methods of the OA_State interface:

- process() method, which makes the call processing decisions. It checks incoming operations, performs application-specific tasks, creates any needed outgoing operations, and transitions to the next state. The API calls the process() method to handle incoming messages.
- reset() method, which allows the application to clean up its call processing resources. The state calls this method when the conversation is over. This method also is called when the SN or a session pool is taken out of service while a call is active.
- getId() method, which returns a user-defined unique ID for the state. The states use these IDs to reference each other when transferring control to another state.
- getStatus() method, which returns whether the session is active with a call. This information is used by the API to choose an idle session when the SN initiates a new conversation. It also is used to determine if the SN or a session pool can be taken out of service without disrupting call traffic.

The four methods of the OA_State interface are described in this section, beginning on page 78.

Example header file

Figure 29 shows the header file containing the OA_State definitions used for the examples in this section.

Figure 29 Example header file with OA_State definitions

```
#include "oa/state.h"

class MyFirstState : public OA_State
{

  OA_UWORD getId();
  OA_STATUS getStatus();
  OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* ev, OA_AbstractContext* cxt );
  void reset( OA_INTERNAL_TAKS_TYPE mtcTask,
        OA_AbstractContext* cxt );

};

class MySecondState : public OA_State
{

  OA_UWORD getId();
  OA_STATUS getStatus();
  OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* ev, OA_AbstractContext* cxt );
  void reset( OA_INTERNAL_TAKS_TYPE mtcTask,
        OA_AbstractContext* cxt );

};

// ...

class MyLastState : public OA_State
{

  OA_UWORD getId();
  OA_STATUS getStatus();
  OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* ev, OA_AbstractContext* cxt );
  void reset( OA_INTERNAL_TAKS_TYPE mtcTask,
        OA_AbstractContext* cxt );

};
```

process() method

The process() method makes the call processing decisions. It examines a single incoming event, creates outgoing events, and determines which state should be next in the call flow.

This section discusses the following areas related to the process() method:

- context object
- incoming events
- outgoing events
- timeouts

- transitioning to the next state
- non-blocking code
- wakeup timer

Context object

The API provides a context object when it calls the process() method. The context maintains information related to the current conversation. Additionally, the context object provides access to communication contexts and protocol contexts. Methods on the context allow the application to queue out going events, set and clear timeout events, and manipulate application session capacity.

The context object can be subclassed by the application developer to provide for storing and retrieving application specific data.

Incoming events

The API provides two standard types of events, OAP events and internal events. OAP events represent OAP messages being received from the DMS. Internal events represent activities such as expiration of timers or maintenance actions (e.g. session pool going out of service). An application event type is provided so the application developers can create their own application events.

The first task to be performed by a state's process method is to determine the type of event being processed. This is done be retrieving the event type of the event. Once the event type is determined, the event subtype is retrieved to further refine the event being processed. In the case of OAP events, the event subtype is the OAP operation identifier of the incoming OAP message.

Figure 30 shows an example of examining an incoming event.

Figure 30 Example of examining an incoming operation

```
OA_INTERNAL_TASK_TYPE
myState::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_APPLICATION_EVENT_TYPE:
      //
      //  Handle application event
      //
      break;

    case OA_OAP_EVENT_TYPE:
      //
      // OAP Event
      //
      switch ( inEvent->getEventSubType( ) )
      {
        case ( OA_OPID_SESSION_BEGIN_INFORM ):
          //
          // Session begin inform.  Handle new call.
          //
          break;
        default:
          //
          // Handle unexpected OAP operation
          //
      }
      break;

    case OA_OAP_API_INTERNAL_EVENT_TYPE:
      //
      //  Handle internal event
      //
      break;

    default:
      //
      // Handle unexpected event type
      //
  }
  return OA_INT_TASK_NULL
}
```

Outgoing events

The application sends the switch only two of the five types of OAP operations—*requests* and *informs*. The application should not send success or error responses because the SN does not receive call processing requests from the switch. The application also should not send rejects, because the API automatically sends them whenever there is a protocol violation.

The context object provides an accessor method, getOapOperationFactory, for obtaining an OAP operation factory. The operation factory contains convenience methods for the creation of OAP request and inform operation events.

- createRequestOapOpEvent - creates a request event
- createInformOapOpEvent - creates an inform event

These methods take an OAP operation identifier as an argument to determine the operation being created and return an OAP event. The event factory populates header fields to appropriate values for the current state of the call. The application is responsible for providing field values for any needed data blocks.

To set field data, the application must first retrieve the operation from the newly created operation event. This is done by invoking the getEventData method on the event object. This returns an OAP operation object which is the high level representation of the OAP operation. The application can then utilize the setField method of the operation object to set field data.

The application can provide a timeout value when creating OAP request events. The timeout value specifies in milliseconds the maximum time the API waits for a response before considering the request to have expired. If the application does not set a timeout value, the API uses a default value, which is specified in the configuration file.

Figure 31 shows an example of creating an outgoing OAP request event.

Figure 31 Example of creating an outgoing OAP request event

```
// Creating a new request
  OA_RCODE RC = OA_UNKNOWN_RC;
  MY_Context* myCxt = (MY_Context*)cxt;

  OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
  OA_AbstractEvent* outEvent = OA_NULL;

  RC = oapOpFactory->createRequestOapOpEvent( outEvent,
                                              OA_OPID_VOICE_CONNECT_REQUEST );

  if ( RC != OA_RC_SUCC )
  {
    OA_Log::swerr("CreateOAPOpError")
      << "Error received while creating voice connect.  RC = " << RC
      << "\nFound at " << __FILE__ << " line " << __LINE__
      << OA_Log ::end;

    return OA_INT_TASK_NULL;
  }

  // Get operation from event to add additional datablocks
  OA_Operation* outOp = (OA_Operation*)outEvent->getEventData( );

  // Set voice link logical channel.
  outOp->setField( myLogicalChan, OA_FID_LOGICAL_VOICE_CHANNEL_ID,
                   OA_DBID_VOICE_CHANNEL );
```

Outgoing events

Once an outgoing event has been created, the application must queue it for processing by the API. The context object provides a method, queueOutEvent, for queueing events for outbound processing. The applications specifies the out going event as the first parameter to this method and optionally the incoming event as the second parameter.

If the incoming event is specified, the source information of the incoming event is used for completing the addressing of the out going event. The application can specify OA_NULL in place of the incoming event when invoking queueOutEvent. If this is done, context data is used to complete the final addressing of the out going event. Queueing an out going event is shown in Figure 32 “Queueing out bound event”.

Figure 32 Queueing out bound event

```
OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
OA_AbstractEvent* outEvent = OA_NULL;

RC = oapOpFactory->createRequestOapOpEvent( outEvent,
                                            OA_OPID_VOICE_RELEASE_REQUEST );

if ( RC != OA_RC_SUCC )
{
  OA_Log::swerr("CreateOAPOpError")
    << "Error received while creating voice release.  RC = " << RC
    << "\nFound at " << __FILE__ << " line " << __LINE__
    << OA_Log ::end;

  return OA_INT_TASK_NULL;
}

cxt->queueOutEvent( outEvent, inEvent );
```

Timeouts

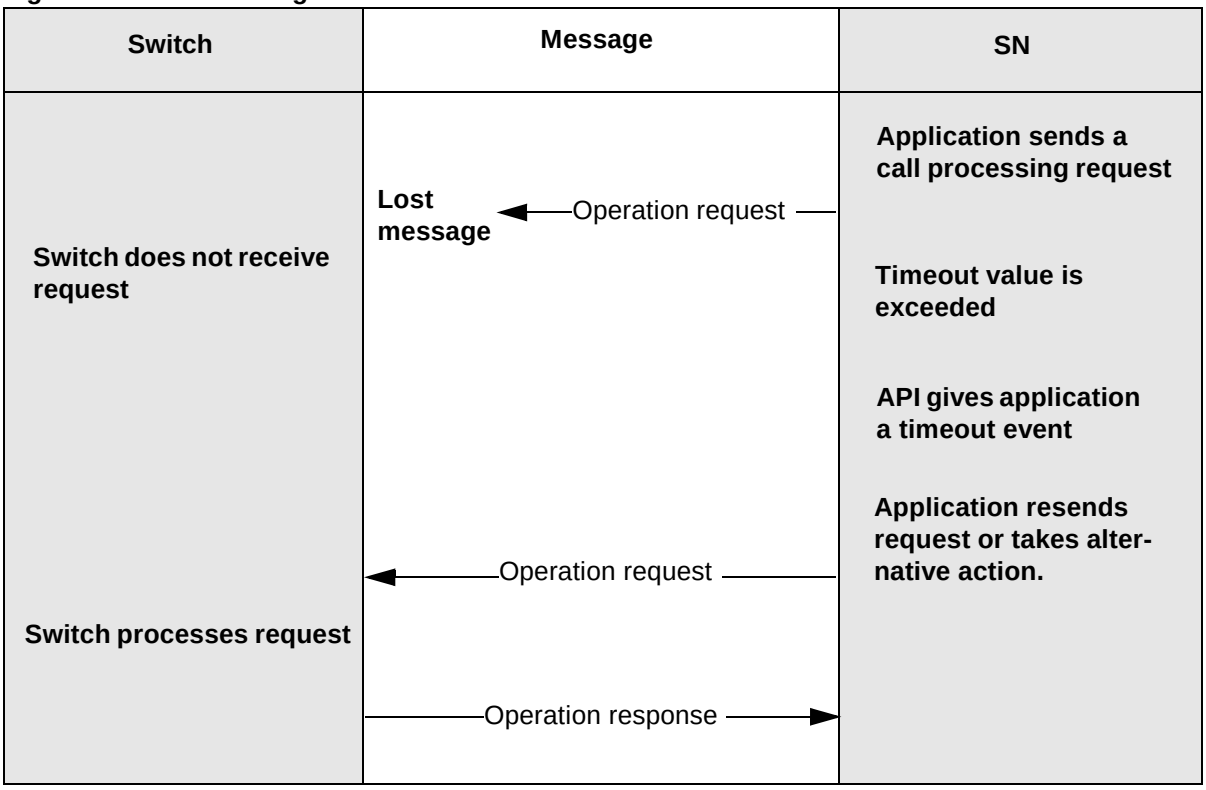
When the application sends an outgoing request, the API monitors incoming messages for a response to that request. If a response does not come in the allowed time interval, the request is considered expired.

Examples of timeout scenarios include:

- The request message was lost in transport to the switch.
- The switch received the request, but did not send a response within the allowed time interval.
- The switch sent a response, but it was lost in transport to the SN.

When a request expires, the API gives the application an API internal event with a sub-type, OA_INT_TASK_REQUEST_TIMEOUT, indicating a request timeout. The event data of this event is the request operation that has just timed out. Figure 33 shows the messaging for a timeout scenario due to a lost request message.

Figure 33 OAP message flow—timeout



When a timeout event is received, the request operation can be obtained from the event using the getEventData method on the event object. The application can then resend the request using the operation. This is illustrated in Figure 34 “Example of handling the expiration of a request”.

Figure 34 Example of handling the expiration of a request

```
// Waiting for response to my reqest
if ( inEvent->getEventType( ) == OA_OAP_API_INTERNAL_EVENT_TYPE )
{
  if ( inEvent->getEventSubType( ) == OA_INT_TASK_REQUEST_TIMEOUT )
  {
    // Request timed out.  Obtain operation from event to resend
    OA_Operation* outOp = (OA_Operation*)inEvent->getEventData( );

    // Turn operation into event
    OA_AbstractEvent* outEvent = new OA_OapEvent( outOp );

    // If resend count is less then configured max, resend request.
    if ( resendCount < RESEND_MAX )
      cxt->queueOutEvent( outEvent, OA_NULL );
  }
}
```

Wakeup timer

The context provides methods for setting and canceling a wakeup timer event. The state can use a wakeup timer to relinquish processing control until a specified amount of time has passed.

The setWakeupEvent() method sets a wakeup timer, specified in milliseconds, for the current session. This method updates the invokeId parameter found in the setWakeupEvent() method with a unique value used to distinguish the wake up event from others. This allows an application to have multiple timed events outstanding for the same session.

After the timer expires, the current state for the session receives an API internal event with a event sub-type of OA_INT_TASK_WAKEUP. If multiple timers have been set, the invokeId of the event can be retrieved to determine which timer request has expired.

An event timer can be cancelled by invoking the cancelWakeupEvent() method on the context object. The invokeId of the timer request to be cancelled must be specified.

Figure 35 shows an example of setting and canceling a wakeup timer.

Figure 35 Example of setting and canceling a wakeup timer

```
OA_INTERNAL_TASK_TYPE
firstState::process(OA_AbstractEvent* inEvent, OA_AbstractContext* cxt)
{
  MY_Context* myCxt = (MY_Context*)cxt;
  OA_UWORD invokeId = 0;

  // Receive control again after 5 seconds
  //
  cxt->setWakeupEvent( 5000, invokeId );

  // Save invokeId in subclass of context for later
  myCxt->timerId = invokeId;

  setNextStateId( SECOND_STATE );
}

OA_INTERNAL_TASK_TYPE
secondState::process(OA_AbstractEvent* inEvent, OA_AbstractContext* cxt)
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_OAP_API_INTERNAL_EVENT_TYPE:

      if ( inEvent->getEventSubType( ) == OA_INT_TASK_WAKEUP )
      {
        // Timer expired.  Verify which one.
        MY_Context* myCxt = (MY_Context*)cxt;
        if ( myCxt->timerId == inEvent->getInvokeId )
        {
          // 5 second timer expired.
        }
      }
      break;
    // ...
  }
   return OA_INT_TASK_NULL;
}
```

Transitioning to the next state

The setNextStateId() method on the OA_State class is used to indicate which state should receive control for the next incoming event. Any user-defined subclasses of OA_State inherit this functionality. States are referenced by the ID they return with the getID() method. If no state is set, the current state is assumed to be the next state.

Figure 36 shows examples of states transitioning to each other.

Figure 36 Examples of states transitioning to each other

```
OA_UWORD MyFirstState::getId()
{
  return MYSTATE_FIRST;
}

void MyFirstState::process( OA_AbstractContext* cxt )
{
  // ...
  setNextStateId( MYSTATE_SECOND );
}


OA_UWORD MySecondState::getId()
{
  return MYSTATE_SECOND;
}

void MySecondState::process( OA_AbstractContext* cxt )
{
  // ...
  setNextStateId( MYSTATE_LAST );
}


OA_UWORD MyLastState::getId()
  return MYSTATE_LAST;
}

void MyLastState::process( OA_AbstractContext* cxt )
{
  // ...
  setNextStateId( MYSTATE_FIRST );
}
```

Putting it all together

This section has discussed various aspects of the process() method. Figure 37 and Figure 38 show an example of implementing a complete process() method.

Figure 37 Example of implementing the process() method

```
OA_INTERNAL_TASK_TYPE
MyFirstState::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  OA_UWORD channelID = 12;

  if ( ( inEvent->getEventType( ) == OA_OAP_EVENT_TYPE ) &&
       ( inEvent->getEventSubType( ) == OA_OPID_SESSION_BEGIN_INFORM ) )
  {
    //
    // Session begin inform.  Handle new call. Ignore any other event.
    //
    OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
    OA_AbstractEvent* outEvent = OA_NULL;

     oapOpFactory->createRequestOapOpEvent( outEvent, OA_OPID_VOICE_CONNECT_REQUEST );

    // Get operation from event to add additional datablocks
    OA_Operation* outOp = (OA_Operation*)outEvent->getEventData( );

    outOp->setField( channelID, OA_FID_VCHNNL_ID, OA_DBID_VOICE_CHANNEL );
    cxt->queueOutEvent( outEvent, inEvent );
    setNextStateId( MYSTATE_SECOND );
  }
  return OA_INT_TASK_NULL;
}
```

Figure 38 Example of implementing the process() method (continued)

```
OA_INTERNAL_TASK_TYPE
MySecondState::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_OAP_EVENT_TYPE:
      //
      // OAP Event
      //
      switch ( inEvent->getEventSubType( ) )
      {
        case ( OA_OPID_VOICE_CONNECT_RETRES ):
          //
          // Voice connect success.
          // Continue with success path of call flow
          //
          break;
        default:
          //
          // Handle unexpected OAP operation
          //
      }
      break;

    case OA_OAP_API_INTERNAL_EVENT_TYPE:
      //
      switch ( inEvent->getEventSubType( ) )
      {
        case ( OA_INT_TASK_REQUEST_TIMEOUT ):
          //
          // Handle voice connect request time out.
          //
          break;

        default:
          //
          // Handle unexpected internal event
          //
      }
      break;

    default:
      //
      // Handle unexpected event type
      //
  }
}
```

Non-blocking code

The API framework can be run in a single or multi-threaded form. Even in the multi-threaded model, a single thread is used to process each event source. An event source is any source of incoming application events such as OAP messages being read from a UDP socket.

Because single thread is used for each event source, no other calls are serviced while the process() method for a state is executing. The state should *not* perform time consuming tasks, such as a blocking system call, because other state machines may not receive enough processing time.

Figure 39 shows an example of using a blocking system call, which adversely affects the operation of the SN.

Figure 39 Example of a blocking system call

```
// Don't do this!
OA_INTERNAL_TASK_TYPE
MyState::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  int transportSocket;   // initialized to socket in blocking mode
  int bytesReceived;
  char buffer[1024];
  struct sockaddr_in ip;
  int ipsize = sizeof( ip );

  // Function call does not return until a message arrives on the socket.
  // No other SN processing is performed while waiting; call processing
  // is halted.
  //
  bytesReceived = recvfrom( transportSocket, buffer, sizeof(buffer), 0,
                    &(ip), &ipsize );
}
```

reset() method

The reset() method is called when the conversation is over. The application should call the reset() method to clean up any call processing resources used for the conversation. It also calls the reset() method if the SN or a session pool is taken out of service while a conversation is active.

The cause for the call on the reset method is provided by the mtcTask parameter of the reset method. The application can use this information to determine reset actions to perform based on the cause of the reset.

Figure 40 shows an example of implementing the reset() method.

Figure 40 Example of implementing the reset() method

```
MyLastState::reset( OA_INTERNAL_TAK_TYPE task, OA_AbstractContext* cxt )
{
  // Tell my sub class of the context object to reset
  MY_Context* myCxt = (MY_Context*)cxt;
  myCxt->reset( );

  setNextStateId( MYSTATE_FIRST );
}
```

getId() method

The getId() method should return an ID that is unique among the state classes. One way to achieve this is to define an enumerated type, preferably in the header file, with one entry for each state.

getStatus() method

The getStatus() method should return whether the state is part of an active call (OA_STATUS_ACTIVE) or the idle period between conversations (OA_STATUS_IDLE). Most state models have one (or very few) idle states that wait for the start of a new session, and several active states that represent points in the call flow.

Step 3—Extending the API maintenance classes

The API provides state models for handling node and session pool maintenance conversations. These states respond to maintenance requests from the switch with successful replies when appropriate.

If the application does not modify these *default* maintenance states, the API always responds to test requests and return to service (RTS) requests with a success response. This means that the SN should be ready to accept calls at any time.

However, if there are any conditions under which the SN should not accept calls, application developers should extend the maintenance behaviors to control when the node or session pool comes into service. For example, the application may need to check the status of various resources or wait for some external event before it accepts calls.

Extending API maintenance behavior involves creating a subclass of the OA_NodeBusy state to control node maintenance, or a subclass of the OA_PoolBusy state to control session pool maintenance (or creating both subclasses, if needed). Applications should override the diagnostic() method to control the response to test requests, and override the readyForService() method to control the response to RTS requests.

diagnostic() method

The diagnostic() method checks to find any problems with the node or a session pool. This method returns with a return code of OA_RC_SUCC if the diagnostic succeeded and a success response should be sent, or with any other return code if the diagnostic failed and an error response should be sent.

This method uses the reportText argument to supply text that reports the success or a failure reason. This text is displayed on the switch maintenance and administration (MAP) position. The method should not attempt to send the actual response; the return code and report text are sufficient.

Note: The report text has a 32-character limit.

If the application does not override the diagnostic() method, any test requests receive a success response. Figure 41 shows an example of overriding the diagnostic() method for node maintenance.

Figure 41 Example of overriding the diagnostic() method

```
class MyNodeBusy : public OA_NodeBusy
{
  OA_RCODE diagnostic( OA_AbstractContext* cxt,
                       char* &reportText );
};


OA_RCODE MyNodeBusy::diagnostic( OA_AbstractContext* cxt,
                                 char* &reportText )
{
  if ( /* application provides test here */ )
  {
    // Test successful
    //
    reportText = "Test successful";

    return OA_RC_SUCC;
  }
  else
  {
    // Test failed
    //
    reportText = "Test failed; reason: xxx";

    return OA_RC_NOTAVAILABLE;
  }
}
```

readyForService() method

The readyForService() method checks if the node or a session pool is ready to come into service. Once a session pool comes into service, it should be ready to perform call processing duties immediately. This method returns with a return code of OA_RC_SUCC if a success response should be sent, or with any other return code if an error response should be sent.

This method uses the reportText argument to supply text that reports the success or a failure reason. This text is displayed on the switch. The method should not attempt to send the actual response; the return code and report text are sufficient.

If the application does not override the readyForService() method, any RTS requests receive a success response, and the node or session pool comes into service. Session pools should be careful not to come into service until they are prepared to handle call processing.

Figure 42 shows an example of overriding the readyForService() method for session pool maintenance replies.

Figure 42 Example of overriding the readyForService() method

```
class MyPoolBusy : public OA_PoolBusy
{
  OA_RCODE readyForService( OA_AbstractContext* cxt,
                    char* &reportText );
};



OA_RCODE MyPoolBusy::readyForService( OA_AbstractContext* cxt,
                          char* &reportText )
{
  if ( /* application provides test here */ )
  {
    // Test successful
    //
    reportText = "Session pool in service";

    return OA_RC_SUCC;
  }
  else
  {
    // Test failed
    //
    reportText = "RTS failed; reason: xxx";

    return OA_RC_NOTAVAILABLE;
  }
}
```

Step 4—Creating a subclass of the API framework

The OA_Framework class creates and configures objects from classes provided by the API, and it provides a basic flow of control method. This step discusses how to extend the framework to create and configure classes provided by the developer. Refer to “Step 5—Implementing the stand-alone application” on page 96 for information on the basic flow of control method.

Creating a subclass of OA_Framework involves implementing the buildCallpStates() method to create objects for application states. Also, the application must override the buildNodeMtcStates() method or the buildPoolMtcStates() method (or both) to create the extended maintenance states. After reading a configuration data file, the OA_Framework class calls these methods to create states needed by node maintenance, session pool maintenance, and call processing state machines.

buildCallpStates() method

The buildCallpStates() method creates objects from the application’s call processing state classes. The following two attributes of the OA_Framework class need to be configured:

- The sesnStateList attribute is a list of all call processing states. The getId() method of each state is used as its unique key in this list.
- The sesnStartState attribute should be set to the initial idle state in the state model.

Figure 43 shows an example of implementing the buildCallpStates() method.

Figure 43 Example of implementing the buildCallpStates() method

```
#include "oa/framewrk.h"

class MyFramework : public OA_Framework
{
  OA_RCODE buildCallpStates();
};

OA_RCODE MyFramework::buildCallpStates()
{
  // First state
  //
   OA_AbstractState* firstState = new MyFirstState();

   sesnStateList->add( firstState->getId(), firstState );

  // Second state
  //
  OA_AbstractState* secondState = new MySecondState();

  sesnStateList->add( secondState->getId(), secondState );

  // Add other states here

  sesnStartState = firstState;
  return OA_RC_SUCC;
}
```

buildNodeMtcStates() and buildPoolMtcStates() methods

The buildNodeMtcStates() method builds the OA_NodeBusy state. The application should override this method when it extends node maintenance behavior. Likewise, the buildPoolMtcStates() method builds the OA_PoolBusy state. The application should override this method when it extends session pool maintenance behavior.

Figure 44 shows an example of overriding the buildNodeMtcStates() method.

Figure 44 Example of overriding the buildNodeMtcStates() method

```
// Example for buildPoolMtcStates() should look very similar
// to this example

OA_RCODE MyFramework::buildNodeMtcStates()
{

  OA_AbstractState* nodeBusy = new MyNodeBusy();

  nodeStateList->add( nodeBusy->getId(), nodeBusy );

  // Use OA_Framework method to build other
  // node maintenance states
  //
  OA_Framework::buildNodeMtcStates();

  nodeStartState = nodeBusy;
  return OA_RC_SUCC;
}
```

Step 5—Implementing the stand-alone application

The developer can implement the application as a complete stand-alone SN by creating a framework object (using the application's framework subclass), and calling its methods to initialize the SN and begin processing.

Note: If the application is not a stand-alone application, refer to Chapter 6: "Advanced application topics" for information on how to integrate it with another application.

Initializing the node

The configure() method of the framework reads a configuration file and uses that data to initialize all needed components of the API. This method also calls the buildNodeMtcStates(), buildPoolMtcStates(), and buildCallpStates() methods implemented by the application to initialize the application state machines.

The application can call the configure() method with the configuration filename or with the list of command line arguments. If it is passed the command line arguments, the configuration filename is assumed to be the first argument. A return code indicates whether the configuration was successful.

Figure 45 shows an example of configuring the application.

Figure 45 Example of configuring the application

```
// Start SN like this:
//
// <exec> <filename>
//
// where <exec> is the executable program name
//   and <filename> is the configuration file name
//
int main( int argc, char** argv )
{
  OA_RCODE RC;
  OA_AbstractFramework* application = new MyFramework();

  RC = application->configure( argc, argv );

  // ...
}
```

Main processing loop

The processLoop() method of the framework contains the main processing loop for the application. The processing loop performs the following three steps until the SN shuts down:

- receive messages from the network
- process messages with their appropriate state machines
- send outgoing messages

Note: Refer to Chapter 6: "Advanced application topics" for information on how to implement an application-specific flow of control.

Figure 46 shows the processLoop() method.

Figure 46 processLoop() method of the OA_Framework class

```
OA_RCODE OA_Framework::processLoop( )
{
  OA_RCODE RC = OA_UNKNOWN_RC;

  while ( ( RC != OA_RC_SHUTDOWN ) &&
        ( RC != OA_RC_RESTART ) )
  {
    proxy->receive();

    if ( systemStatus->getStatus( ) == SHUTDOWN )
    {
       RC = OA_RC_SHUTDOWN;
    }
    else if ( systemStatus->getStatus( ) == RESTART )
    {
      RC = OA_RC_RESTART;
    }
    else
    {
       RC = node->process();
       if ( RC == OA_RC_SHUTDOWN )
       {
         systemStatus->setStatus( SHUTDOWN );
       }
       else if ( RC == OA_RC_RESTART )
       {
         systemStatus->setStatus( RESTART );
       }
       proxy->send();
    }
  }
  return RC;
}
```

Shutting down the node

The processLoop() method exits the main processing loop when receiving a return code of OA_RC_SHUTDOWN or OA_RC_RESTART from the node object or the system status object. The system status object is contained within the framework object and maintains the current state of the API application.

A shutdown or restart is initiated by invoking the setStatus() method of the system status object. The application developer determines when it is appropriate to modify the system status object to trigger a system shutdown or restart. For example, the developer may install a signal handler that modifies the system status to indicate a shutdown event when the control-C key sequence is entered at a command line.

Creating a main function

Figure 47 shows an example of implementing a stand-alone SN application.

Figure 47 Example of implementing a stand-alone SN application

```
int main( int argc, char** argv )
{
  OA_RCODE RC;
  OA_AbstractFramework* application = new MyFramework();
  int exitcode;

  RC = application->configure( argc, argv );

  if ( RC == OA_RC_SUCC )
  {
    exitcode = application->go();
  }

  delete application;

  exit( exitcode );
}
```

Chapter 6: Advanced application topics

The basic application approach described in the previous chapter works well when developing a new application from the ground up. When a developer wishes to integrate OAP into the existing application, two approaches can be used.

- integrating OAP with an API maintenance framework and PI layer
- utilize the Corba based OAP Server application

The objective of each of these approaches is to enable the application developer to build OAP compatible applications. Key enablers are the automatic handling of OAP maintenance events and the decoding and encoding of OAP call processing events.

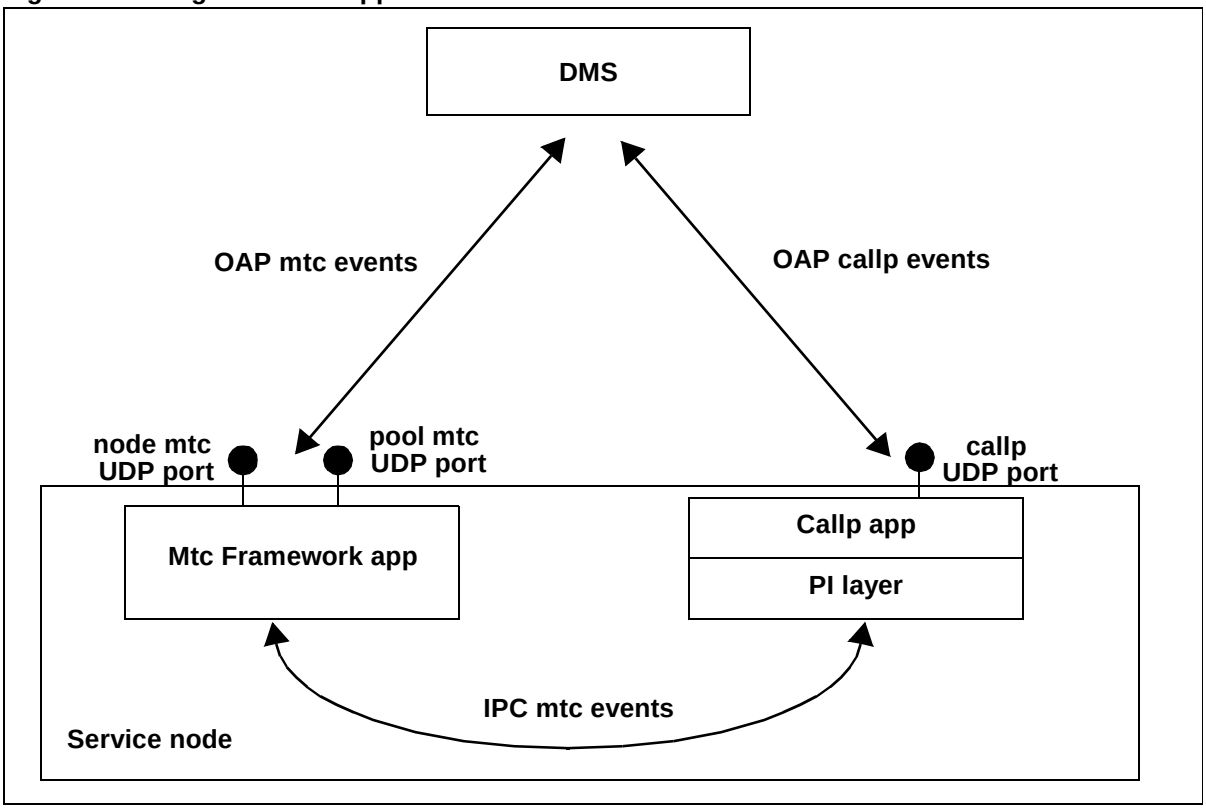
API maintenance framework and PI layer

OSSAIN allows for the use of different UDP port identifiers for node, session pool, and call processing class messages. Because of this, it is possible to direct maintenance class messages (node and session pool messages) and call processing messages to separate processes on the same service node. This approach introduces a maintenance framework application for automatic handling of OAP maintenance events.

The maintenance framework application is built from standard OAP API classes but only handles OAP maintenance events. Call processing events are handled directly by the call processing application. The application developer has the option of utilizing the PI (protocol interface layer) of the API to facilitate processing of the OAP call processing events. Additionally, an IPC (inter process communication) device can be used between the maintenance framework application and the call processing application to inform the call processing application of maintenance events.

This approach is illustrated in Figure 48 “Integrated OAP application'.

Figure 48 Integrated OAP application



Building the maintenance framework application

The maintenance application is built using the API framework class. The API automatically handles OAP node and session pool maintenance events. In most cases, it is necessary for the call processing application to be informed of major OAP maintenance events such as the node or session pool being brought into service. To do this, API node and session pool maintenance classes are subclassed in order to inform the call processing application of maintenance events.

Subclassing node and session pool maintenance

By subclassing the node and session pool maintenance states, the developer can coordinate maintenance events between the maintenance framework application and the call processing application.

At a minimum, the diagnostic() and the readyForService() methods of the OA_NodeBusy and OA_PoolBusy classes should be overriden. The reset() and process() methods of the OA_NodeInSvc and OA_PoolInSvc classes should also be overriden.

Figure 49 “Example header file for session pool state subclass' shows the definition of a session pool maintenance subclass.

Figure 49 Example header file for session pool state subclass

```
#include "oa/poolmtc.h"

class My_PoolBusy : public OA_PoolBusy
{
  public:
    OA_RCODE diagnostic( OA_AbstractContext* cxt, char* &reportText );
    OA_RCODE readyForService( OA_AbstractContext* cxt, char* &reportText );
};

class My_PoolInSvc : public OA_PoolInSvc
{
  public:
    void reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt );
    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
}
```

The diagnostic() method is invoked when the node or session pool is in the busy state and an OAP test request is received. By overriding the diagnostic method, the application has the opportunity to perform application specific checks when the OAP test request is received.

The readyForService() method is invoked when an OAP RTS request is received and the node or session pool is in the busy state. This method provides an opportunity for the maintenance application to send an event to the call processing application that a session pool is coming into service. A simple example is shown in Figure 50. In this example, a pipe is used as the IPC device for sending string events to the call processing application from the maintenance application. After the string event is written to the pipe, the readyForService() method of the superclass is invoked to receive default behavior for this method.

Figure 50 Example of session pool readyForService method

```
OA_RCODE My_PoolBusy::readyForService( OA_AbstractContext* cxt,
                                       char* &reportText )
{
  // Build RTS notification event and send via pipe to callp app
  char pipeBuff[32];
  ostrstream oss( pipeBuff, 32, ios::out );
  OA_AbstractCommunicationContext* oapComCxt =
    cxt->getCommContext( OA_OAP_EVENT_TYPE );

  // We will send a string indicating an RTS occurred and the OAP
  // protocol version to use.
  oss.seekp(ios::out);
  oss << "RTS_EVENT:"<< oapComCxt->getMyCurrentVersion( ) << ends;

  // Write the string to our mtc pipe
  write( mtcPipe, pipeBuff, strlen(pipeBuff) );

  // Receive default processing from the super class
  return OA_PoolBusy::readyForService( cxt, reportText );
}
```

The reset() method is handled in a similar manner as the readyForService method. The reset method is invoked when the state of the node or session pool is being reset due to a maintenance event. This method is overridden in order to inform the call processing application of a change of state from in service to busy. An example implementation of the reset() method is shown in Figure 51.

Figure 51 Example of session pool reset method

```
OA_RCODE My_PoolBusy::reset( OA_INTERNAL_TASK_TYPE mtcTask,
                             OA_AbstractContext* cxt )
{
  if ( mtcTask == OA_INT_TASK_RESETBUSY )
    // Session pool going out of service
    write( myPipe, "SPBusy", 6 );

  // Receive default processing from the super class
  return OA_PoolBusy::reset( mtcTask, cxt );
}
```

In the OSSAIN centralized environment, service nodes are kept up to date on whether or not a remote OSSAIN switch has the service node in a busy or in service state. This state information is made available to the service nodes through a session pool state inform message from the remote switch to the service node. It is necessary to override the session pool process method in order to inform the call processing application when a session pool state inform message is received.

Figure 52 Example of session pool process method

```
OA_INTERNAL_TASK_TYPE
My_PoolInSvc::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  if ( ( inEvent->getEventType == OA_OAP_EVENT_TYPE ) &&
       ( inEvent->getEventSubType == OA_OPID_SESSION_POOL_STATE_INFORM ) )
  {
    // Session pool state inform.  Extract switch id, oap version, and
    // session pool state to send to call processing application
    OA_Operation* inOp = (OA_Operation*)inEvent->getEventData( );
    OA_UWORD oapVersion = 0;
    OA_UWORD status = 0;
    OA_UWORD switchId = 0;
    char pipeBuff[64];
    ostrstream oss( pipeBuff, 64, ios::out );

    inOp->getField( &oapVersion, OA_CH_MSGVERSION, OA_CLASS_HEADER );
    inOp->getField( &switchId, OA_CH_SOURCENODEID, OA_CLASS_HEADER );
   inOp->getField( &status, OA_FID_MAINTENANCE_STATE, OA_DBID_MAINTENANCE_STATE
);

    oss.seekp(ios::out);
    oss << "POOL_STATEINF:POOLID:"<< cxt->getPoolId( )
       << ":OAPVER:" << oapVersion << ":SWITCHID:" << switchId
       << ":MTCSTATE:" << status << ends;

    // Write the string to our mtc pipe
    write( mtcPipe, pipeBuff, strlen(pipeBuff) );

  }

  return OA_PoolInSvc::process( inEvent, cxt );
}
```

Subclassing the framework

In this approach, the framework class is subclassed to achieve two goals. The first is to add our subclassed node and session pool maintenance states to the node and session pool state machines. The second it to provide an additional event source (IPC device) to the framework flow of control. This is done so that the call processing application can initiate maintenance events such as throttling a session pool or placing a node out of service.

The framework's buildNodeMtcStates() and buildPoolMtcStates() are overridden so that our custom maintenance states are added to the API framework's state machine. The buildCallpStates() method is also overridden since every subclass of the OA_Framework class must provide an implementation of the buildCallpStates() method.

An example implementation of the subclassed framework is shown in Figure 53.

Figure 53 Example framework subclass implementation

```
#include "oa/framework.h"
#include "myMtcStates.h"

class My_Framework : public OA_Framework
{
  public:
    OA_RCODE buildNodeMtcStates( )
      {
        addNodeMtcState( new My_NodeBusy( ) );
        addNodeMtcState( new My_NodeInSvc( ) );
        return OA_Framework::buildNodeMtcStates( );
      }

    OA_RCODE buildPoolMtcStates( )
      {
        addPoolMtcState( new My_PoolBusy( ) );
        addPoolMtcState( new My_PoolInSvc( ) );
        return OA_Framework::buildPoolMtcStates( );
      }

    OA_RCODE buildCallpStates( )
      {
        return OA_RC_SUCC;
      }

    // .....
};
```

Adding external event sources

In a single threaded environment, the framework's processLoop() method can be overridden to introduce the IPC device as an external event source. During the event flow processing of the processLoop() method, the IPC device can be checked for incoming events from the call processing application. If one is found, and application event can be created and placed on the inbox for processing. The node and session pool maintenance classes can be further refined to handle these call processing application originated maintenance events.

It is important to note that the attempt to receive an event from the IPC device should not block in the single threaded environment. This will cause the entire framework to block until an event is received from the IPC device.

Figure 54 " Example implementation of the processLoop method' shows how the processLoop() method can be modified to include polling of an IPC device.

Figure 54 Example implementation of the processLoop method

```
#include "oa/framework.h"
#include "myMtcStates.h"

class My_Framework : public OA_Framework
{
    // .....
  OA_RCODE processLoop( )
    {
      OA_RCODE RC = OA_UNKNOWN_RC;
      while ( ( RC != OA_RC_SHUTDOWN ) &&
            ( RC != OA_RC_RESTART ) )
      {
        proxy->receive( );
        IPCDevice->receive( );

        if ( systemStatus->getStatus( ) == SHUTDOWN )
          RC = OA_RC_SHUTDOWN;
        else if ( systemStatus->getStatus( ) == RESTART )
          RC - OA_RC_RESTART;
        else
        {
          RC = node->process( );
          if ( systemStatus->getStatus( ) == SHUTDOWN )
            RC = OA_RC_SHUTDOWN;
          else if ( systemStatus->getStatus( ) == RESTART )
            RC - OA_RC_RESTART;
          proxy->send( );
        }
      }
    }
};
```

In a multi-threaded environment, an application thread can be added to perform a blocking read on an IPC device. Once an event is received on the IPC device, the application thread can lock the framework and instruct the node object to process the event.

The createApplicationThreads() method is used to start an application specific thread. This method is invoked by the framework object during start up. A dependency count is used by the framework to monitor the number of threads that have been started.

Figure 55 and Figure 56 show how an application specific thread can be started.

Figure 55 Starting an application thread in Windows NT

```
DWORD WINAPI startMyThread( LPVOID arg )
{
  My_Framework* framework = (My_Framework*)arg;
  return framework->processMyThread( );
}

OA_RCODE My_Framework::processMyThread( )
{
  OA_RCODE RC = OA_UNKNOWN_RC;
  frameworkLock->lock( );

  dependencyCount++;
  RC = processIPCLoop( );
  dependencyCount--;
  PulseEvent( dependencyEvent );

  frameworkLock->release( );

  return RC;
}

OA_RCODE My_Framework::processIPCLoop( )
{
  OA_RCODE RC = OA_UNKNOWN_RC;
  OA_AbstractEvent* event = OA_NULL;

  while ( systemStatus->getStatus( ) == RUNNING )
  {
    frameworkLock->release( );
    event = IPCDevice->receive();
    frameworkLock->lock( );
    if ( event != OA_NULL )
    {
      RC = node->process( event );

      if ( RC == OA_RC_SHUTDOWN )
        systemStatus->setStatus( SHUTDOWN );
      else if ( RC ==  OA_RC_RESTART )
        systemStatus->setStatus( RESTART );

      proxy->send();
    }
  }

  return RC;
}
```

Figure 56 Starting an application thread in Windows NT (continued)

```
void
My_Framework::createApplicationThreads( OA_UWORD &threadIndex )
{
  HANDLE myHandle = 0;
  OA_UWORD myThreadId = 0;

  myHandle =  CreateThread(NULL,0, startMyThread,
                  (LPVOID)this,0,(LPDWORD)&myThreadId);

  if ( myHandle == NULL )
  {
    LPVOID lpMsgBuf;

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
        );
    OA_Log::swerr("Thread create error")
      << "Error creating thread.  " << lpMsgBuf
      << OA_Log::end;

    LocalFree( lpMsgBuf );
  }
}
```

In the above example, the My_Framework::createApplicationThreads() method utilizes the NT system call CreateThread to create a thread using procedure startMyThread(). The startMyThread() procedure simply takes a pointer to the My_Framework object and invokes the processMyThread() method.

The first thing done by the processMyThead() method is to increment the framework thread dependency count. The framework object monitors the number of active threads with the dependency count attribute. Every time a new thread is started, the dependency count is incremented. Every time a thread ends, the dependency count is decremented.

Before the dependency count can be modified, the thread must have exclusive control of the framework. To do this, the framework is locked by invoking lock() on the frameworkLock object. Once this is done, all other threads are blocked until the release() method of the framework lock object is invoked.

The processIPCLoop() method is invoked to process incoming maintenance events from the IPC device. This method preforms blocking reads on the IPC device. Before doing so, the framework is unlocked via the release() method. The receive() method of the IPC device should block for some amount of time while waiting for an incoming event. This is to allow the application thread an opportunity to check the status of the system for restart or shutdown instructions if no external events are being received from the IPC device. If the receive() method returns due to an incoming event, the framework is locked and the node object is instructed to process the incoming event. Once the event has been processed, the proxy is instructed to send any waiting outgoing events. We then return to the top of the receive loop in order to perform another blocking receive on the IPC device being sure to release the framework lock prior to invoking the IPC device receive() method.

During shutdowns and restarts, a dependency event object is used as a signalling device by child threads to inform the framework's main thread of a child thread termination. Once all child threads have ended, signalled by the dependency count reaching zero, the framework can complete the shutdown or restart activity.

IPC maintenance events

In this approach, the IPC device is used to send maintenance events between the maintenance application and the call processing application. The developer determines the format of the events that are passed between the two application processes. The following list of events should be considered for coordination of activities between the two application processes.

- OAP node test request (mtc app to callp app)
- OAP node RTS request (mtc app to callp app)
- OAP node busy request (mtc app to callp app)
- OAP node audit request (mtc app to callp app)
- OAP session pool test request (mtc app to callp app)
- OAP session pool RTS request (mtc app to callp app)
- OAP session pool busy request (mtc app to callp app)
- OAP session pool audit request (mtc app to callp app)
- OAP session pool state inform (mtc app to callp app)
- Service node throttle request (callp app to mtc app)
- Service node busy request (callp app to mtc app)

Utilizing the PI layer

This approach provides for automatic handling of OAP maintenance events by the maintenance application. All OAP call processing interactions are the responsibility of the call processing application. Utilization of the PI (Protocol Interface) software layer in the call processing application facilitates the handling of OAP messages by the application. Refer to Chapter 4: "OAP interface" for a description of how to encode and decode OAP messages using only the PI layer.

Corba based OAP Server

The OAP Server is an intermediary process between the DMS and the service node application. Its role is to route OAP messages between the DMS and one or more call processing applications. OAP Server utilizes Corba interfaces for interprocess communications with application processes. The OAP Server is described in Appendix: "OAP Corba Server".

Chapter 7: Sample SN applications

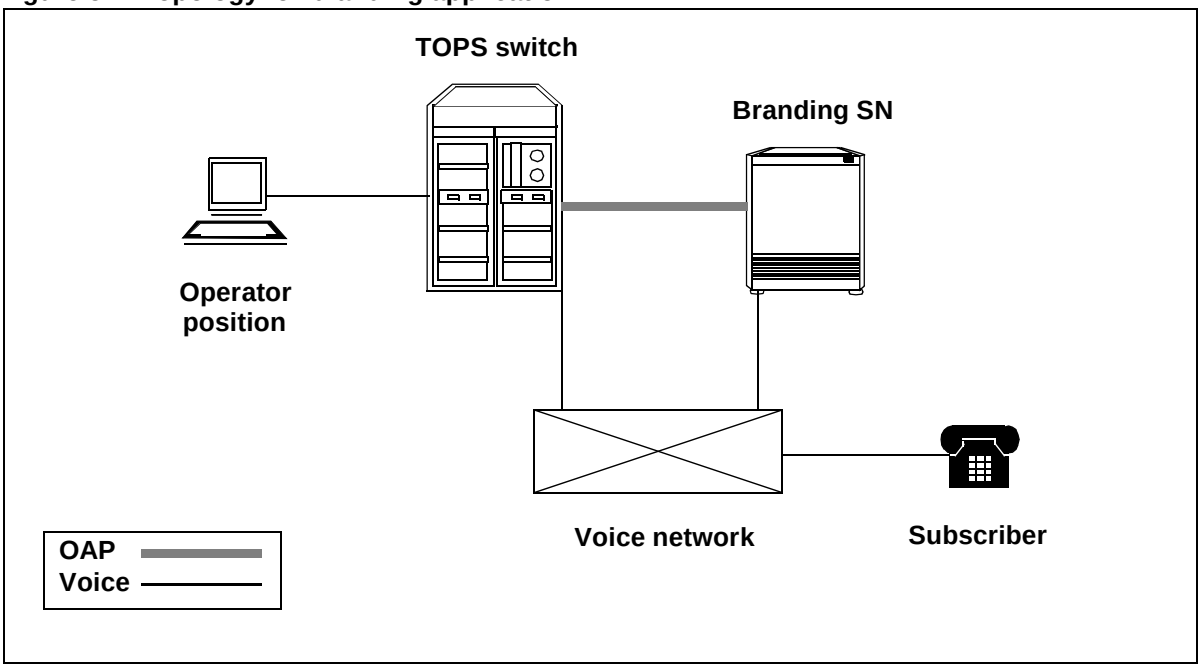
This chapter illustrates a sample SN application.

Sample basic standalone application

This section shows each step of developing a *branding* application for the SN. With branding, the SN controls the call to play an announcement (such as the name of the service provider) to the subscriber. The branding SN contains voice hardware that plays the announcement over a voice link in the OSSAIN network.

Figure 57 shows a simple network topology for the sample standalone branding application.

Figure 57 Topology for branding application



When the DMS switch determines that the SN should brand a call, the switch initiates an OAP session with the SN, giving it control of the call. The SN requests the switch to connect the call to a voice channel on the SN.

If the request is successful, the SN plays the announcement. After the announcement finishes, the SN requests the switch to transfer the call to an operator. If the request is unsuccessful, or if no voice channels are available, the SN requests the transfer to an operator without attempting to play the announcement. If the request to transfer the call is unsuccessful, or if the subscriber goes on hook, the SN requests the switch to end the call.

Sample voice hardware interface

In the sample branding application, the voice hardware has the following interface:

- VOCinit(), which initializes the voice hardware
- VOCtest(), which returns whether the voice hardware is ready (1 if ready, 0 if not)
- VOCallocChannel(), which returns an unused channel number or -1 if no channels are free
- VOCfreeChannel(), which frees a channel for other use
- VOCstart(), which starts playing an announcement on the specified channel

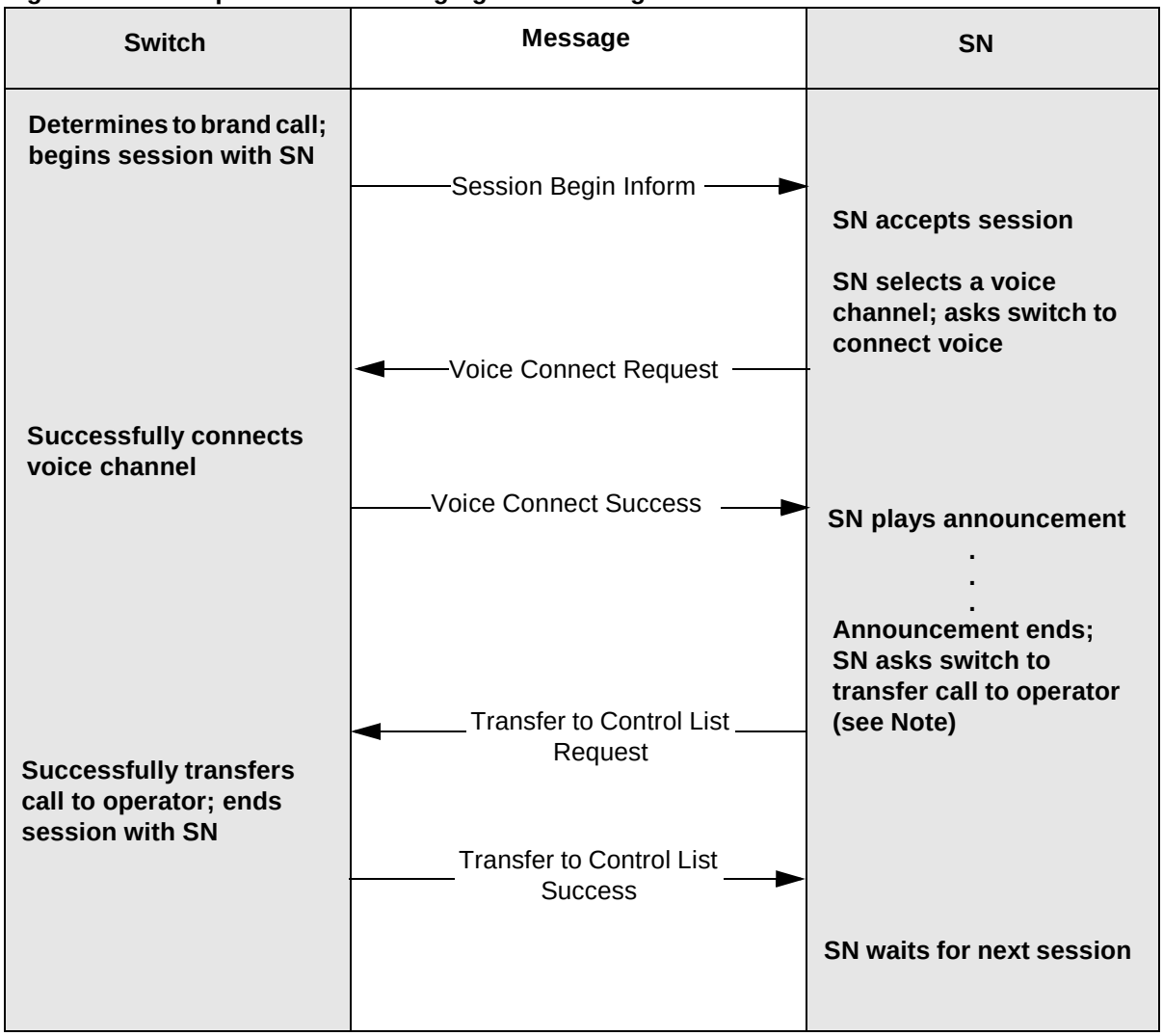
  Note: This is a non-blocking system call.
- VOCstop(), which stops playing the announcement on the specified channel
- VOCanncLength(), which returns the length in milliseconds of the specified announcement

Note: This sample voice hardware interface is for illustration purposes only. The API software does not provide a voice hardware interface. An actual application would use the interface provided by the voice hardware vendor.

Sample branding scenario message flows

Figure 58 shows the OAP message flow for the success path.

Figure 58 Example of OAP messaging for branding success



Note: OSSAIN provides a *control list* mechanism to move a call between a service provided by an SN and a service provided by another node or an operator. A control list datafilled at the DMS switch contains the name of a provider associated with the service (function). In the example, the SN sends a Transfer to Control List Request to transfer the call to an operator. For more information on function and control list datafill, refer to " Additional datafill" on page 197.

Figure 59 shows the OAP message flow for an error path when the switch cannot connect the voice channel.

Figure 59 Example of OAP messaging for voice connect error

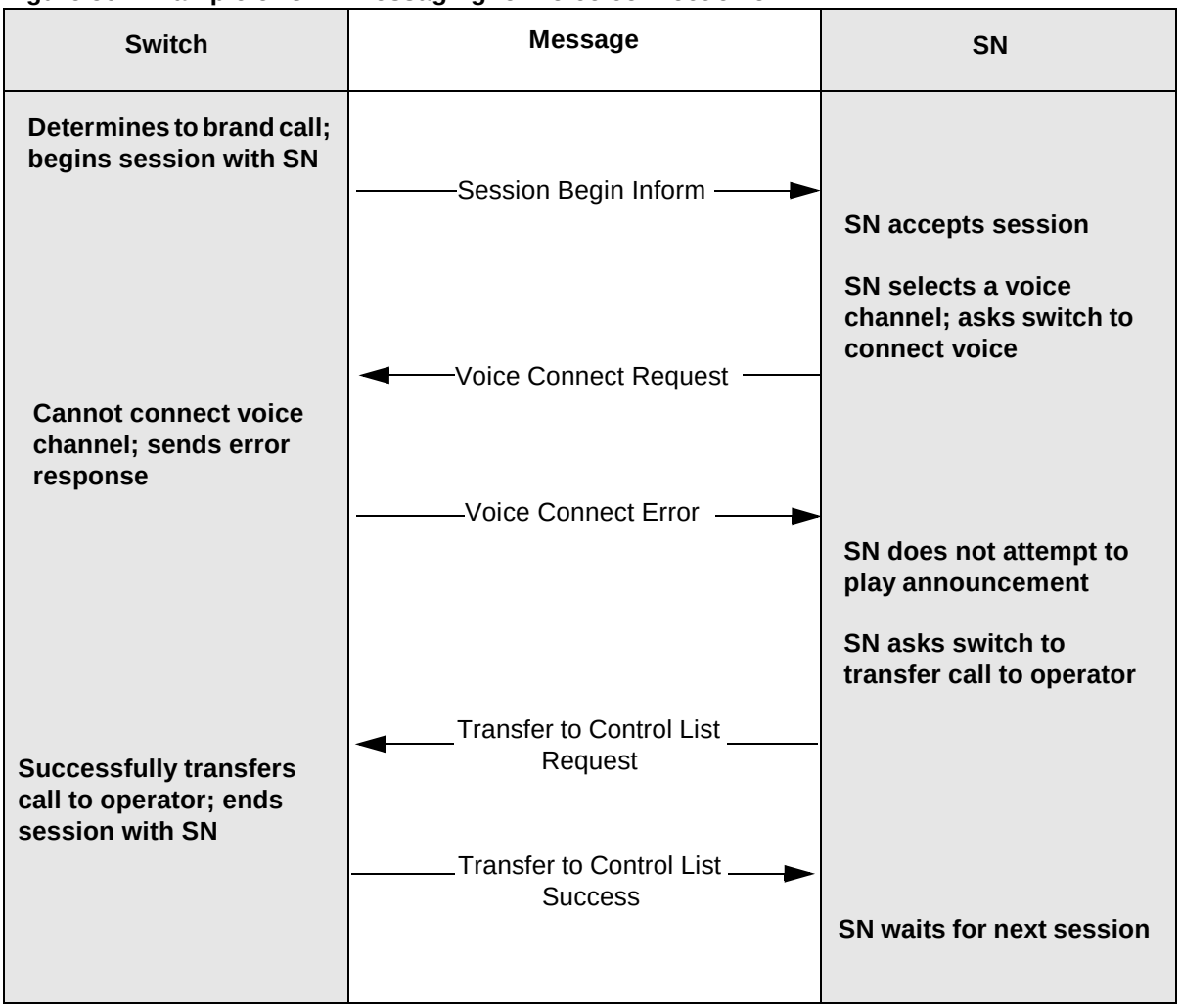
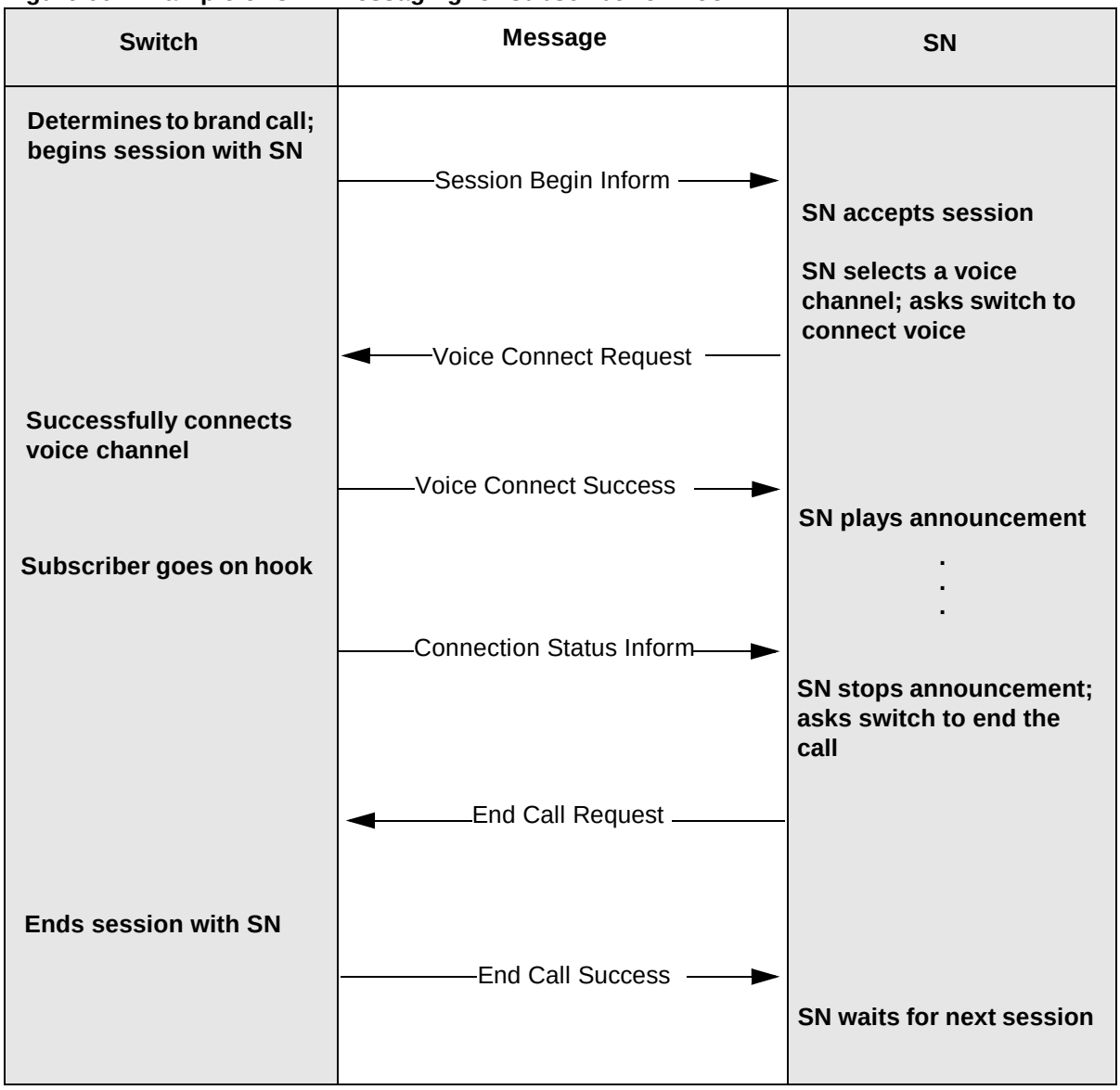


Figure 60 shows the OAP message flow when the subscriber goes on hook during the announcement.

Figure 60 Example of OAP messaging for subscriber on hook



The sample branding application proceeds through the following five development steps:

1 Designing a state model

2 Implementing the call state model

3 Extending the API maintenance classes

4 Creating a subclass of the OSSAIN API framework

5 Implementing the standalone application.

Note: For details on the steps, refer to Chapter 5: "Building a basic application."

Step 1—Designing a state model

This step shows the application call flow for each of the three sample branding scenario flows: success, error, and on hook. The following five call states appear in the flows:

- Waiting for Session
- Waiting for Connection
- Waiting for Announcement End
- Waiting for Transfer
- Waiting for Call End

Application tasks that correspond to OAP operations for the five call states are listed in the tables beginning on page 122.

Figure 61 shows the mapping between OAP operations and call states for the success path.

Figure 61 Mapping between OAP operations and call states—success path

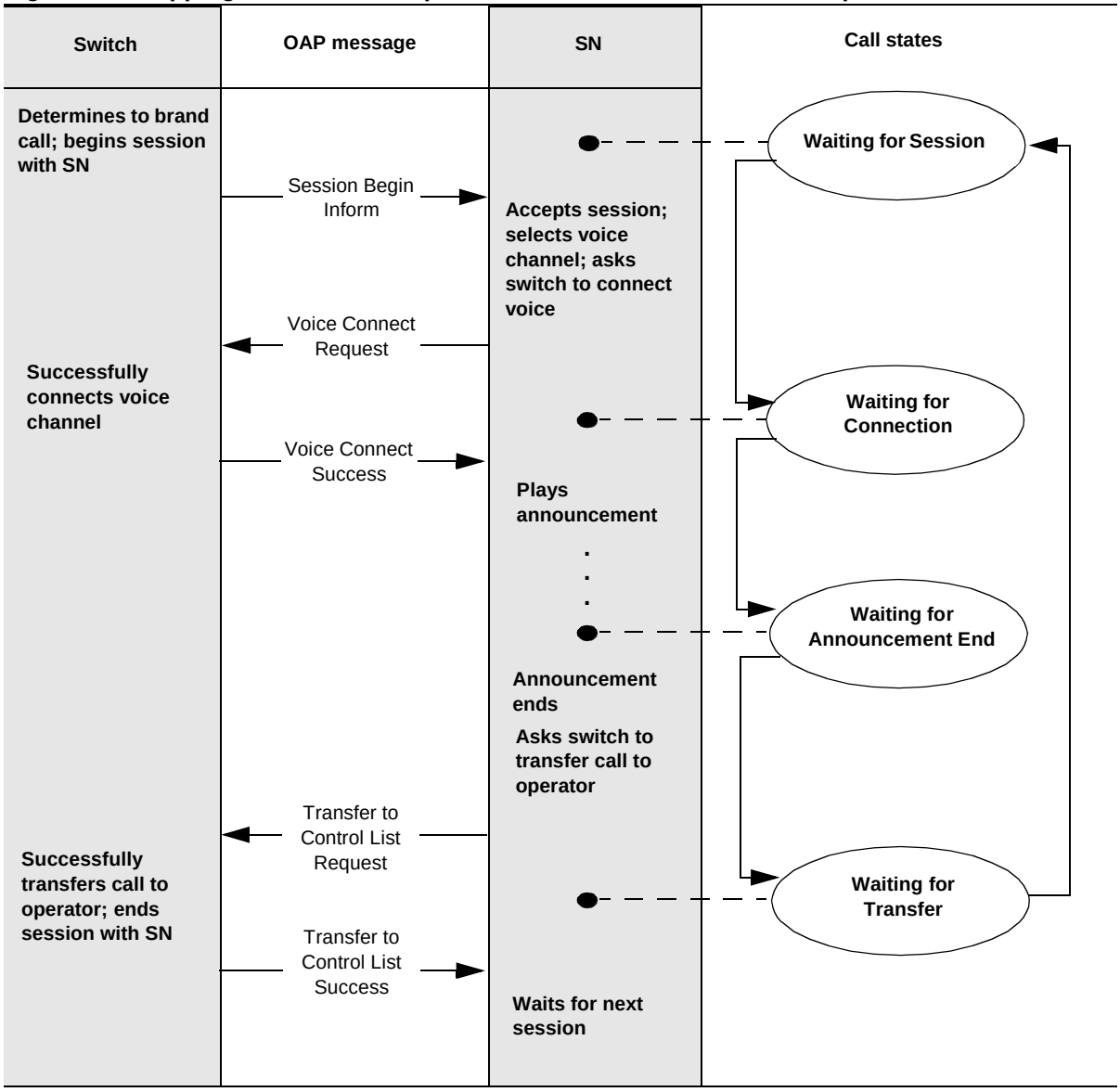


Figure 62 shows the mapping between OAP operations and call states for the voice connect error path.

Figure 62 Mapping between OAP operations and call states—voice connect error path

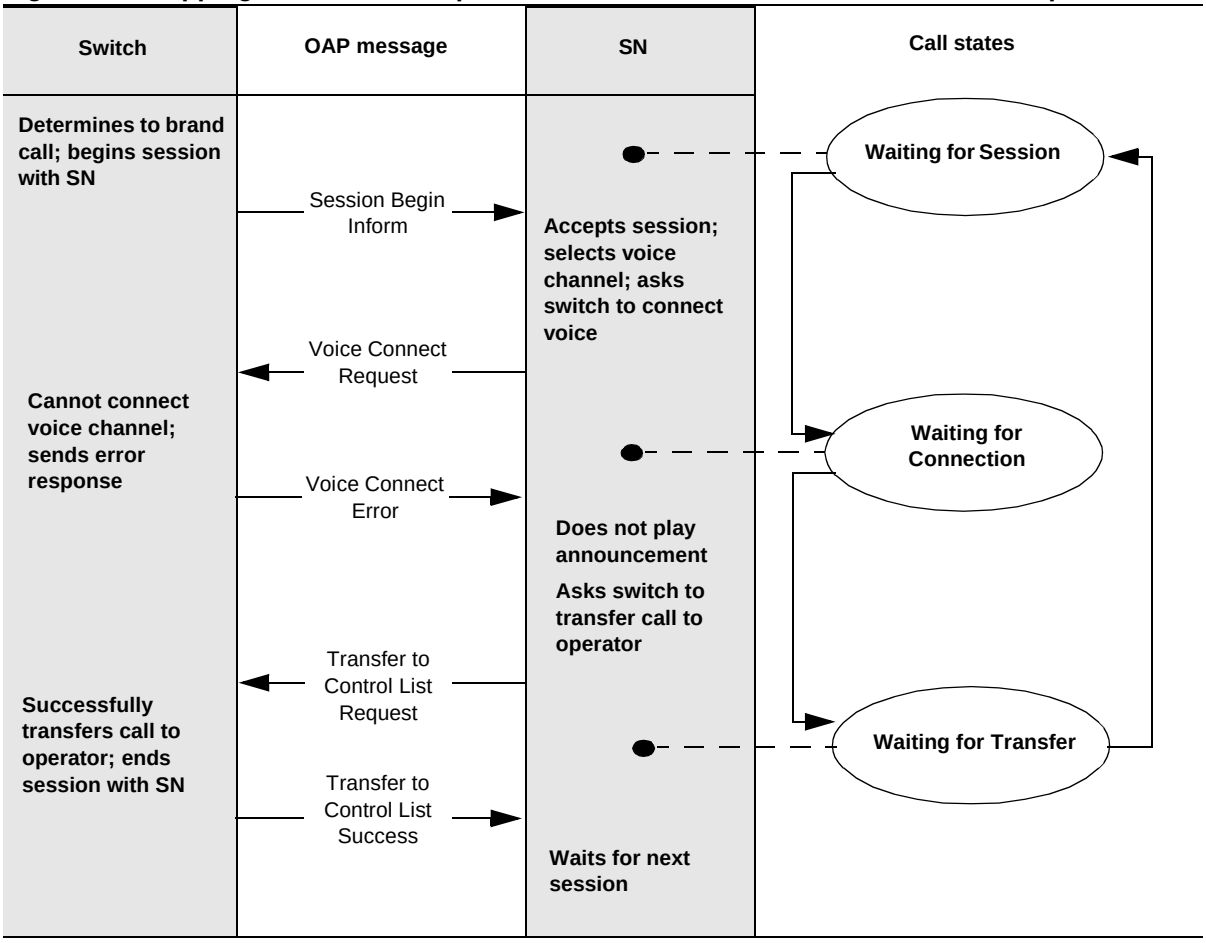
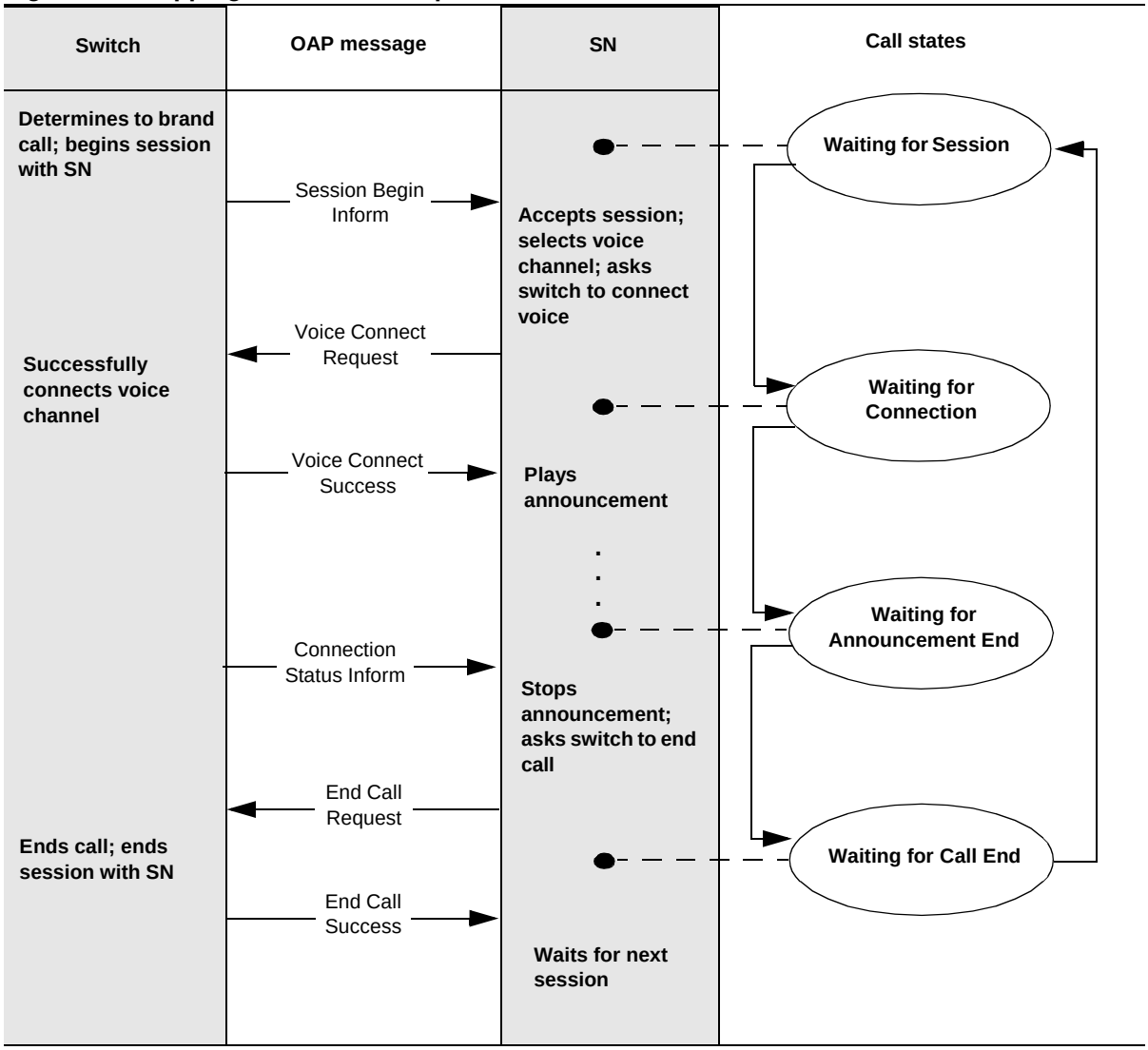


Figure 63 shows the mapping between OAP operations and call states for the subscriber on hook.

Figure 63 Mapping between OAP operations and call states—subscriber on hook



Waiting for Session

This initial state waits for a new Session Begin Inform operation. Table 5 lists the application tasks for each operation ID.

Table 5 Waiting for Session

Operation ID	Application task
Session Begin Inform	Allocates voice channel. If successful, sends Voice Connect Request and goes to Waiting for Connection state. If unsuccessful, sends Transfer to Control List Request and goes to Waiting for Transfer state.
Other	Ignores error and continues in Waiting for Session state.

Waiting for Connection

This state waits for the response to a Voice Connect Request. Table 6 lists the application tasks for each operation ID.

Table 6 Waiting for Connection

Operation ID	Application task
Voice Connect Success	Starts playing announcement and sets Wakeup. Goes to Waiting for Announcement End state.
CallP Error Response	Timeout: Resends request once. After one resend, frees channel and sends Transfer to Control List Request. Goes to Waiting for Transfer state. Already connected: Starts playing announcement and sets Wakeup. Goes to Waiting for Announcement End state. Other error: Frees channel and sends Transfer to Control List Request. Goes to Waiting for Transfer state.
Connection Status Inform	On hook or release: Frees voice channel and sends End Call Request. Goes to Waiting for Call End state. Off hook or flash: Ignores and continues in Waiting for Connection state.
Session Begin Inform	Frees voice channel and goes to Waiting for Session state, where it handles the Session Begin Inform.
Call End Inform	Frees voice channel and goes to Waiting for Session state.
DTMF Digit Inform	Ignores and continues in Waiting for Connection state.
Other	Frees voice channel and sends End Call Request. Goes to Waiting for Call End state.

Waiting for Announcement End

This state waits for the end of the announcement. Table 7 lists the application tasks for each operation ID.

Table 7 Waiting for Announcement End

Operation ID	Application task
Wakeup	Stops announcement and frees voice channel. Sends Transfer to Control List Request and goes to Waiting for Transfer state.
Connection Status Inform	On hook or released: Stops announcement and frees voice channel. Sends End Call Request and goes to Waiting for Call End state. Off hook or flash: Ignores and continues in Waiting for Announcement End state.
Session Begin Inform	Stops announcement and frees voice channel. Goes to Waiting for Session state, where it handles the Session Begin Inform.
Call End Inform	Stops announcement and frees voice channel. Goes to Waiting for Session state.
DTMF Digit Inform	Ignores and continues in Waiting for Announcement End state.
Other	Stops announcement and frees voice channel. Sends End Call Request and goes to Waiting for Call End state.

Waiting for Transfer

This state waits for the completion of the transfer to operator. Table 8 lists the application tasks for each operation ID.

Table 8 Waiting for Transfer

Operation ID	Application task
Transfer to Control List Success	Goes to Waiting for Session state.
Callp Error Response	Timeout: Resends request once. After one resend, sends End Call Request and goes to Waiting for Call End state. Other error: Sends End Call Request and goes to Waiting for Call End state.
Connection Status Inform	On hook or released: Sends End Call Request and goes to Waiting for Call End state. Off hook or flash: Ignores and continues in Waiting for Transfer state.
Session Begin Inform	Goes to Waiting for Session state, where it handles the Session Begin Inform.
Call End Inform	Goes to Waiting for Session state.
DTMF Digit Inform	Ignores and continues in Waiting for Transfer state.
Other	Sends End Call Request and goes to Waiting for Call End state.

Waiting for Call End

This state waits for the end of the call. Table 9 lists the application tasks for each operation ID.

Table 9 Waiting for Call End

Operation ID	Application task
End Call Success	Goes to Waiting for Session state.
Callp Error Response	Resends request once. After one resend, goes to Waiting for Session state.
Connection Status Inform	Ignores and continues in Waiting for Call End state.
Session Begin Inform	Goes to Waiting for Session state, where it handles the Session Begin Inform.
Call End Inform	Goes to Waiting for Session state.
DTMF Digit Inform	Ignores and continues in Waiting for Call End state.
Other	Ignores error and continues in Waiting for Call End state.

Step 2—Implementing the call state model

This section shows examples of the header file and source file for the branding call processing states. The header file contains definitions for each of the state subclasses. Figure 64 and Figure 65 show example code in the header file, brcallp.h.

Figure 64 Example header file for branding call processing states

```
#include "oa/state.h"
#include "oa/log.h"
#include "brcxt.h"
#include "voiceapi.h"     // Header file with voice hardware API

ypedef enum
{
  UNKNOWN_BRANDING_STATE,

  WAITING_FOR_SESSION,
  WAITING_FOR_VOICE_CONNECTION,
  WAITING_FOR_ANNOUNCEMENT_END,
  WAITING_FOR_TRANSFER,
  WAITING_FOR_CALL_END

} BRANDING_STATE;



class BrandingState : public OA_State
{
  public:
    void reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt );
    OA_INTERNAL_TASK_TYPE handleOther( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void connectVoice( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void startAnnouncement( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void stopAnnouncement( OA_AbstractContext* cxt );
    void transfer( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void endCall( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
};

class WaitingForSession : public BrandingState
{
  public:
    OA_UWORD getId();
    const char* getName();

    OA_STATUS getStatus();

    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt );
};

class WaitingForVoiceConnection: public BrandingState
{
  public:
    OA_UWORD getId();
    const char* getName();

    OA_STATUS getStatus();

    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt );
};
```

Figure 65 Example header file for branding call processing states (continued)

```
class WaitingForAnnouncementEnd: public BrandingState
{
  public:
    OA_UWORD getId();
    const char* getName();

    OA_STATUS getStatus();

    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );
    void reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt );
};

class WaitingForTransfer: public BrandingState
{
  public:
    OA_UWORD getId();
    const char* getName();

    OA_STATUS getStatus();

    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );

    // This state can use the reset() method provided by BrandingState
    // without modification; no need to override
};

class WaitingForCallEnd : public BrandingState
{
  public:
    OA_UWORD getId();
    const char* getName();

    OA_STATUS getStatus();

    OA_INTERNAL_TASK_TYPE process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt );

    // This state can use the reset() method provided by BrandingState
    // without modification; no need to override
};
```

The source file contains the implementations of each method of each class defined in the header file. Figure 66 through Figure (page 127 to page 141) show example code in the source file, brcallp.c.

Figure 66 Example source file for branding call processing states

```
#include "brcallp.h"

void BrandingState::reset( OA_INTERNAL_TASK_TYPE /*mtcTask*/, OA_AbstractContext* cxt )
{
  // Cancel timers for outstanding OAP requests
  OA_TimedEventList* timerList = cxt->getTimedEventList( );
  timerList->cancelAllRequests( OA_OAP_EVENT_TYPE );

  setNextStateId( WAITING_FOR_SESSION );
}

OA_INTERNAL_TASK_TYPE BrandingState::handleOther( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_OAP_EVENT_TYPE:
      //
      switch ( inEvent->getEventSubType( ) )
      {
        case OA_OPID_SESSION_BEGIN_INFORM:
          //
          //  The switch may send a Session Begin Inform at any time during
          //  an active call flow.  If this happens, the previous call is over
          //  and the Session Begin Inform should be handled like any new call.
          //  Clean up resources associated with the old call, go to the
          //  WaitingForSession state, and handle the new session as usual.
          //
          reset( OA_INT_TASK_RESET, cxt );
          setPositiveAssertion();
          break;

        case OA_OPID_CONNECTION_STATUS_INFORM:
        {
          //
          //  If hook status is onhook or released, caller has hung up.
          //  End the call.
          //
          OA_UWORD hookStatus = 0;
          OA_Operation* inOp = (OA_Operation*)inEvent->getEventData( );

          inOp->getField( &hookStatus, OA_FID_CONNECTION_EVENT, OA_DBID_CONNECTION_EVENT );

          if (( hookStatus == OA_ONHOOK_CONNECTION_EVENT) ||
              ( hookStatus == OA_RELEASED_CONNECTION_EVENT ) )
          {
            endCall( inEvent, cxt );
          }
         break;
          }
```

Figure 67 Example source file for branding call processing states (continued)

```
        case OA_OPID_CALL_END_INFORM:
          //
          //  The switch may send a Call End Inform at any time.
          //  The call is over; clean up resources and reset state machine.
          //
          reset( OA_INT_TASK_RESET, cxt );
          break;

        case OA_OPID_CALLP_REJECT:
          //  The switch may send a reject if it receives a message that
          //  violates OAP syntax or rules.  In call processing, this usually
          //  represents an error in the SN software that created the message.
          //  Log the error and end the call.
          endCall( inEvent, cxt );
          break;

        case OA_OPID_DTMF_DIGIT_DETECTED_INFORM:
          //
          //  The subscriber pressed a digit.
          //  Ignore the event.
          //
          break;

        default:
          //  A completely unexpected operation arrived at the SN.
          //  This could represent a failure by the SN developer to consider
          //  a possible call scenario, or a problem with how the SN is
          //  configured at the switch.
          //  Log the error and end the call.
          endCall( inEvent, cxt );
          break;
      }
        break;

    default:
      // Unexpected event type.
      endCall( inEvent, cxt );
      break;
  }
  return OA_INT_TASK_NULL;
}
```

Figure 68 Example source file for branding call processing states (continued)

```
void
BrandingState::connectVoice( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  OA_DWRD chnl = -1;
  BrandingContext* brCxt = (BrandingContext*)cxt;

  // Defensive programming: make sure no channels are currently active.
  //
  if ( brCxt->getVoiceChannel( chnl ) == OA_TRUE )
  {
    VOCfreeChannel( chnl );
    brCxt->clearVoiceChannel( );
  }

  // Allocate new voice channel and save it in the context.
  // If no channels available, transfer the call.
  //
  chnl = VOCallocChannel();
  if ( chnl == -1 )
  {
    transfer( inEvent, cxt );
    return;
  }
  brCxt->setVoiceChannel( chnl );

  // Request switch to connect voice to the new channel and
  // wait for the connection
  //
  OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
  OA_AbstractEvent* outEvent = OA_NULL;
  OA_RCODE RC = oapOpFactory->createRequestOapOpEvent( outEvent,
                                                       OA_OPID_VOICE_CONNECT_REQUEST,
                                                       OA_UNKNOWN_ADDR );

  if ( RC != OA_RC_SUCC )
  {
    OA_Log::swerr("CreateOAPOpError")
      << "Error received while creating oap reaquest.  RC = " << RC
      << "found in " << __FILE__ << " at line " << __LINE__
      << OA_Log::end;
    transfer( inEvent, cxt);
    return;
  }

  // Set channel id in operation
  OA_Operation* outOp = (OA_Operation*)outEvent->getEventData( );
  outOp->setField( chnl, OA_FID_VOICE_CHANNEL_ID, OA_DBID_VOICE_CHANNEL );

  cxt->queueOutEvent( outEvent, inEvent );
  setNextStateId( WAITING_FOR_VOICE_CONNECTION );
}
```

Figure 69 Example source file for branding call processing states (continued)

```
void BrandingState::startAnnouncement( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  BrandingContext* brCxt = (BrandingContext*)cxt;
  OA_DWRD chnl = -1;
  OA_DWRD annc = -1;
  OA_UWORD invokeId = 0;

  // Get the previously saved voice channel.
  // Defensive programming: if voice channel not set, transfer call
  //
  if ( brCxt->getVoiceChannel( chnl ) == OA_FALSE )
  {
    transfer( inEvent, cxt );
    return;
  }

  // Get announcement ID from context.
  // If not available, free the voice channel and transfer the call.
  //
  if ( brCxt->getAnnouncement( annc ) == OA_FALSE )
  {
    VOCfreeChannel( chnl );
    brCxt->clearVoiceChannel( );

    transfer( inEvent, cxt );
    return;
  }

  // Delete the in event here or it will be leaked.
  delete inEvent;

  // Play the branding announcement
  VOCstart( annc, chnl );

  // set timer event to time for end of announcement
  cxt->setWakeupEvent( VOCanncLength( annc ), invokeId );
  brCxt->setWakeupInvokeId( invokeId );

  setNextStateId( WAITING_FOR_ANNOUNCEMENT_END );
}

void BrandingState::stopAnnouncement( OA_AbstractContext* cxt )
{
  OA_DWRD chnl = -1;
  BrandingContext* brCxt = (BrandingContext*)cxt;

  // Use saved voice channel ID to stop announcement
  //
  if ( brCxt->getVoiceChannel( chnl ) == OA_TRUE )
  {
    VOCstop( chnl );
    VOCfreeChannel( chnl );
    brCxt->clearVoiceChannel( );
  }
}
```

Figure 70 Example source file for branding call processing states (continued)

```
voidBrandingState::transfer( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  OA_UWORD controlListId = 0;
  BrandingContext* brCxt = (BrandingContext*)cxt;
  OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
  OA_AbstractEvent* outEvent = OA_NULL;
  OA_Operation* outOp = OA_NULL;

  // Use configured control list ID to transfer call.  Control list ID
  // identifies the set of operators that will receive the transferred call.
  // If control list not set, end the call.
  //
  if ( brCxt->getControlList( controlListId ) == OA_FALSE )
  {
    endCall( inEvent, cxt );
    return;
  }

  OA_RCODE RC =
    oapOpFactory->createRequestOapOpEvent( outEvent, OA_OPID_TRANSFER_TO_CONTROL_LIST_REQUEST,
                                           OA_UNKNOWN_ADDR );

  if ( RC != OA_RC_SUCC )
  {
    OA_Log::swerr("CreateOAPOpError")
      << "Error received while creating oap reaquest.  RC = " << RC
      << "found in " << __FILE__ << " at line " << __LINE__
      << OA_Log::end;
    return;
  }

  // Set field info in request operation
  outOp = (OA_Operation*)outEvent->getEventData( );
  outOp->setField( controlListId,
                   OA_FID_CONTROL_LIST_ID,
                   OA_DBID_TRANSFER_CONTROL_LIST );

  // Tell switch to take down call if the subscriber goes on hook
  // while in queue for operator
  //
  outOp->setField( OA_TAKE_CALL_DOWN_CALL_IN_QUEUE_TAKEDOWN,
                   OA_FID_CALL_IN_QUEUE_TAKE_DOWN,
                   OA_DBID_TRANSFER_CONTROL_LIST );

  cxt->queueOutEvent( outEvent, inEvent );
  setNextStateId( WAITING_FOR_TRANSFER );
}
```

Figure 71 Example source file for branding call processing states (continued)

```
void BrandingState::endCall( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  OA_OapOperationFactory* oapOpFactory = cxt->getOapOperationFactory( );
  OA_AbstractEvent* outEvent = OA_NULL;

  reset( OA_INT_TASK_RESET, cxt );

  OA_RCODE RC = oapOpFactory->createRequestOapOpEvent( outEvent, OA_OPID_END_CALL_REQUEST,
                                                       OA_UNKNOWN_ADDR );

  if ( RC != OA_RC_SUCC )
  {
    OA_Log::swerr("CreateOAPOpError")
      << "Error received while creating oap reaquest.  RC = " << RC
      << "found in " << __FILE__ << " at line " << __LINE__
      << OA_Log::end;
    return;
  }
  cxt->queueOutEvent( outEvent, inEvent );
  setNextStateId( WAITING_FOR_CALL_END );
}

////////////////////////////////////////////////////////////////////////
// WaitingForSession
////////////////////////////////////////////////////////////////////////
OA_UWORD
WaitingForSession::getId()
{
  return WAITING_FOR_SESSION;
}

const char*
WaitingForSession::getName()
{
  return "Waiting For Session";
}

OA_STATUS
WaitingForSession::getStatus()
{
  return OA_STATUS_IDLE;
}
```

Figure 72 Example source file for branding call processing states (continued)

```
OA_INTERNAL_TASK_TYPE
WaitingForSession::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case ( OA_OAP_EVENT_TYPE ):
      //
      switch ( inEvent->getEventSubType( ))
      {
        case OA_OPID_SESSION_BEGIN_INFORM:
          //
          //  Begin servicing the new call by connecting voice
          //
            connectVoice( inEvent, cxt );
            break;

        default:
          //
          //  Not expecting any other kind of message.  The SN believes the
          //  call to be over, so extra messages are ignored.
          //
          break;
      }
      break;

    default:
      //
      handleOther( inEvent, cxt );
  }
  return OA_INT_TASK_NULL;
}

void
WaitingForSession::reset( OA_INTERNAL_TASK_TYPE, OA_AbstractContext* )
{
  // This is an idle period between calls, so there are no resources
  // to clean up.  This method will never be called by the API, since
  // the status is OA_STATUS_IDLE.
  //
  return;
}

////////////////////////////////////////////////////////////////////////
// WaitingForVoiceConnection
////////////////////////////////////////////////////////////////////////
OA_UWORD WaitingForVoiceConnection::getId()
{
  return WAITING_FOR_VOICE_CONNECTION;
}

const char* WaitingForVoiceConnection::getName()
{
  return "Waiting For Voice Connection";
}
```

Figure 73 Example source file for branding call processing states (continued)

```
OA_STATUS WaitingForVoiceConnection::getStatus()
{
  return OA_STATUS_ACTIVE;
}

OA_INTERNAL_TASK_TYPE
WaitingForVoiceConnection::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_OAP_EVENT_TYPE:
      //
      switch ( inEvent->getEventSubType( ))
      {
        case OA_OPID_VOICE_CONNECT_RETRES:
          // Voice channel is connected.  Start announcement
          startAnnouncement( inEvent, cxt );
          break;

        case OA_OPID_CALLP_ERROR_RESPONSE:
          // Error encountered.  Keep things simple and transfer the call.
          transfer( inEvent, cxt );
          break;

        default:
          handleOther( inEvent, cxt );
      }
      break;

    case OA_OAP_API_INTERNAL_EVENT_TYPE:
      //
      switch ( inEvent->getEventSubType( ))
      {
        case OA_INT_TASK_REQUEST_TIMEOUT:
          // Request timed out.  Keep things simple and transfer the call.
          transfer( inEvent, cxt );
          break;

        default:
          handleOther( inEvent, cxt );
      }
      break;

    default:
      handleOther( inEvent, cxt );
  }
  return OA_INT_TASK_NULL;
}
```

Figure 74 Example source file for branding call processing states (continued)

```
void WaitingForVoiceConnection::reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt )
{
  // Free voice channel.
  OA_DWRD chnl = -1;
  BrandingContext* brCxt = (BrandingContext*)cxt;

  if ( brCxt->getVoiceChannel( chnl ) == OA_TRUE )
  {
    VOCfreeChannel( chnl );
    brCxt->clearVoiceChannel( );
  }
  BrandingState::reset( mtcTask, cxt );
}

//////////////////////////////////////////////////////////////////
// WaitingForAnnouncementEnd
//////////////////////////////////////////////////////////////////
OA_UWORD WaitingForAnnouncementEnd::getId()
{
  return WAITING_FOR_ANNOUNCEMENT_END;
}

const char* WaitingForAnnouncementEnd::getName()
{
  return "Waiting For Ann. End";
}

OA_STATUS WaitingForAnnouncementEnd::getStatus()
{
  return OA_STATUS_ACTIVE;
}
```

Figure 75 Example source file for branding call processing states (continued)

```
OA_INTERNAL_TASK_TYPE
WaitingForAnnouncementEnd::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case OA_OAP_API_INTERNAL_EVENT_TYPE:
      switch ( inEvent->getEventSubType( ))
      {
        case OA_INT_TASK_WAKEUP:
          // Announcement timer expired.
          stopAnnouncement( cxt );
          transfer( OA_NULL, cxt );
          delete inEvent;
          break;
        default:
          handleOther( inEvent, cxt );
      }
      break;

    default:
      //
      handleOther( inEvent, cxt );
  }
  return OA_INT_TASK_NULL;
}

void
WaitingForAnnouncementEnd::reset( OA_INTERNAL_TASK_TYPE mtcTask, OA_AbstractContext* cxt )
{
  stopAnnouncement( cxt );

  // Free voice channel.
  OA_DWRD chnl = -1;
  BrandingContext* brCxt = (BrandingContext*)cxt;

  if ( brCxt->getVoiceChannel( chnl ) == OA_TRUE )
  {
    VOCfreeChannel( chnl );
    brCxt->clearVoiceChannel( );
  }

  BrandingState::reset( mtcTask, cxt );
}
```

Figure 76 Example source file for branding call processing states (continued)

```
//////////////////////////////////////////////////////////////////////
// WaitingForTransfer
//////////////////////////////////////////////////////////////////////
OA_UWORD WaitingForTransfer::getId()
{
  return WAITING_FOR_TRANSFER;
}

const char* WaitingForTransfer::getName()
{
  return "Waiting For Transfer";
}

OA_STATUS WaitingForTransfer::getStatus()
{
  return OA_STATUS_ACTIVE;
}
```

Figure 77 Example source file for branding call processing states (continued)

```
OA_INTERNAL_TASK_TYPE
WaitingForTransfer::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case ( OA_OAP_API_INTERNAL_EVENT_TYPE ):
      switch ( inEvent->getEventSubType( ))
      {
        case OA_INT_TASK_REQUEST_TIMEOUT:
          // Request time out.
           OA_Log::minorAlarm("RequestTimeout")
               << "Request timeout received while waiting for transfer ret res."
               << OA_Log::end;
          delete inEvent;
          endCall( OA_NULL, cxt );
          break;

        default:
          return handleOther( inEvent, cxt );
      }
      break;

    case ( OA_OAP_EVENT_TYPE ):
      switch ( inEvent->getEventSubType( ))
      {
        case OA_OPID_TRANSFER_TO_CONTROL_LIST_RETRES:
          //  Call transfer successfully.  Return to start state.
          setNextStateId( WAITING_FOR_SESSION );
          delete inEvent;
          break;

        case OA_OPID_CONNECTION_STATUS_INFORM:
          //  Change in hook status.
          //  The call is transfering, so ignore the event.
          break;

        default:
          return handleOther( inEvent, cxt );
      }
      break;

    default:
      return handleOther( inEvent, cxt );
  }
  return OA_INT_TASK_NULL;
}
```

Figure 78 Example source file for branding call processing states (continued)

```
//////////////////////////////////////////////////////////////////////
// WaitingForCallEnd
//////////////////////////////////////////////////////////////////////
OA_UWORD WaitingForCallEnd::getId()
{
  return WAITING_FOR_CALL_END;
}

const char* WaitingForCallEnd::getName()
{
  return "Waiting For Call End";
}

OA_STATUS WaitingForCallEnd::getStatus()
{
  return OA_STATUS_ACTIVE;
}
```

Figure 79 Example source file for branding call processing states (continued)

```
OA_INTERNAL_TASK_TYPE
WaitingForCallEnd::process( OA_AbstractEvent* inEvent, OA_AbstractContext* cxt )
{
  switch ( inEvent->getEventType( ) )
  {
    case ( OA_OAP_API_INTERNAL_EVENT_TYPE ):
      switch ( inEvent->getEventSubType( ))
      {
        case OA_INT_TASK_REQUEST_TIMEOUT:
          // Request time out, assume it was received by DMS.  Must delete
          // the event since it contains the original request and will be leaked.
          OA_Log::minorAlarm("RequestTimeout")
            << "Request timeout received while waiting for end call ret res."
            << OA_Log::end;
          setNextStateId( WAITING_FOR_SESSION );
          delete inEvent;
          break;

        default:
          return handleOther( inEvent, cxt );
      }
      break;

    case ( OA_OAP_EVENT_TYPE ):
      //
      switch ( inEvent->getEventSubType( ))
      {
        case OA_OPID_END_CALL_RETRES:
          //  Call ended successfully.
          //  Return to start state.
          setNextStateId( WAITING_FOR_SESSION );
          delete inEvent;
          break;

        case OA_OPID_CONNECTION_STATUS_INFORM:
          //  Change in hook status.
          //  The call is ending, so ignore the event.
          break;

        default:
          return handleOther( inEvent, cxt );
      }
      break;

    default:
      return handleOther( inEvent, cxt );
  }
  return OA_INT_TASK_NULL;
}
```

Step 3—Extending the API maintenance classes

The voice hardware for the sample branding application provides a function to test whether the voice hardware is ready for use. The branding application should not come into service until the voice hardware is ready, so the application must extend the provided API maintenance routines.

To extend the maintenance, this application creates subclasses of the OA_NodeBusy and OA_PoolBusy states. The application overrides the readyForService() method of the subclass of the OA_PoolBusy state so that the session pool only comes into service when the voice hardware is ready. It also overrides the diagnostic() method of both states to perform a diagnostic of the voice hardware when the SN receives a test request.

Figure 80 shows example code in the header file, brmtc.h.

Figure 80 Example header file for branding maintenance

```
#include "oa/nodemtc.h"
#include "oa/poolmtc.h"
#include "voiceapi.h"     // Header file with voice hardware API


class BrandingNodeBusy : public OA_NodeBusy
{
  public:
    OA_RCODE diagnostic( OA_AbstractContext* cxt,
                         char* &reportText );
};



class BrandingPoolBusy : public OA_PoolBusy
{
  public:
    OA_RCODE diagnostic( OA_AbstractContext* cxt,
                         char* &reportText );
    OA_RCODE readyForService( OA_AbstractContext* cxt,
                           char* &reportText );
};
```

Figure 81 shows example code in the source file, brmtc.c.

Figure 81 Example source file for branding maintenance

```
#include "brmtc.h"


OA_RCODE BrandingNodeBusy::diagnostic( OA_AbstractContext*,
                                char* &reportText )
{
  OA_RCODE RC;

  if ( VOCtest() == 1 )
  {
    reportText = "Voice OK";
    RC = OA_RC_SUCC;
  }
  else
  {
    reportText = "Voice test failed";
    RC = OA_RC_NOTAVAILABLE;
  }

  return RC;
}


OA_RCODE BrandingPoolBusy::diagnostic( OA_AbstractContext*,
                                char* &reportText )
{
  OA_RCODE RC;

  if ( VOCtest() == 1 )
  {
    reportText = "Voice OK";
    RC = OA_RC_SUCC;
  }
  else
  {
    reportText = "Voice test failed";
    RC = OA_RC_NOTAVAILABLE;
  }

  return RC;
}


OA_RCODE BrandingPoolBusy::readyForService( OA_AbstractContext*,
                                    char* &reportText )
{
  OA_RCODE RC;

  if ( VOCtest() == 1 )
  {
    reportText = "Ready for branding";
    RC = OA_RC_SUCC;
  }
  else
  {
    reportText = "Voice test failed";
    RC = OA_RC_NOTAVAILABLE;
  }

  return RC;
}
```

Step 4—Creating a subclass of the API framework

The buildCallpStates(), buildNodeMtcStates() and the buildPoolMtcStates() methods of the OA_Framework are overridden to incorperate the call processing and maintenance state subclasses specific to the branding application.

A subclass of the Context object is created to allow custimization of call context data. The buildSesnContext() method is overridden so that branding context objects, BrandingContext, are created for each session instead of the default Context object.

Finally, the init() method of the OA_Framework is overridden in order to initialize some branding specifiic attributes that have been added to the branding framework object.

Figure 82 shows example code in the header file, brframe.h.

Figure 82 Example header file for branding framework

```
#include "brcallp.h"      // Call processing states
#include "brmtc.h"        // Maintenance states

#include "oa/framewrk.h"


class BrandingFramework : public OA_Framework
{
  public:
    OA_RCODE buildCallpStates();
    OA_RCODE buildNodeMtcStates();
    OA_RCODE buildPoolMtcStates();

    OA_DWRD getNumberChannels( );

  protected:
    OA_RCODE init( OA_ConfigData* nodeData );
    OA_RCODE buildSesnContext( OA_AbstractContext* &cxt,
                               OA_ConfigData* nodeData,
                               OA_ConfigData* poolData,
                               OA_UWORD sesnId );

    OA_DWRD announcementId;
    OA_UWORD controlListId;
    OA_DWRD numberChannels;
```

Figure 83, Figure 85, and Figure 85show example code in the source file, brframe.c.

Figure 83 Example source file for framework

```
#include "brframe.h"

// User Data keys that must be in configuration file
// along with configurable data values, like this:
//
// BeginUserData AnnouncementId
//    AnnouncementId 12
//    ControlListId 5
//    NumberOfChannels 24
// EndUserData
//
const char* ANNOUNCEMENT_TAG = "AnnouncementId";
const char* CONTROL_LIST_ID_TAG = "ControlListId";
const char* NUMBER_CHANNELS_TAG = "NumberChannels";

OA_RCODE BrandingFramework::buildCallpStates()
{
  addCallpState( new WaitingForSession() );
  addCallpState( new WaitingForVoiceConnection() );
  addCallpState( new WaitingForAnnouncementEnd() );
  addCallpState( new WaitingForTransfer() );
  addCallpState( new WaitingForCallEnd() );

  return OA_RC_SUCC;
}

OA_RCODE BrandingFramework::buildNodeMtcStates()
{
  addNodeMtcState( new BrandingNodeBusy() );

  // Build other node maintenance states
  //
  return OA_Framework::buildNodeMtcStates();
}

OA_RCODE
BrandingFramework::buildPoolMtcStates()
{
  addPoolMtcState( new BrandingPoolBusy() );

  // Build other session pool maintenance states
  //
  return OA_Framework::buildPoolMtcStates();
}
```

Figure 84 Example source file for framework (continued)

```
OA_RCODE BrandingFramework::init( OA_ConfigData* nodeData )
{
  OA_UDWRD blockNum = 0;
  OA_ConfigData* userData = OA_NULL;
  OA_UDWRD temp = 0;

  announcementId = 0;
  controlListId = 0;
  numberChannels = 0;

  // Attempt to obtain block of user data from node config
  OA_RCODE RC = nodeData->getFirstBlock( OA_BEGINUSERDATA_BEGIN, blockNum, userData );
  if ( RC == OA_RC_SUCC )
  {
    // Retrieve branding announcement index and control list id from
    // config file.
    if ( userData->isSet( ANNOUNCEMENT_TAG ) == OA_TRUE )
    {
      userData->getValueField( ANNOUNCEMENT_TAG, 1, temp );
        announcementId = (OA_DWRD)temp;
    }
    else
      OA_Log::minorAlarm("MissingAnnouncementId")
        << "Config value " << ANNOUNCEMENT_TAG
        << " not found.  Defaulting to announcement id 0."
        << OA_Log::end;

    if ( userData->isSet( CONTROL_LIST_ID_TAG ) == OA_TRUE )
    {
      userData->getValueField( CONTROL_LIST_ID_TAG, 1, temp );
        controlListId = (OA_UWORD)temp;
    }
    else
      OA_Log::minorAlarm("MissingCtrlListId")
        << "Config value " << CONTROL_LIST_ID_TAG
        << " not found.  Defaulting to control list id 0."
        << OA_Log::end;

    if ( userData->isSet( NUMBER_CHANNELS_TAG ) == OA_TRUE )
    {
      userData->getValueField( NUMBER_CHANNELS_TAG, 1, temp );
        numberChannels = (OA_UWORD)temp;
    }
    else
      OA_Log::minorAlarm("NumberChannels")
        << "Config value " << NUMBER_CHANNELS_TAG
        << " not found.  Defaulting to 0 channels."
        << OA_Log::end;
  }
  return OA_Framework::init( nodeData );
}
```

Figure 85 Example source file for framework (continued)

```
OA_RCODE
BrandingFramework::buildSesnContext( OA_AbstractContext* &cxt,
                            OA_ConfigData* nodeData,
                            OA_ConfigData* poolData,
                            OA_UWORD sesnId )
{
  cxt = new BrandingContext( evFactories, timer, OA_NULL, nodeData, outbox,
                        mtcQueue, &externalNodes, addrBook,
                        0, /*tlsindex*/
                        nodeData->getWordValue( OA_BEGINNODE_BEGIN ),
                        poolData->getWordValue( OA_BEGINPOOL_BEGIN ),
                        0, /* max sessions, meaningless here */
                        sesnId,
                        nodeOAPMultiOMList,
                        poolOAPMultiOMList );

  // Set the converted OM dump time in the context so that future om display events
  // can be created with the dump time specified in the config file.
  cxt->setOMDumptime(conv_om_dump_time);

  BrandingContext* brCxt = (BrandingContext*)cxt;
  brCxt->setControlList( controlListId );
  brCxt->setAnnouncement( announcementId );
  return OA_RC_SUCC;
}

OA_DWRD
BrandingFramework::getNumberChannels( )
{
  return numberChannels;
}
```

Step 5—Implementing the standalone application

Developers can use the subclass of the OA_Framework in the application's main() function. Figure 86 shows an example of implementing the sample standalone branding application.

Figure 86 Implementing the main() function in the branding application

```
#include "brframe.h"
#include "voiceapi.h"

OA_AbstractFramework* branding;
void initializeVoice();

int main( int argc, char** argv )
{
  int exitcode = 1;
  OA_RCODE RC;

  branding = new BrandingFramework();

  if ( argc == 1 )
    RC = branding->configure( "branding.cfg" );
  else
    RC = branding->configure( argc, argv );

  if ( RC == OA_RC_SUCC )
  {
    VOCinit( branding->getNumberChannels( ) );
    exitcode = branding->go();
  }

  delete branding;

  exit( exitcode );
}
```

Chapter 8: Additional API components

This chapter provides details on additional components of the OSSAIN API that are contained in the base software layer. The discussion focuses on the following areas:

Note: For details on the configuration utility, refer to Chapter 9: "Configuration and administration."

Log utility

Logs provide feedback to the maintainer of an OSSAIN API application. Various events can cause a log to be generated, such as service interruptions, unexpected operating conditions, API and user application software bugs, and configuration errors.

Using logs, application developers can trace message flows, state transitions, and message contents for debugging purposes. With the log utility, developers can create logs that are specific to the application.

Note: The log utility should not use any other component of the base software layer except standard type definitions and environment settings. This allows other API software components to use the log utility.

The API generates service node (SN) logs and it provides a mechanism to generate custom logs and alarms at the DMS switch. This section describes the types of SN logs defined by the API, and it discusses the interfaces for generating logs and alarms at the SN and at the switch.

SN log types

The API defines six log types. SN applications can generate these types or types that are defined by the developer.

Alarm logs

Alarm logs indicate problems that affect the operating status of the SN. The level of an alarm indicates its severity—critical, major, or minor. Levels are cumulative such that if major alarms are enabled, so are critical alarms. And if minor alarms are enabled, so are major and critical alarms.

Critical alarms indicate a complete loss of service for a node or session pool, or problems that could cause a complete outage. *Major* alarms indicate a partial loss of service, or conditions that adversely affect the operation of a node or session pool, but that may not necessarily cause a loss of service. *Minor* alarms indicate small and localized errors, or conditions that could potentially affect the operation of a node or session pool.

Nodes in a live switch should *always* have alarm logs enabled (with the possible exception of minor alarms).

Debug logs

Debug logs contain information used to debug the OSSAIN API or an application. The level of debug logs can be set to brief or detailed. Levels are cumulative such that if detailed logs are enabled, so are brief logs.

Nodes in a live switch should *never* have debug logs enabled. To prevent their use in a live node, any debug logs in the API that are left in a production release should be enclosed in a conditional compilation directive, such as the following:

```
#ifdef OA_DEBUG
// Debug log here
#endif // OA_DEBUG
```

Report logs

Report logs show information and statistics collected by the node, such as operational measurements. The level of report logs can be set to brief or detailed. Levels are cumulative such that if detailed logs are enabled, so are brief logs.

Nodes in a live switch can have report logs enabled if desired. The frequency at which report logs are displayed is governed by the OMDumpTime parameter in the application configuration file.

Status logs

Status logs report the current operating status of a node, session pool, or session. The level of status logs can be set to brief or detailed. Levels are cumulative such that if detailed logs are enabled, so are brief logs.

Status logs report SN activation, entry into service, shutdown, and session pool entry into service.

Nodes in a live switch can have status logs enabled if desired.

Swerr logs

Swerr logs report a software error, either in the API or in the user application. Swerr logs do not have a level to set.

The API generates swerr logs to indicate an unexpected code path, an illegal parameter to a function call, some other error in code, or a fundamental flaw in the operating environment.

Note: Invalid external input or messages should cause an alarm log (not a swerr log) to be generated.

Nodes in a live switch should *always* have swerr logs enabled. Swerr logs appear in call processing only if there is a software error—a condition that is potentially hazardous to the operation of the node.

Trace logs

Trace logs show all messaging and state machine activity, and are useful for debugging. The level of trace logs can be set to brief or detailed. Levels are cumulative such that if detailed logs are enabled, so are brief logs.

The API generates trace logs for all incoming and outgoing messages, and for all state machine transitions. Brief message logs show only the source/ destination address, byte count, and bytestream in hexadecimal format; detailed logs show a high-level parsed representation of the message.

Nodes in a live switch should *never* have trace logs enabled.

SN log interface

The log utility should be used to generate text output from the API whenever possible. This provides users a consistent interface for display of textual information and nearly complete control of API output through the configuration file.

Note: For details on the configuration file, refer to Chapter 9: "Configuration and administration."

Generating logs

To generate a log, the log utility requires the following information:

- the log type (such as alarm)
- the log level (such as major, or detailed)
- the unique log ID
- the log text

Users can detect whether the log type and level are enabled before choosing to generate a log.

Log output

The API provides mechanisms for logs to be displayed on the screen console or recorded in a file. Console output is sent through the standard output or standard error streams.

Multiple files can be used for recording logs. Each file optionally can be archived so that a prescribed maximum number of logs are recorded to a single file. Once it is full, the file is closed and renamed as an archive file, and a new file with the original name is opened. Users can configure the maximum number of logs per file and maximum number of archives per file.

Each log type is associated with one or more log output devices. Log levels for a single log type are set separately for each output device. Users can define their own output device by providing a routine to handle output directly. In this way, users can generate API logs using their own log tools.

Buffering logs

The user can buffer multiple logs before output. Buffering can significantly improve real-time performance, but there can be a delay between a logged event and actual log output.

Buffers are flushed when the node shuts down gracefully, but all buffered logs are lost during abnormal process termination. Applications can choose to manually flush buffered logs at any time.

One buffer is associated with each log output device (for example, one for standard output, one for standard error, one for each file). So multiple log types can be associated with a single buffer.

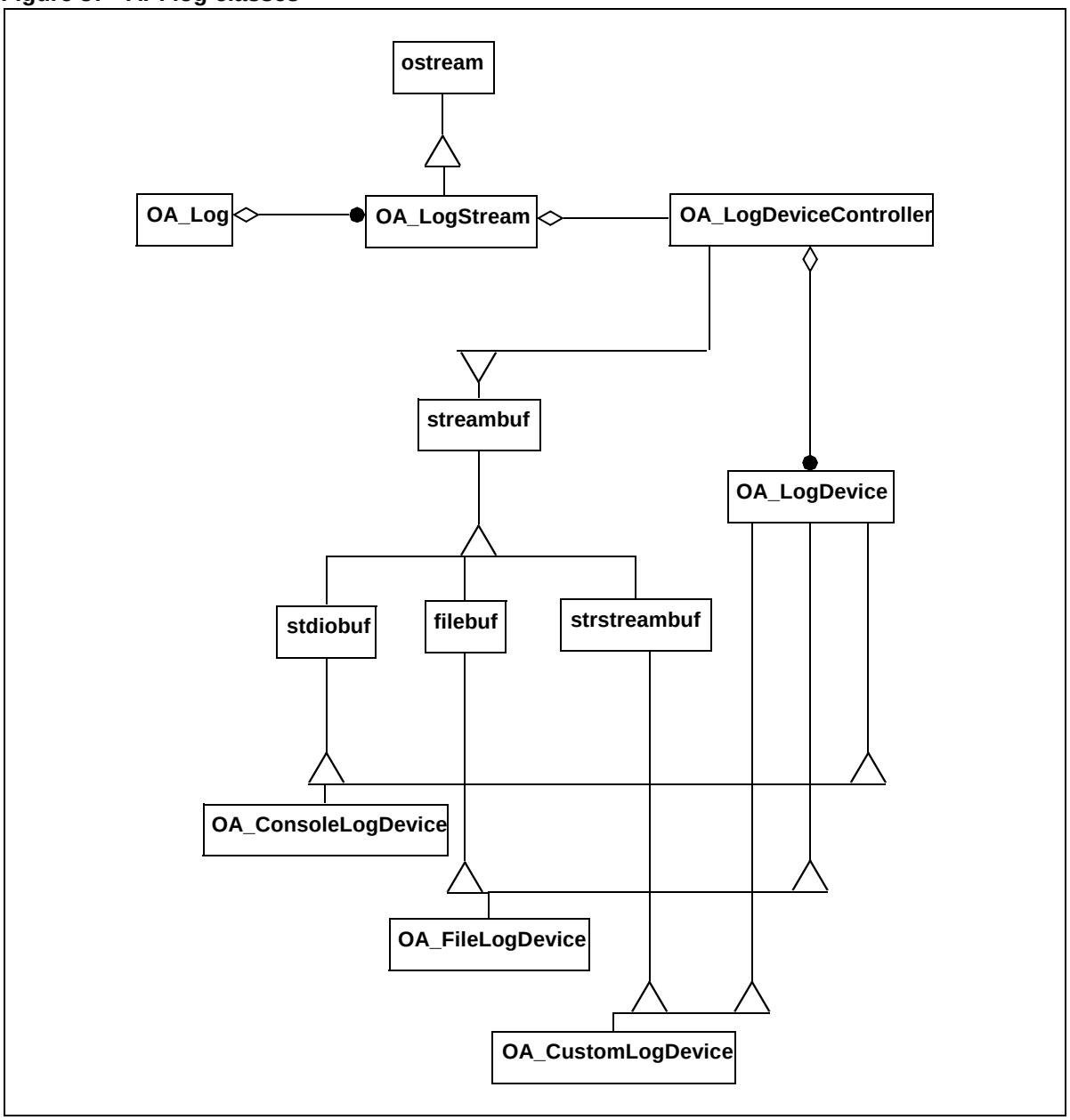
The following list describes buffering for each log type:

- Swerr and alarm logs should probably not be buffered, because they indicate errors that may be time-dependent and they may offer clues that would be lost if buffered during a fatal error.
- Because status logs typically are useful as screen updates for application maintainers, buffering them may cause confusion due to the time delay of when an action is performed and when a status log is generated.
- Debug and trace logs can appear frequently when activated, so buffering them can significantly improve execution time. Any application-specific logs that occur frequently (once or more each call) should be buffered.

Overview of log classes

This section provides a brief overview of the API log classes. Figure 87 shows the relationship of the API log classes and subclasses.

Figure 87 API log classes



OA_Log class

This base class is responsible for initializing and cleaning up the entire log utility. Only one OA_Log instance is needed per application.

OA_LogStream class

Objects of this class are the primary interface for log utility users. One OA_LogStream object represents a level of output for a log type. For example, critical alarms, detailed trace logs, or all swerr logs.

OA_LogStream is a subclass of the ostream class provided in the standard C++ library, so its heavily overloaded << operator is inherited. All manipulators defined for ostream are available for use with OA_LogStream, and any class that grants friend privileges to the ostream << operator can be printed with OA_LogStream.

OA_LogDeviceController class

Each object of class ostream contains a streambuf object to mediate output to a physical device. OA_LogStream objects use objects of class OA_LogDeviceController, a subclass of streambuf, to mediate output.

OA_LogDevice class

Each OA_LogDevice object handles all output to a specific destination (standard output, standard error, or a single file). Because multiple output destinations can be associated with a single log level (for example, critical alarms sent to standard error and a file), multiple OA_LogDevice objects can be referenced by a single OA_LogDeviceController object.

Likewise, because multiple log levels can be associated with a single output destination (for example, alarms and swerrs sent to one file), multiple OA_LogDeviceController objects can have references to a single OA_LogDevice object.

Operational measurements

Operational measurements (OM) are counters that are incremented each time a measured event takes place. The OM counters can be displayed by the application during runtime through the log system.

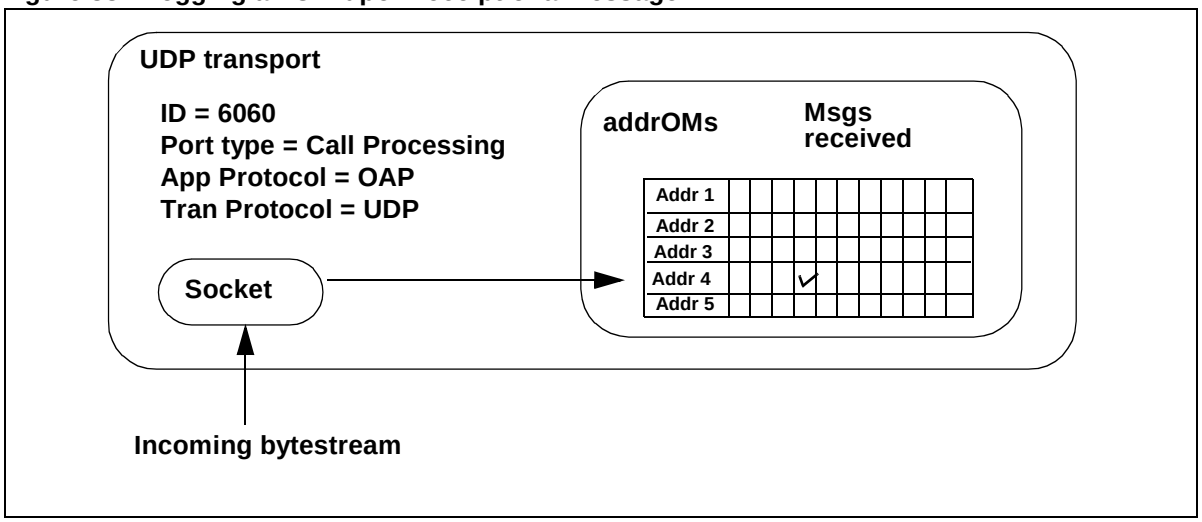
OMs are organized into OM groups and lists of OM groups. An OM group can be created to track events for an object instance in an application, and a list of such groups can track similar statistics for groups of objects.

Figure 88 illustrates the use of OMs in the communication classes of the API. In the example, the addrOMs object is a list of OM groups. Each row in the table represents and OM group object.

Note: The communication classes are discussed in detail beginning on page 163.

In the figure, the transport object receives a message on its contained socket. The transport changes the incoming IP address in the message to an address index, then uses that index to look into an array of OM groups titled addrOMs. It increments the Messages Received OM for the address from which the message was received. In this way the transport can keep track of the number of messages received, as well as other measurements such as communication errors, for each external address it communicates with.

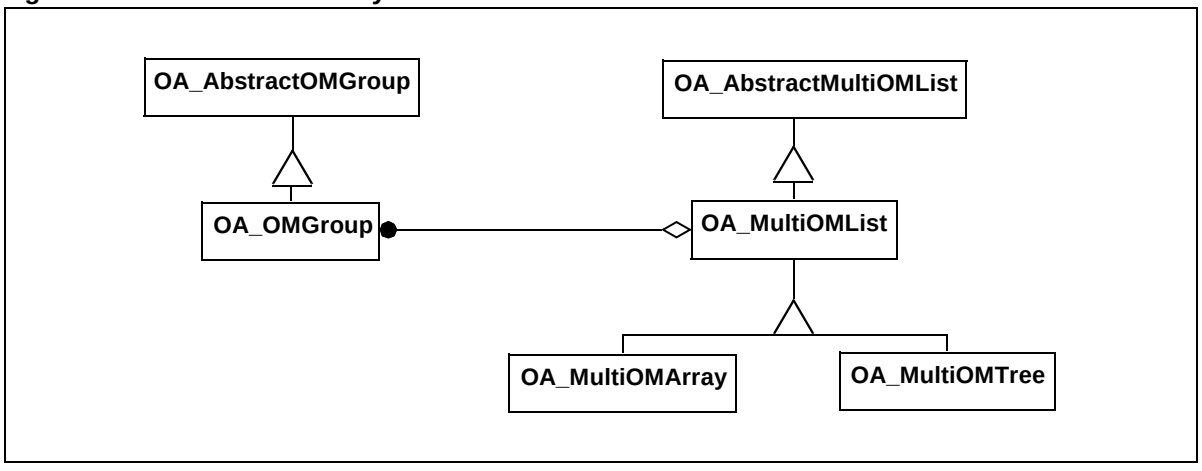
Figure 88 Pegging an OM upon receipt of a message



OM interfaces

Figure 89 shows the class hierarchy for OMs.

Figure 89 OM class hierarchy



The OA_OMGroup class contains an array of long integers where counts for the named registers are kept. The OA_AbstractOMGroup interface contains routines for incrementing an OM register in the array, checking for a zero count, resetting the count, and displaying the registers and their titles. The routines keep track of whether overflow or underflow has occurred and will generate error logs in these cases. The class overrides operator[] in order to provide convenient indexing into the array.

The OA_MultiOMList class allows the application to add OM groups to the list, delete them, and traverse the list. It also overrides operator[] to allow access to an OM group.

Figure 90 shows an example of the transport accessing its OMs. The code fragment shows the transport incrementing an OM for the address it is sending to as well as its own address, for which it keeps an OM group called portOM.

Figure 90 Example pegging of OM

```
RC = sendBuffer( (void*)msg->getBuffer(), bufSize, addr );

if ( RC != OA_RC_SUCC )
{
  if ( RC == OA_RC_SYSERR )
    OA_Log::majorAlarm("SocketSendError")
      << "Error sending message from " << this << " to " << addr
      << "\nReason: " << socketErrorText() << OA_Log::end;

  portOM->peg( OA_MESSAGE_OM_SEND_FAIL );
  (*addrOMs)[ destIndex ].peg( OA_MESSAGE_OM_SEND_FAIL );
  //..... om group ..... om within group .....
  return RC;
```

List classes

Many of the classes in the OSSAIN API use list data structures defined in the base layer. The lists structures allow an application to write object containers without having to implement common data structures to hold the objects.

A description of the lists structures follows:

- Numeric-key lists

  Lists keyed by a numeric key include a binary tree, a single-linked list, and a fixed-size array.

- Comparable-key lists

  The list classes also include lists keyed by instances of an abstract comparable class. This class allows the user to store pointers to objects using a variety of keys. As long as the key adheres to the OA_AbstractComparable interface, it can be used to store objects in a binary tree or a single-linked list.

- No-key lists

  No-key lists include a fixed queue, fixed stack, and free queue (FIFO queue), which store data without keys ordered.

- Hash tables

  A hash table that stores data by string key, and a subclass that stores data by the first character of a string are also available.

All the list classes have the following characteristics:

- The lists all store void pointers to object instances, allowing them to store any kind of object. The user must cast the stored instance to the correct object upon retrieval from a structure.
- The user is responsible for allocating and deleting store associated with elements stored in lists.
- Interfaces to lists are consistent and are kept in three abstract superclasses corresponding to the numeric-key, comparable-key, and no-key categories of lists. The consistency of the interface makes it easy to change data structure choices during coding.

Numeric-key lists

Figure 91 shows numeric-key lists descended from the class OA_AbstractNumericKeyList.

Figure 91 Numeric-key lists class hierarchy

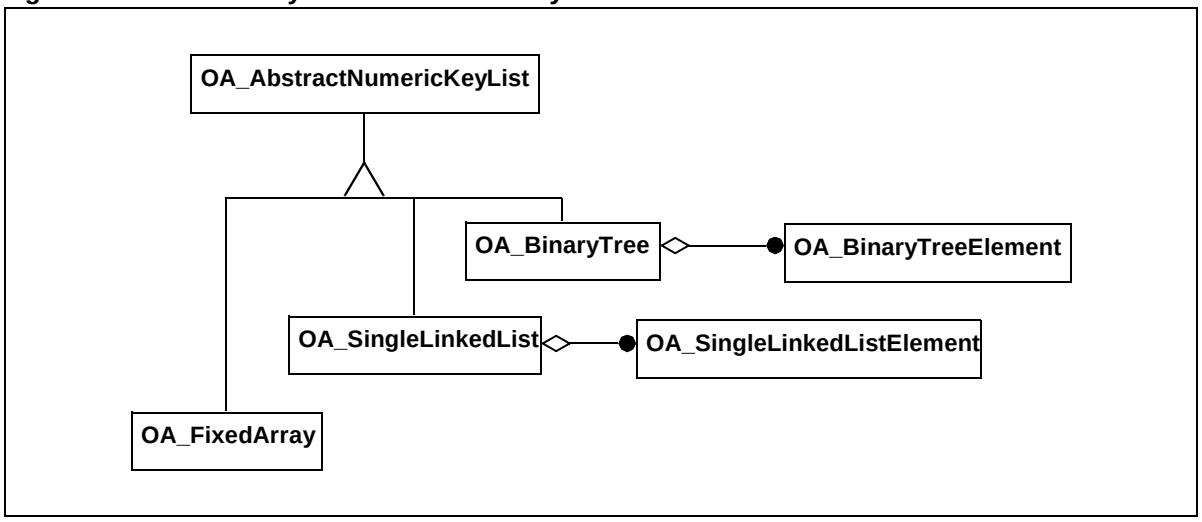


Table 10 shows the interface to which numeric-key classes adhere.

Table 10 OA_AbstractNumericKeyList **interface**

Methods (all are abstract virtual in abstract class)	**Description**
OA_RCODE add(OA_UDWRD key, void* elem)	Adds element to list using provided key value.
void* remove(OA_UDWRD key, OA_RCODE &RC)	Removes pointer from list but does not delete object reference by pointer
void* find(OA_UDWRD key, OA_RCODE &RC)	If RC is set to OA_RC_SUCC, key has been found and method returns void pointer to element. If not, method returns OA_NULL.
void* first(OA_UDWRD &key, OA_RCODE &RC)	Updates key with value of first element's key
void* next(OA_UDWRD &key, OA_RCODE &RC)	Key argument must have value for which you wish to find the next element; key is updated with value of found element's key
OA_BOOL isEmpty ( )	Returns OA_TRUE if no elements in list. Returns OA_FALSE if one or more elements in list.
OA_UDWRD size ( )	Returns number of elements in list.
~OA_AbstractNumericKeyList ( )	Virtual destructor, overridden by subclasses

Figure 92 shows an example of the use of the numeric-key list structure. The example code is taken from a validation routine that begins by recording the numbers of each type of datablock that are found in a given operation.

The routine creates a new binary tree. It calls find using as a key the datablock identifier, which happens to be a 16-bit integer. If a record for the datablock identifier cannot be found in the list, it adds a new record to the list.

Figure 92 Example for use of numeric-key binary tree

```
OA_RCODE
OA_ospSpecVersion::validateOpStructure( OA_ospAbstOper* OpObject,
                    OA_DBID* dbidsFound, OA_UWORD dbidCount )
{
...
// Define a new binary tree
OA_BinaryTree  *dbsFound = new OA_BinaryTree();
...
  for( Index = 0; Index < NumOfDBInfo; Index++ )
    {
    // Use the protocol id as a key to attempt to find an
    // existing record of the datablock having been found
    aDbRec = (OA_dbFoundRec *) dbsFound->find
          ( ListOfDBInfo[Index].dblk.protocolID(), TMP_RC );

    if ( TMP_RC != OA_RC_NOTFOUND )
      {
      // Already a record for this set or part of this set based
      // on protocol ID.
      //
      continue;
      }

    // Not found, so add a new record to the list with appropriate
    // info for the datablock found
    aDbRec = new OA_dbFoundRec( ListOfDBInfo[Index].
                        dblk.protocolID() );
    aDbRec->rule = ListOfDBInfo[Index].rule;
    aDbRec->min  = ListOfDBInfo[Index].minNumber;
    aDbRec->max  = ListOfDBInfo[Index].maxNumber;
    aDbRec->setid =  ListOfDBInfo[Index].setId;
    dbsFound->add( ListOfDBInfo[Index].dblk.protocolID(), aDbRec );
    }  // end for
...
}
```

Comparable-key lists

Figure 93 shows comparable-key lists descended from the class OA_AbstractComparableKeyList.

Figure 93 Comparable-key lists class hierarchy

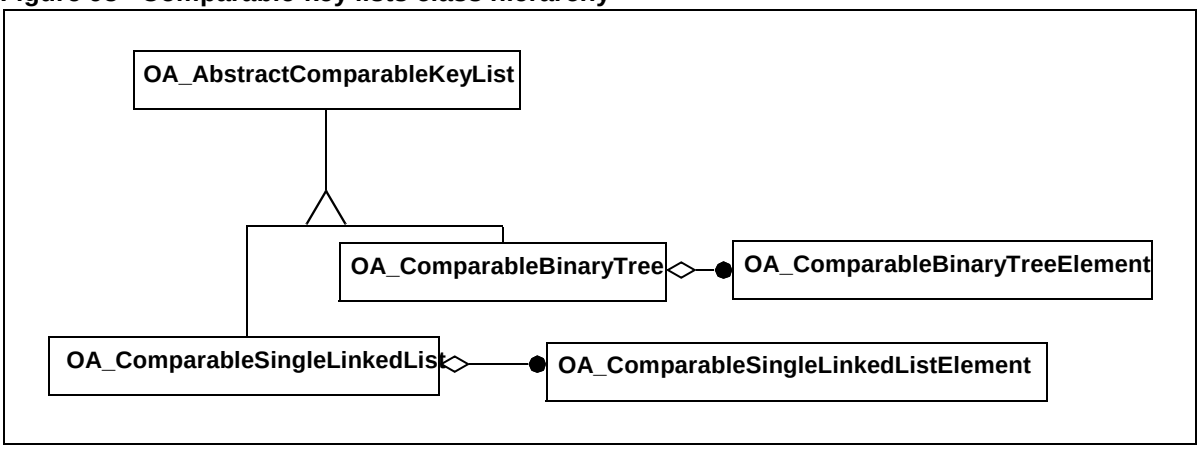


Table 11 shows the interface to which comparable-key classes adhere.

Table 11 OA_AbstractComparableKeyList **interface**

Methods (all are abstract virtual in abstract class)	**Description**
OA_RCODE add(OA_AbstractComparable* key, void* elem)	Adds element to list using provided key value.
void* remove(OA_AbstractComparable* key, OA_RCODE &RC)	Removes pointer from list but does not delete object reference by pointer
void* find(OA_AbstractComparable* key, OA_RCODE &RC)	If RC is set to OA_RC_SUCC, key has been found and method returns void pointer to element. If not, method returns OA_NULL.
void* first(OA_AbstractComparable* &key, OA_RCODE &RC)	Updates key with value of first element's key
void* next(OA_AbstractComparable* &key, OA_RCODE &RC)	Key argument must have value for which you wish to find the next element; key is updated with value of found element's key

Table 11 OA_AbstractComparableKeyList **interface**

Methods (all are abstract virtual in abstract class)	Description
OA_BOOL isEmpty ( )	Returns OA_TRUE if no elements in list. Returns OA_FALSE if one or more elements in list.
OA_UDWRD size ( )	Returns number of elements in list.
~OA_AbstractComparableKeyList ( )	Virtual destructor, overridden by subclasses

No-key lists

No-key lists include a fixed-size stack, a fixed-size FIFO queue, and a FIFO queue, which dynamically allocate store for pointers as they grow. Figure 94 shows no-key lists descended from the class OA_AbstractNoKeyList.

Figure 94 No-key lists class hierarchy

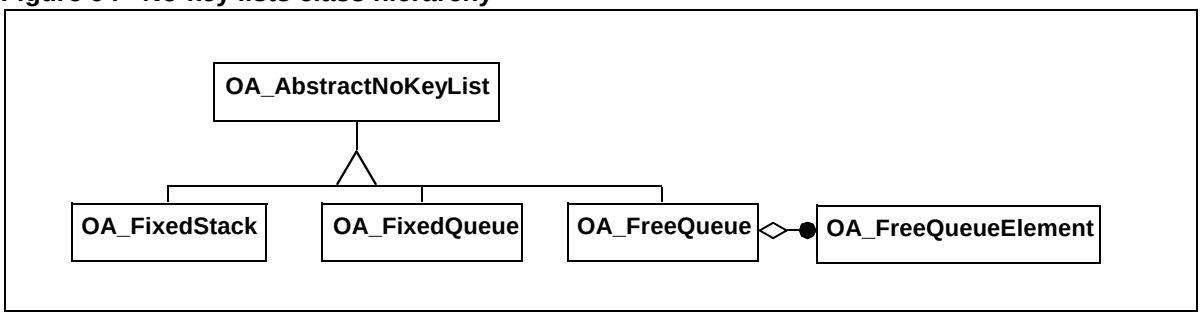


Table 12 shows the interface to which no-key classes adhere.

Table 12 OA_AbstractNoKeyList **interface**

Methods (all are abstract virtual in abstract class)	Description
OA_RCODE add(void* elem)	Adds element to list.
void* remove(OA_RCODE &RC)	Removes pointer from list but does not delete object reference by pointer
OA_BOOL isEmpty ( )	Returns OA_TRUE if no elements in list. Returns OA_FALSE if one or more elements in list.
OA_UDWRD size ( )	Returns number of elements in list.
~OA_AbstractNoKeyList ( )	Virtual destructor, overridden by subclasses

Hash tables

Two hash table classes are provided with the OSSAIN API. The first, OA_Hashtable, stores pointers to elements by hashing on a char* key. The second, OA_AlphaHashtable, stores data in 26 hash buckets based on the first letter of a string key.

The OA_Hashtable class has the following two parameters to its constructor:

- OA_BOOL caseSensitive determines whether the hash function considers case in its string hashing. The default value is OA_FALSE.
- OA_UWORD prime is the number of hash buckets to create. The default value is 23; a prime number is required. The greater the number of buckets, the faster the list access is for a large list, but the more memory the list uses.

Figure 95 shows the hash table class hierarchy.

Note: The OA_alphaHashTable subclass is identical to the OA_Hashtable class except that it overrides the hash function to hash on the first letter of the key.

Figure 95 Hash tables class hierarchy

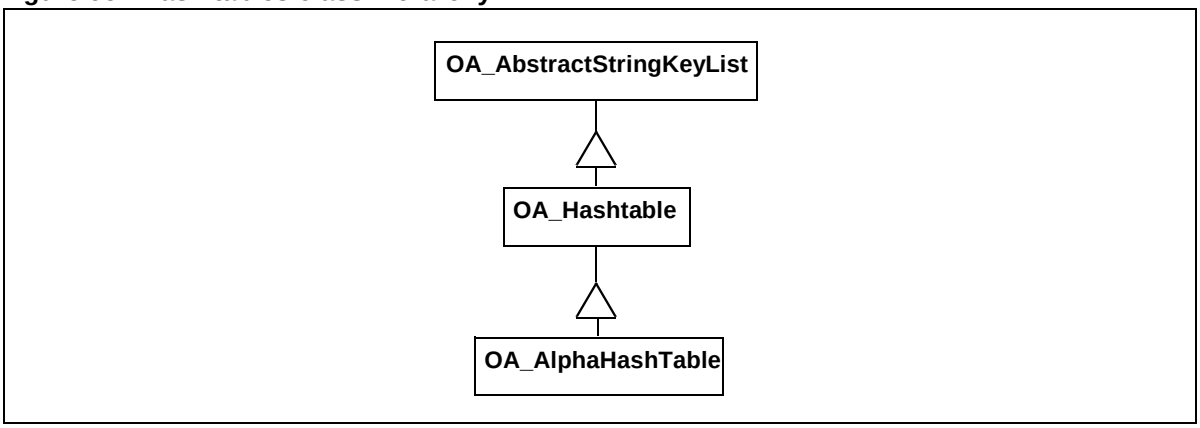


Table 13 shows the interface to which hash table classes adhere.

Table 13 OA_AbstractStringKeyList **interface**

Methods (all are abstract virtual in abstract class)	Description
OA_RCODE add(const char* key, void* elem)	Adds element to list using provided key value.
void* remove(const char* key, OA_RCODE &RC)	Removes pointer from list but does not delete object reference by pointer
void* find(const char* key, OA_RCODE &RC)	If RC is set to OA_RC_SUCC, key has been found and method returns void pointer to element. If not, method returns OA_NULL.

Table 13 OA_AbstractStringKeyList **interface**

Methods (all are abstract virtual in abstract class)	Description
void* first(const char* &key, OA_RCODE &RC)	Updates key with value of first element's key
void* next(const char* &key, OA_RCODE &RC)	Key argument must have value for which you wish to find the next element; key is updated with value of found element's key
OA_BOOL isEmpty ( )	Returns OA_TRUE if no elements in list. Returns OA_FALSE if one or more elements in list.
OA_UDWRD size ( )	Returns number of elements in list.
~OA_AbstractStringKeyList ( )	Virtual destructor, overridden by subclasses

Communication classes

The OSSAIN API base layer includes a set of communication classes that are dedicated to sending and receiving messages over the LAN or WAN to which the SN is connected. The communication classes have the following three other purposes:

- They abstract the idea of a bytestream into a class that knows its source and destination addresses.
- They include classes to store addresses for external communication peers in an address book.
- They wrap the transport layer implementation (such as sockets) in classes that hide the details of sending and receiving datagrams.

Communication classes are used by the NI classes to communicate with OSSAIN peer nodes. For example, the OA_proxy class contains a list of OA_AbstractTransport objects that it uses to send and receive operations. The transport class hierarchy includes specialized classes for UDP and TCP messaging.

Messages

The OA_AbstractMessage class sets up the responsibilities of an object that is to be sent or received through the communication software of the OSSAIN API.

The message must store integer representations of its source and destination addresses, called address indices. An address index is by convention either an index into the node's list of external addresses, or a port number.

The address index is used in conjunction with the node address book to provide a convenient representation of an external address. The NI classes and any customized subclasses can send to a destination address index. The communication software translates the destination index into an IP/port pair before sending.

A message object is an instance of the OA_Message class, which is the only subclass of OA_AbstractMessage. OA_Message provides a straightforward implementation of the abstract interface.

Address book and addresses

The address book classes hold data for external nodes that can be mapped to and from an address index. Classes OA_AbstractAddressBook and OA_AddressBook manage lists of addresses.

The node infrastructure adds addresses to the address book when it reads user datafill upon program startup. Addresses are embodied in the OA_AbstractAddress and a subclass, OA_IPAddress.

Transports

A transport object contains the implementation of a way to send and receive messages over the OSSAIN network. This means that each transport object contains a single socket instance, and its methods concern the management of the socket and the sending or receiving of messages over the socket.

The OSSAIN API transport includes the following classes:

- an abstract interface, OA_AbstractTransport
- a generic implementation, OA_Transport
- an IP-specific transport, OA_IPTransport
- three subclasses:
  - OA_TCPTransport
  - OA_TCPFixedLengthTransport
  - OA_UDPTransport

The three subclasses include protocol-specific routines for sending and receiving. Most of the socket management functions are independent of the protocol used, so they are located in OA_IPTransport or OA_Transport.

Lists of transports are support by the OA_AbstractTransportList and OA_TransportList classes.

OA_TCPFixedLengthTransport class

The OA_TCPFixedLengthTransport class contains a TCP socket. However, instead of receiving a continuous stream of data, it chops the data up into equal-length packets. The mechanism takes advantage of the reliability of the TCP, while allowing the receiving of fixed-length OAP messages.

The fixed length TCP transport is used in MIS applications where the DMS switch streams a series of event datablocks to the SN using OAP. The event datablocks are stored in a fixed-length operation, which has any unused space at the end padded with 0xff bytes.

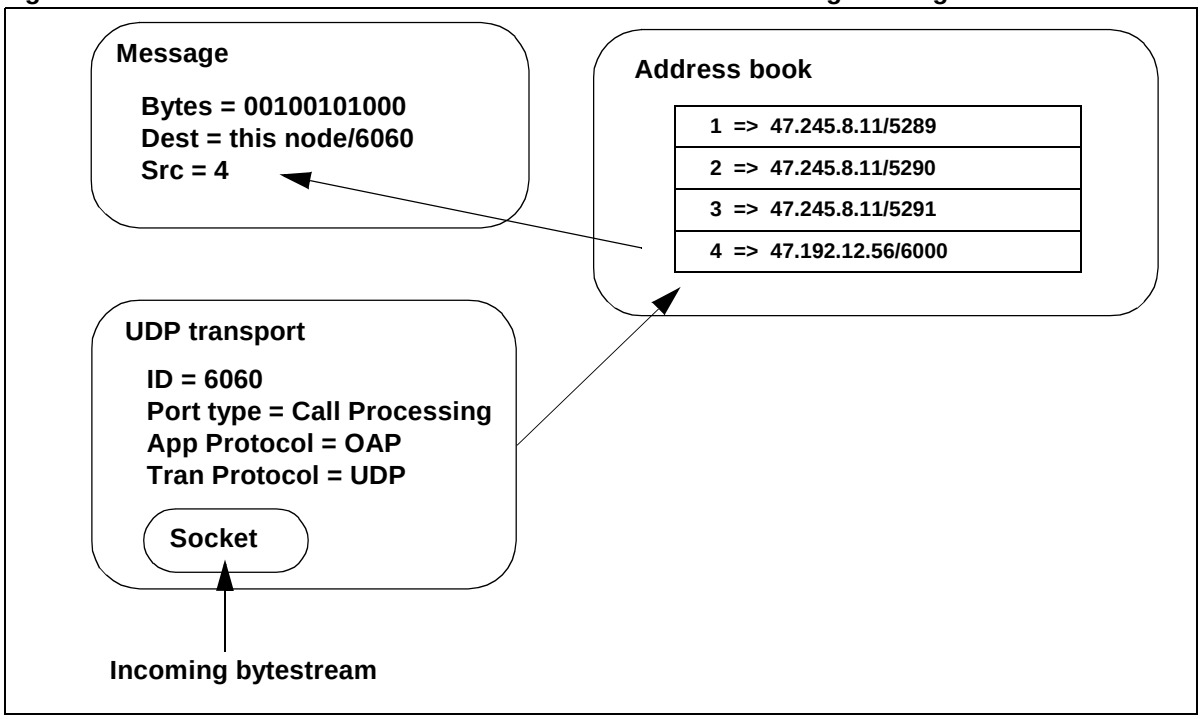
Sending and receiving over transport classes

In the NI layer and any application classes, the code knows about only the destination address indices that it needs to use. It also knows its transport ID number, which is an index into a list of transports held by the NI. Its transport ID is in fact the socket number that it uses to send on.

Figure 96 shows the work of the communication classes. In the example, an incoming bytestream is wrapped in a message object. The transport object that receives the message does a reverse lookup of IP and port number of the source of the message to an address index. It places the source address index in the message.

If the recipient of the message replies to it, it can simply copy the source address index of the received message into the destination address index field of the message it is sending. The mapping of address index to IP address/port occurs upon sending in the outgoing direction.

Figure 96 Determination of source address index for an incoming message



Parser utility

The parser utility allows an OSSAIN API component or application to read input and set execution parameters during runtime. For example, the API communication classes rely on the parser to read configuration data concerning the external node with which the SN can communicate.

To use the parser, the component or application supplies a specification for data to be read when it creates the parser. Once the specification is taken in by the parser, the using component or application can ask the parser to read a stream of input and parse it according to the specification. The output from this exchange is a data structure that contains (in a standard form) the data from the file.

The benefit of the parser is that it provides simple syntax checking for input, and thus frees the component or application from having to do routine error checking on input.

The parser is used in the API code as a mechanism for reading in configuration information. It is also used by the OSSAIN OAP Simulator tool (built on top of the API) to specify syntax of simulator scripts that can drive a simulation.

Note: For details on the configuration utility, refer to Chapter 9: "Configuration and administration."

When deciding how to use the parser to supply data, the developer has the following two options, which are discussed in this section:

- adding user data with the BeginUserData block
- building a parser object

Adding user data with the BeginUserData block

The node configuration specification includes a BeginUserData block that allows an application to enter a set of key-value pairs. The NI layer code reads the user data into an instance of the node configuration data structure OA_ConfigData upon node startup.

Figure 97 shows an example of user data in a configuration file.

Figure 97 Example of user data included in configuration file

```
BeginNode 103

# MAX_ANNOUNCEMENT_WAIT_TIME is the number of seconds our
# application will wait before deciding that the IVR
# is not responding
  BeginUserData
    MAX_ANNOUNCEMENT_WAIT_TIME 10
  EndUserData

EndNode
```

The OA_Framework class Configure() method should be called by the application when it initializes. This method tells the framework to instruct the parser to read in a file and set up an OA_ConfigData instance containing all the data read from the file. From that point on, the node framework has a pointer to the configuration data.

The context object provides a user data area in the form of an OA_Datablock. Although a class from the PI layer, the OA_Datablock class provides a conveniant mechanism in which to store user information in the form of key-value pairs. This class provides setField() and getField() methods for storing and retrieving data.

The buildNodeContext(), buildPoolContext(), and buildSesnContext() methods of the OA_Framework class can be overridden to allow an application developer to retrieve user specific data from the OA_ConfigData instance and place it into the OA_Datablock object passed to the context object constructor.

At this point the application can retrieve user data from the context object during processing in a maintenance or call processing state machine. This is illustrated in Figure 98.

Figure 98 Retrieving user data

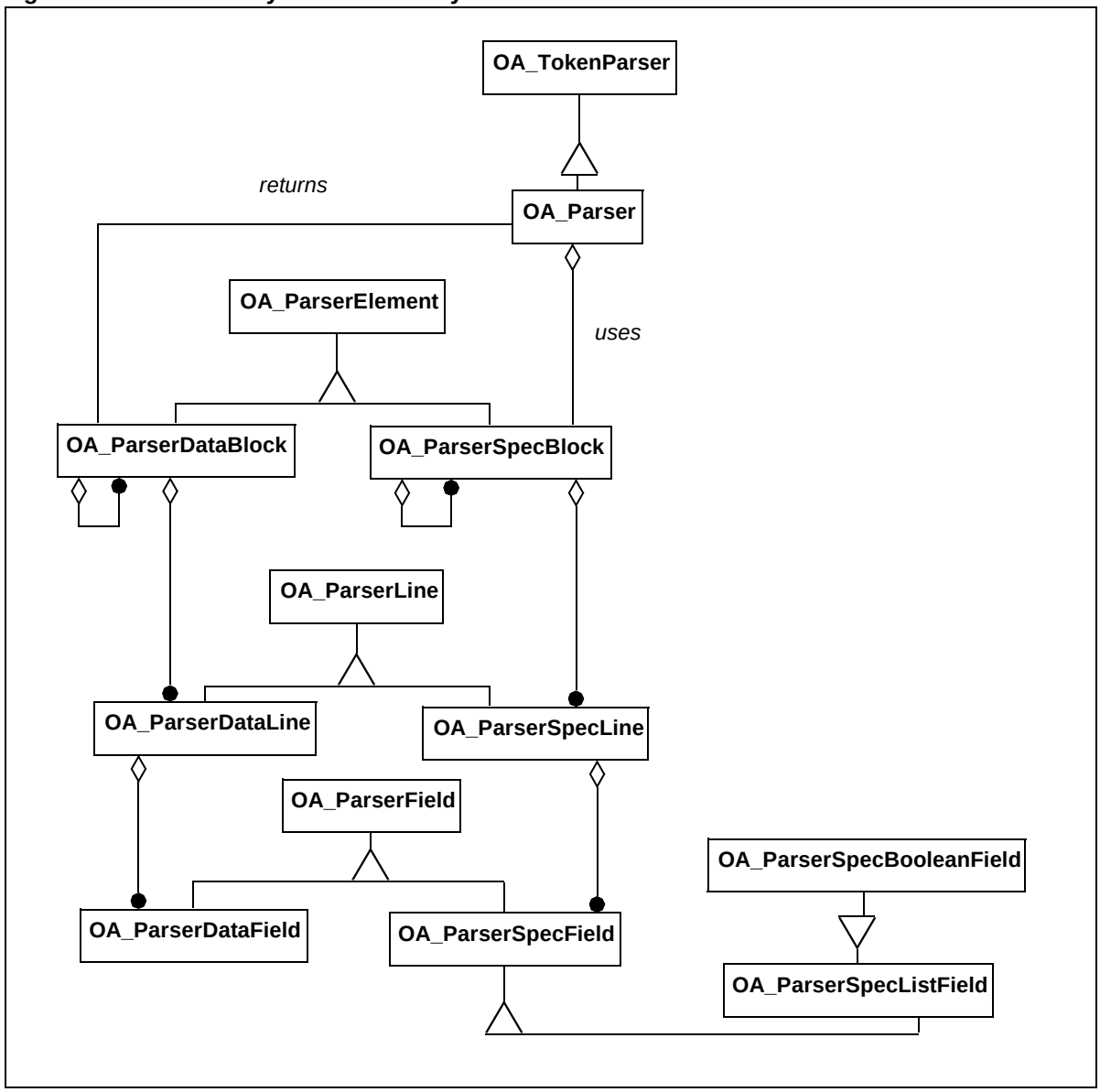
```
// Node infrastructure asks user state to process an incoming event
// The context also contains saved user data.
UserState->process( inEvent, aContext );
...
// In process()...
OA_UDWRD waitTime = 0;
OA_UDWRD indexOfFieldToGet = 1;
OA_RCODE RC = OA_RC_SUCC;

aContext->userData->getField( waitTime,  "MAX_ANNOUNCEMENT_WAIT_TIME" );
```

Building a parser object

If an application needs to use input data for a purpose besides configuration data, the best solution may be for the application to construct its own parser instance and have it read the data.

Figure 99 shows the class hierarchy for the parser utility. The important lines in this figure are the two labeled *uses* and *returns*. The parser uses the specification it receives from the application in parsing input, and the product of its activity is a parsed data block containing the data read.

Figure 99 Parser utility class hierarchy



As Figure 99 shows, a specification is an instance of the OA_ParserSpecBlock class. The instance can contain nested instances of itself as well as a list of OA_ParserSpecLine instances. The OA_ParserSpecLine instances in turn contain OA_ParserSpecField instances. A spec line specifies the syntax of a line by setting forth the fields that can appear in the line.

To create a specification, the application must create an instance of the spec block class, then load it with contained blocks or lines (or both). A block has a tag line that identifies the beginning of the block, and this line can contain a value that further identifies the block.

A block also has a tag line that identifies the end of a block. This line cannot take a value. (Refer to Figure 97 on page 166 for an example of the BeginNode and EndNode block in the API configuration data specification. For more information on the configuration file, refer to Chapter 9: "Configuration and administration.")

Parsing rules

The parser follows a standard set of rules when using the specification. These rules cannot be changed without changing the code of the parser itself.

- Syntax checking is done on a per-line basis. A line is identified to the parser by the tag string at the beginning of the line. Fields extend to the right. The right most field or fields can be optional.
- Nested blocks or lines can be designated as optional. A default value can be assigned to a field.
- Fields are verified to be within a numeric range, to fit a regular expression in the field specification, or to match a literal string, depending on the type of field specification used.
- The parser does not care about the ordering of lines or blocks. However, the parser preserves the order in which the input is read, and the application can do its own verification on the data to ensure that the input follows a desired ordering.
- A field name can appear only once in a line. A line name can appear only once in a particular block.
- Special characters, including the comment, delimiter, continuation, and quote and ignore characters can be changed upon construction of the parser. Multiple characters can be designated for each purpose.
- By default, an octothorpe (#) comments from its position on a line to the end of line. The \ character by default continues a line, and within a line it escapes the following character. A string that includes blanks must be quoted to be considered a single field value, and quoted fields cannot span lines. Default ignore characters include tabs, spaces, and form feeds.

A syntax error, such as extra fields, causes the parser to set a boolean in its member data called problemFound. An OA_RCODE returned from the parsing indicates which type of error occurred. Also, many syntax errors are flagged using alarm logs.

Parsing and using parsed input

Once the application sets up its specification and creates a parser, it can instruct the parser to read input. The class constructor, OA_Parser(), builds the parser object. The parseFile(), parseBuffer(), and parseStream() routines read input and apply the specification to the input. The OA_ParseDataBlock pointer received from the parser offers an interface for the application to get and use the input data.

Table 14 shows the public interface of the class OA_Parser.

Table 14 OA_Parser **class public interface**

Public methods	Description
OA_RCODE parseFile(const char* fileName, OA_ParserDataBlock* &data)	Read input from file
OA_RCODE parseBuffer(const char* buffer, OA_ParserDataBlock* &data)	Read from char buffer
OA_RCODE parseStream(istream& source, OA_ParserDataBlock* &data)	Read from stream source
OA_Parser(OA_ParserSpecBlock* spec, const char* delimiter = OA_PARSER_DEFAULT_DELIMITER, const char* cont = OA_PARSER_DEFAULT_CONTINUE, const char* comment = OA_PARSER_DEFAULT_COMMENT, const char* eat = OA_PARSER_DEFAULT_EAT, const char* quote = OA_PARSER_DEFAULT_QUOTE)	Constructor that takes a specification and optional strings of special characters
~OA_Parser()	Destructor

Table 15 shows the public interface of the class OA_ParseDataBlock.

Table 15 OA_ParseDataBlock **public interface**

Public methods	Description
virtual OA_RCODE addLine(OA_ParserDataLine* line)	Used by parser to add parsed input to data
virtual OA_RCODE addBlock(OA_ParserDataBlock* block)	Used by parser to add parsed input to data
OA_RCODE getValue(const char* lineTag, const char* fieldName, char* &value)	Application can use to get a field value
OA_RCODE getValue(const char* lineTag, const char* fieldName, OA_UDWRD &value)	Application can use to get a field value

Table 15 OA_ParseDataBlock **public interface**

Public methods	Description
OA_RCODE getFirstElement(OA_UDWRD &num, OA_ParserElement* &elem)	Get first for numeric fields
OA_RCODE getNextElement(OA_UDWRD &num, OA_ParserElement* &elem)	Get next for numeric fields
OA_RCODE getFirstElement(const char* tag, OA_UDWRD &num, OA_ParserElement* &elem)	Get first for string fields
OA_RCODE getNextElement(const char* tag, OA_UDWRD &num, OA_ParserElement* &elem)	Get next for string fields
OA_ParserDataBlock(const char* blockTag)	Constructor; the parser creates one to export parsed data
~OA_ParserDataBlock()	Destructor

Figure 100 shows a program that constructs a specification block for data called BeginData. The program populates the block with a single field, the Hello field. The Hello field must contain a value that conforms to the regular expression [0-9A-Fa-f]*.

Figure 100 Creating and using a parser

```
#include "oa/parser.h"

// Name of file to read in and parse
const char* PARSERFILENAME = "simple_test.txt";

int main( )
{
  OA_Parser*          parser;
  OA_ParserSpecBlock* specBlock1;
  OA_ParserSpecLine*  specLine;
  OA_ParserDataBlock* dataBlock1;

  char* dataValue = OA_NULL;
  OA_RCODE RC = OA_UNKNOWN_RC;

  // Turn on log utility
  OA_Log::quickConfigure();

  // Create a block for our spec with a BeginData tag and
  // a generic end tag.
  OA_ParserSpecLine* genericEnd = new OA_ParserSpecLine( "End", 1, 1 );
  specBlock1 = new OA_ParserSpecBlock(
      new OA_ParserSpecLine( "BeginData", 1, 1), genericEnd );

  // Create a line for our spec that will accept the string
  // "Hello <hexidecimal value>"
  specLine  = new OA_ParserSpecLine(  "Hello" );
  specLine->addStringField( "val", OA_TRUE, "[0-9A-Fa-f]*" );
  specBlock1->addLine( specLine );

  // Create the parser using the spec
  parser = new OA_Parser( specBlock1 );

  // Parse the file and display the received datablock
  RC = parser->parseFile( PARSERFILENAME, dataBlock1 );

  if ( RC == OA_RC_SUCC )
    cout << dataBlock1;
  else
    cout << RC << endl;

  delete dataBlock1;
  delete parser;

  return 0;
}
```

Figure 101 shows a parse file that does not meet the specification due to invalid value “World”.

Figure 101 Incorrect parse file

```
# Comment
#
Begindata
  Hello World
end
```

Figure 102 shows the log printed by the program when run against the incorrect parse file.

Figure 102 Results of incorrect parse

```
 ALARM     Minor  #     1     ParserFieldValidationFailed  1998 Jun 15 14:41:47
--------------------------------------------------------------------------------
String value "World" fails to match regular expression "[0-9A-Fa-f]*"
required for field 'val'.

 ALARM     Major  #     1      ParserMandatoryFieldError  1998 Jun 15 14:41:47
--------------------------------------------------------------------------------
Error parsing file "simple_test.txt" at line 4 (line beginning "Hello")
Validation error parsing "World" as mandatory field 'val', line ignored.

12
```

Figure 103 shows a correct parse file.

Figure 103 Correct parse file

```
# Comment
#
Begindata
  Hello ABCDEF
end
```

Figure 104 shows the results of the correct parse. The output results from the program inserting the returned data block object on the cout stream.

Figure 104 Results of correct parse

```
Block Begindata
{
  Line Begindata

  Line Hello
    Field val == "ABCDEF"

  Line end
}
```

Chapter 9: Configuration and administration

This chapter describes configuration and administration for the SN application. Developers, testers, and maintainers need this information to set up the configuration file for the application. Operating company personnel need this information to understand how to coordinate the data between the SN and the DMS switch.

This chapter provides the following information:

- the file format of the configuration data
- port settings
- parallel datafill considerations
- a sample configuration file
- engineering considerations
- administration considerations

File format

The configuration file format consists of key-value pairs. The key is the predefined symbol that identifies the type of data provided. The value is the actual data. The key-value pairs are contained inside Begin-End blocks, which group related data together.

The following rules apply to the file format:

- The Begin key requires a data value; the End key does not.
- The key and value are separated by at least one space.
- A key can appear only once per block.
- If the value is a string of characters that contains a space, it must be contained in double quotation marks (“ ”).
- Numeric values are assumed to be in decimal (base 10) format, unless preceded by 0x, which indicates hexadecimal (base 16).
- The keys and the Begin . . . End symbols are not case-sensitive, but data values are.

- Lines beginning with a hash mark (#) are ignored and can be used for comments.

Figure 105 shows an example of key-value pairs in a configuration file. In the example for SN 103, the value for the NodeName key is MYNODE and the value for the NodeType is SN.

Figure 105 Example of key-value pairs

```
BeginNode 103

  NodeName MYNODE
  NodeType SN

# . . .

EndNode
```

BeginNode block

The configuration file consists of the BeginNode block, which contains all the data for an SN. This block contains the nine other blocks shown in the following list:

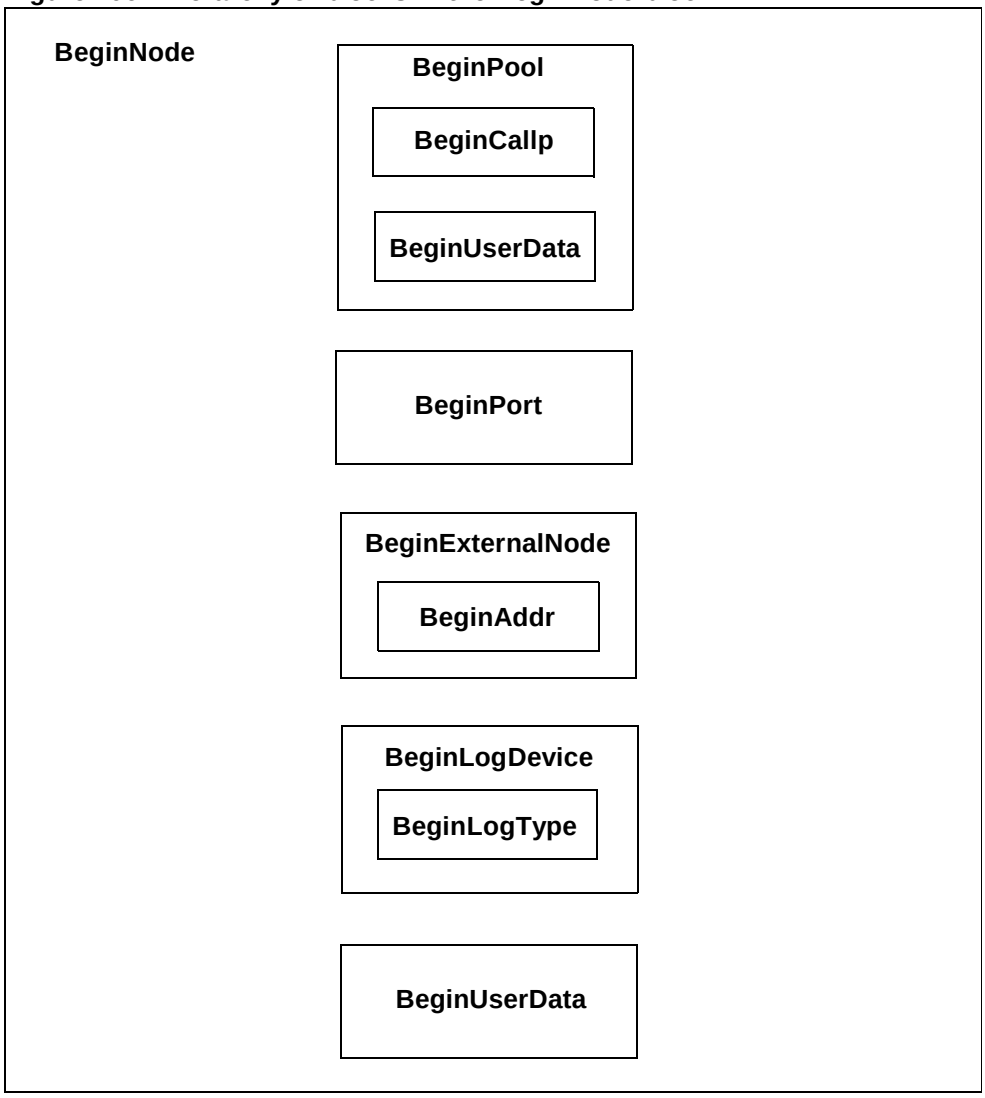
- BeginPool block, which contains:
  - — BeginCallp block
  - — BeginUserData block
- BeginPort block
- BeginExternalNode block, which contains:
  - — BeginAddr block
- BeginLogDevice block, which contains:
  - — BeginLogType block
- BeginUserData block

Note 1: The BeginUserData block can appear under both the BeginPool block and the BeginNode block.

Note 2: Refer to the next section for information on which blocks and key-value pairs are mandatory.

Figure 106 illustrates the hierarchy of blocks in the BeginNode block. Other than the hierarchy shown, no order applies to the arrangement of blocks in the configuration file. Likewise, no order applies to the arrangement of key-value pairs within a block.

Figure 106 Hierarchy of blocks in the BeginNode block



The tables in this section provide details on the key-value pairs in each block. Each table includes the following information:

- an explanation of the key, including whether the value must match DMS switch datafill
- the range of valid values
- whether the key-value pair is mandatory (M) or optional (O)
- the default value, where applicable

BeginNode block

The BeginNode block is mandatory and can appear only once in the configuration file. Table 16 lists valid values for the BeginNode key-value pairs.

Table 16 BeginNode key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginNode	The node ID as datafilled in the switch table OANODNAM (OSSAIN Node Name)	Numeric 0 to 767	M	n/a
NodeName	The name of the node as datafilled in table OANODNAM	Alphanumeric string up to 12 characters	O	See Note 1
NodeType	The type of OSSAIN node configured	See Note 2	O	SN
Priority	The node processing priority, which determines how many messages the node processes before polling the transport for more messages	Numeric 1 to 999999999	O	50
MailboxSize	The maximum number of messages per mailbox, which is the upper limit for the number of received messages waiting to be processed	Numeric 1 to 999999999	O	5000
NodeMtcPort	The number of the node maintenance port as datafilled in table OANODINV (OSSAIN Node Inventory)	Numeric 1 to 65535 (for details, refer to "Setting a value for NodeMtcPort" on page 189)	O	n/a
MISDataPort	The number of the MIS data port as datafilled in table QMSMIS.	Numeric 1 to 65535 (for details, refer to "Setting a value for MISDataPort" on page 189)	O	n/a
RequestTimeout	The request timeout in milliseconds for node messages	Numeric 1 to 999999999	O	20000
MinVersion	The minimum OAP version supported	X.Y (see Note 3)	O	See Note 3

Table 16 BeginNode key-value pairs

Key	Explanation	Valid value	M or O?	Default value
MaxVersion	The maximum OAP version supported	X.Y (see Note 3)	O	See Note 3
Validation	Whether incoming operations are validated for correct syntax	On, Off	O	On (see Note 4)
ClassHeaderValidation	Whether the class header of an incoming operation is validated for correct syntax	On, Off	O	On (see Note 4)
OpHeaderValidation	Whether the operation header of an incoming operation is validated for correct syntax	On, Off	O	On (see Note 4)
DatablockValidation	Whether the data blocks of an incoming operation are validated for correct syntax	On, Off	O	On (see Note 4)
OpStructureValidation	Whether the presence of the correct data blocks of an incoming operation is verified	On, Off	O	On (see Note 4)
DMSLog	Whether sending an SN log to the switch is enabled or disabled for the node	On, Off	O	On
DMSAlarm	Whether sending an SN alarm to the switch is enabled or disabled for the node	On, Off	O	On
DMSAlarmLevel	The severity level of the alarm sent	Critical, major, minor (see Note 5)	O	Minor

Table 16 BeginNode key-value pairs

Key	Explanation	Valid value	M or O?	Default value
EndNode	The End key for the BeginNode block	n/a	n/a	n/a

Note 1: If no NodeName value is specified, the API displays the node ID instead in any API-generated logs.

Note 2: The API recognizes the following NodeType values: SN, Host, Remote, Admin, MISHost, and MISNode. Call processing service nodes should use a value of SN. Nodes that collect QMS MIS data should use a value of MISNode. Host, Remote, and MISHost values represent DMS switches, and should be used only by a DMS simulator. A value of Admin should be used only by OA&M tools.

Note 3: Versions of OAP are represented by the release number, a period, and the increment number. For example, 4.0 represents OAP release 4 increment 0. The proper value depends on the design of the application.The default value is the minimum or maximum version that the API can support.

Note 4: If the Validation value is set to Off, the following validation values are ignored: ClassHeaderValidation, OpHeaderValidation, DatablockValidation, and OpStructureValidation. The Validation value must be set to On before any of the other validations are performed.

Note 5: Severity levels are cumulative. Critical means only critical alarms, major means both major and critical alarms, and minor means all alarms.

BeginPool block

The BeginPool block contains the data for a session pool. Each session pool datafilled in table OASESNPL (OSSAIN Session Pool) requires a BeginPool block. This block is optional and can appear more than once under the BeginNode block. The BeginPool block contains the BeginCallp block and a BeginUserData block. Table 17 lists valid values for the BeginPool key-value pairs.

Table 17 BeginPool key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginPool	The session pool ID as datafilled in table OASESNPL	Numeric 0 to 4094	O	n/a
PoolName	The name of the session pool as datafilled in table OASESNPL	Alphanumeric string up to16 characters	O	See Note 1
MaxSessions	The maximum number of sessions as datafilled in table OASESNPL	Numeric 0 to 1023	M	n/a
OrigType	The session origination type as datafilled in table OASESNPL	SN or SUBSCRIBER	M	n/a

Table 17 BeginPool key-value pairs

Key	Explanation	Valid value	M or O?	Default value
CallpPort	The call processing port as datafilled in table OASESNPL	Numeric 1 to 65535 (for details, refer to "Setting a value for CallpPort" on page 189)	O	n/a
PoolMtcPort	The session pool maintenance port as datafilled in table OASESNPL	Numeric 1 to 65535 (for details, refer to "Setting a value for PoolMtcPort" on page 189)	O	n/a
RequestTimeout	The request timeout in milliseconds for session pool messages	Numeric 1 to 999999999	O	20000
MinVersion	The minimum OAP version supported	X.Y (see Note 2)	O	See Note 2
MaxVersion	The maximum OAP version supported	X.Y (see Note 2)	O	See Note 2
DMSLog	Whether sending an SN log to the switch is enabled or disabled for the session pool	On, Off	O	On
DMSAlarm	Whether sending an SN alarm to the switch is enabled or disabled for the session pool	On, Off	O	On
DMSAlarmLevel	The severity level of the alarm sent	Critical, major, minor (see Note 3)	O	Minor
EndPool	The End key for the BeginPool block	n/a	n/a	n/a

Note 1: If no PoolName value is specified, the API displays the session pool ID instead in any API-generated logs.

Note 2: Versions of OAP are represented by the release number, a period, and the increment number. For example, 4.0 represents OAP release 4 increment 0. The proper value depends on the design of the application.The default value is the minimum or maximum version that the API can support.

Note 3: Severity levels are cumulative. Critical means only critical alarms, major means both major and critical alarms, and minor means all alarms.

BeginCallp block

The BeginCallp block identifies the call processing application for the session pool. This block is optional and can appear only once in the BeginPool block. Table 18 lists valid values for the BeginCallp key-value pairs.

Table 18 BeginCallp key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginCallp	The name of the call processing application	Alphanumeric string up to 30 characters	M	See Note
RequestTimeout	The request timeout in milliseconds for session messages	Numeric 1 to 999999999	O	5000
EndCallp	The End key for the BeginCallp block	n/a	n/a	n/a
Note: If no BeginCallp value is specified, the API displays the session pool name in any API-generated logs.				

BeginUserData block

The BeginUserData block contains application-specific configuration data. This block is optional and can appear no more than once in the BeginNode block and no more than once in each BeginPool block.

There are no OSSAIN API-recognized tags that can or must appear in the BeginUserData block; applications define their own key-value pairs. Users should consult application documentation for any required key-value pairs.

Table 19 lists valid values for the BeginUserData block key-value pairs.

Table 19 BeginUserData key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginUserData	The Begin key for the BeginUserData block	n/a	n/a	n/a
Value	Any key-value pair needed by the application; can appear any number of times until the EndUserData tag	String up to 255 characters	M	n/a
EndUserData	The End key for the BeginUserData block	n/a	n/a	na/

BeginPort block

The BeginPort block contains the data related to a transport. This block is optional and can appear more than once in the BeginNode block. Table 20 lists valid values for the BeginPort key-value pairs.

Table 20 BeginPort key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginPort	The port number (see Note 1)	Numeric1 to 65535	O	n/a
TransProtocol	The protocol of the transport	UDP, TCP	M	UDP
AppProtocol	The protocol of the application	Alphanumeric string up to 255 characters (see Note 2)	M	OAP
Priority	The transport processing priority, which determines how many messages are received before processing them	Numeric 1 to 999999999	O	10
BufferSize	The size in bytes of the socket buffer available to store unread messages	8192 to 999999999 (see Note 3)	O	65535
TimedBlocking	The timed blocking value in milliseconds, which specifies how much time is spent polling a transport while no messages are available	Numeric 0 to 999999999	O	0
PortType	The function of the transport	See Note 4	O	Unknown
IsPassive	Whether TCP port passively accepts connections (see Note 5)	Y, YES, N, NO	O	Y
KeepAlive	Whether to verify the TCP connections of the socket if it becomes idle for a long period of time (see Note 5)	Y, YES, N, NO	O	N
NoDelay	Whether to send small TCP packets immediately, rather than buffer and collect them into large packets before shipment (see Note 5)	Y, YES, N, NO	O	N
InMessageLength	Fixed size of incoming TCP messages (see Note 5)	Numeric 0 to 4096	O	0 (see Note 6)

Table 20 BeginPort key-value pairs

Key	Explanation	Valid value	M or O?	Default value
OutMessageLength	Fixed size of outgoing TCP messages (see Note 5)	Numeric 0 to 4096	O	0 (see Note 7)
EndPort	The End key for the BeginPort block	n/a	n/a	n/a

Note 1: The port number must not conflict with other ports on the system. If no BeginPort blocks are present, the default port settings are used based on the NodeType of the BeginNode block. Refer to Table 26, “Default port settings for node types,” on page 190.

Note 2: Enter OAP for ports that send or receive OAP messages. For other protocols, enter the name of the protocol.

Note 3: The maximum buffer size has platform-specific limitations. HP-UX 9 restricts buffer size to no more than 58254 bytes of real memory (virtual memory cannot be used for socket buffers). Windows NT 4.0 makes no restriction other than total memory available to the operating system.

Note 4: Valid port types depend on the NodeType of the BeginNode block. Refer to Table 26, “Default port settings for node types,” on page 190 for valid port types per node type. If port type NodeMtc is specified, the port number must match the NodeMtcPort value in the BeginNode block (if present). If port type PoolMtc or Callp is specified, the port number must match the PoolMtcPort or CallpPort (respectively) in a BeginPool block (if present).

Note 5: This value is meaningful only for TCP transports and is ignored for UDP transports.

Note 6: If InMessageLength is set to 0, incoming TCP messages are assumed to be variable length.

Note 7: If OutMessageLength is greater than 0, outgoing TCP messages are padded with extra bytes (if necessary) until they reach OutMessageLength. If OutMessageLength is set to 0, no padding is done.

BeginExternalNode block

The BeginExternalNode block contains data about an external OSSAIN node. This block is mandatory and can appear more than once in the BeginNode block. The BeginExternalNode block contains the BeginAddr block. Table 21 lists valid values for the BeginExternalNode key-value pairs.

Table 21 BeginExternalNode key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginExternalNode	The node ID of the external node as datafilled in table OANODINV	Numeric 0 to 767	M	n/a
NodeName	The name of the node as datafilled in table OANODNAM	Alphanumeric string up to 12 characters	M	See Note 1
NodeType	The relationship with the external node	See Note 2	O	Unknown
IpAddr	The IP address of the external node	Numeric 0 to 255 (see Note 3)	M	n/a
StatusPerPool	Whether the status of the external node is kept on a per-pool basis (see Note 4)	Y, Yes, N, No	M	Y
EndExternalNode	The End key for the BeginExternalNode block	n/a	n/a	n/a

Note 1: If no NodeName value is specified, the API displays the node ID instead in any API-generated logs.

Note 2: The API recognizes the following NodeType values: SN, Host, Remote, Admin, MISHost, and MISNode. Call processing nodes should configure the standalone DMS switch, or host switch in an OSAC environment, as Host. MIS nodes should configure the standalone or host switch as MISHost. OSAC remote switches should be configured as Remote. External SNs should be configured as SN or MISNode. Administrative nodes should be configured as Admin.

Note 3: The IP address is in the form of x.x.x.x. The API supports IPv4.

Note 4: The StatusPerPool value determines how the status of the external node is stored. When this value is set to Y, the status is kept independently for each session pool. When this value is set to N, all session pools are updated with any status change. DMS switches should always be set to Y.

BeginAddr block

The BeginAddr block contains data about a port on an external node. This block is optional and can appear only once in the BeginExternalNode block. Table 22 lists valid values for the BeginAddr key-value pairs.

Table 22 BeginAddr key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginAddr	The address index, which must match the value used by the application to refer to the port of the external node	Numeric 1 to 65535	O	n/a
Port	The port number, which must match the port number of the external node	Numeric 1 to 65535	M	n/a
TransProtocol	The protocol of the transport	UDP, TCP	M	UDP
AppProtocol	The protocol of the application	Alphanumeric string up to 255 characters (see Note 1)	M	OAP
PortType	The function of the external port	See Note 2	O	Unknown
IsPassive	Whether TCP port passively accepts connections (see Note 3)	Y, YES, N, NO	O	N
EndAddr	The End key for the BeginAddr block	n/a	n/a	n/a

Note 1: Enter OAP for ports that send or receive OAP messages. For other protocols, enter the name of the protocol.

Note 2: Valid port types depend on the NodeType of the BeginExternalNode block. Refer to Table 26, “Default port settings for node types,” on page 190 for valid port types per node type.

Note 3: This value is meaningful only for TCP ports and is ignored for UDP ports.

BeginLogDevice block

The BeginLogDevice block defines the characteristics of a particular log output device. This block is optional and can appear more than once in the BeginNode block. The BeginLogDevice block contains the BeginLogType block.

The application can define its own device types. The API defines the following device types:

- file
- console

Table 23 lists valid values for the BeginLogDevice key-value pairs.

Table 23 BeginLogDevice key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginLogDevice	The type of device	Console, file, or other string up to 255 characters (see Note 1)	M	n/a
BufferSize	The size in bytes of the buffer available to store logs before they are flushed to output	0 to 999999999 bytes (see Note 2)	O	1024
MaxLogsInBuffer	The maximum number of logs stored in the buffer before they are flushed	Numeric 1 to 999999999	O	1
Stream	The type of output stream (see Note 3)	Stdout, stderr	O	Stdout
Filename	The name of the log file (see Note 4)	See Note 5	M	n/a
NumFiles	The number of log files (see Note 4)	Numeric 1 to 999999999	O	1
LogsPerFile	The maximum number of logs per file (see Note 4)	Numeric 0 to 999999999 (see Note 6)	O	0
EndLogDevice	The End key for the BeginLogDevice block	n/a	n/a	na/

Note 1: For application-defined log devices, specify the name of the device as recognized by the application.

Note 2: Buffer size is limited by available memory.

Note 3: This field is only relevant to console log devices; it is ignored for other devices.

Note 4: This field is only relevant to file log devices; it is ignored for other devices.

Note 5: Filenames must be valid for the operating system used. Log files are kept in the current working directory unless otherwise specified by a full or relative directory path. If multiple log files are used, log file numbers are appended to the specified filename. Any existing files are overwritten without notice.

Note 6: Specify 0 to indicate no limit on the number of logs in the log file. The NumFiles field is ignored in this case. Log file size is restricted by available disk space. No further logs are written after disk space is exhausted.

BeginLogType block

The BeginLogType block defines the log types and levels that are active for a particular log output device. This block is mandatory when the BeginLogDevice block appears. It can appear more than once in the BeginLogDevice block.

The application can define its own log types and levels. Table 24 lists the API-defined log types and levels, and the default level.

Table 24 API-defined log types and levels

Log type	Levels	Default level
Alarm	Critical, major, minor	Minor
Debug	Brief, detailed	Brief
Report	Brief, detailed	Brief
Status	Brief, detailed	Brief
Trace	Brief, detailed	Brief
Swerr	n/a	n/a

Refer to Chapter 8: "Additional API components," for details on the API log utility. Table 25 lists valid values for the BeginLogType key-value pairs.

Table 25 BeginLogType key-value pairs

Key	Explanation	Valid value	M or O?	Default value
BeginLogType	The type of log for the device specified in the BeginLogDevice block	Refer to Table 24 for API-defined values	M	n/a
Level	The level of log output	Refer to Table 24 for API-defined values	O	Refer to Table 24 for API-defined values
EndLogType	The End key for the BeginLogType block	n/a	n/a	n/a
Note: For application-defined log types, specify the name of the type as recognized by the application; specifying a level is optional.				

BeginUserData block

Refer to "BeginUserData block" on page 182 for information.

Port settings

The DMS switch has a lower bound of 1024 for port numbers, because it does not communicate through OAP with applications using well-known port numbers below this value. The SN is not restricted to OAP communication, and should not have this port number restriction. The switch has an upper bound of 32767 due to table control restrictions. The SN uses its own data storage facility, so this restriction does not apply.

Setting a value for NodeMtcPort

If a value for NodeMtcPort is absent, the default value is based on NodeType. Service node types (SN and MISNode) use 7000 as a default. Switch types (Host, Remote, and MISHost) use 5290 as a default. Admin nodes do not use the NodeMtcPort value.

A BeginPort block that matches the NodeMtcPort value can be used to configure node maintenance port settings. If no corresponding BeginPort block exists, default port settings are used as shown in Table 26, "Default port settings for node types," on page 190.

Setting a value for CallpPort

If a value for CallpPort is absent, the default value is based on NodeType. SNs use 7001 as a default. Switch types (Host, Remote, and MISHost) use 5289 as a default. Admin nodes and MISNodes do not use the CallpPort value.

A BeginPort block that matches the CallpPort value can be used to configure call processing port settings. If no corresponding BeginPort block exists, default port settings are used as shown in Table 26, "Default port settings for node types," on page 190.

Setting a value for PoolMtcPort

If a value for PoolMtcPort is absent, the default value is based on NodeType. SNs use 7002 as a default. Switch types (Host, Remote, and MISHost) use 5291 as a default. Admin nodes and MISNodes do not use the PoolMtcPort value.

A BeginPort block that matches the PoolMtcPort value can be used to configure pool maintenance port settings. If no corresponding BeginPort block exists, default port settings are used as shown in Table 26, "Default port settings for node types," on page 190.

Setting a value for MISDataPort

If a value for MISDataPort is absent, the default value is based on NodeType. MISNode types use 5000 as a default. MISHost types use 8298 as a default. True MIS host switches dynamically choose the port number; the MIS host simulator uses the arbitrarily chosen port number 8298. No other node types use the MISDataPort value.

A BeginPort block that matches the MISDataPort value can be used to configure MIS data port settings. If no corresponding BeginPort block exists, default port settings are used as shown in Table 26, "Default port settings for node types," on page 190.

Default port settings for node types

Table 26 shows the default settings for the valid port types that correspond to each node type.

Table 26 Default port settings for node types

Node type	Valid port type	Default settings
SN	NodeMtc	Port: 7000 (see Note 1) Transp: UDP
	Callp	Port: 7001 (see Note 2) Transp: UDP TimedBlocking: 100
	PoolMtc	Port: 7002 (see Note 2) Transp: UDP
Host	NodeMtc	Port: 5290 Transp: UDP
	Callp	Port: 5289 Transp: UDP TimedBlocking: 100
	PoolMtc	Port: 5291 Transp: UDP
	Logs	Port: 5292 Transp: UDP
	Alarms	Port: 5293 Transp: UDP
Remote	Callp	Port: 5289 Transp: UDP TimedBlocking: 100
	PoolMtc	Port: 5291 Transp: UDP

Table 26 Default port settings for node types

Node type	Valid port type	Default settings
MISNode	NodeMtc	Port: 7000 (see Note 1) Transp: UDP
	MISData	Port: 5000 Transp: TCP Passive: True InMsgLen: 4096 OutMsgLen: 34 TimedBlocking: 100
MISHost	NodeMtc	Port: 5290 (see Note 1) Transp: UDP
	MISData	Port: 8298 (see Note 3) Transp: TCP Passive: False InMsgLen: 34 OutMsgLen: 4096 TimedBlocking: 100
Admin	Admin	Port: 4294 Transp: UDP

Note 1: If the NodeMtcPort tag is present in the BeginNode block, that value is used instead.

Note 2: The first BeginPool block with an OrigType of SUBSCRIBER has default callp/poolMtc ports 7001/7002. Each subsequent subscriber-origination BeginPool block has default ports incremented by 1000: the second BeginPool block uses 8001/8002, the third uses 9001/9002, and so on, until 32001/32002.

The first BeginPool block with an OrigType of SN has default callp/poolMtc ports 7011/7012. EAch subsequent SN-origination BeginPool block has default ports incremented by 1000 up to 32011/32012.

If the Callp/PoolMtcPort tags are present in the BeginPool block, these values are used instead.

Note 3: A true MIS host switch dynamically chooses the TCP port number for MIS data before establishing the connection. MIS host simulators use the arbitrarily chosen port number 8298. MIS nodes can accept any TCP port number.

Parallel datafill considerations

Some OSSAIN datafill needs to be coordinated in the switch and the SN. This section discusses parallel datafill considerations and shows sample data, focusing on the following areas:

- node, session pool, and session
- standalone configuration
- OSAC configuration
- additional datafill for functions, control lists, and voice links

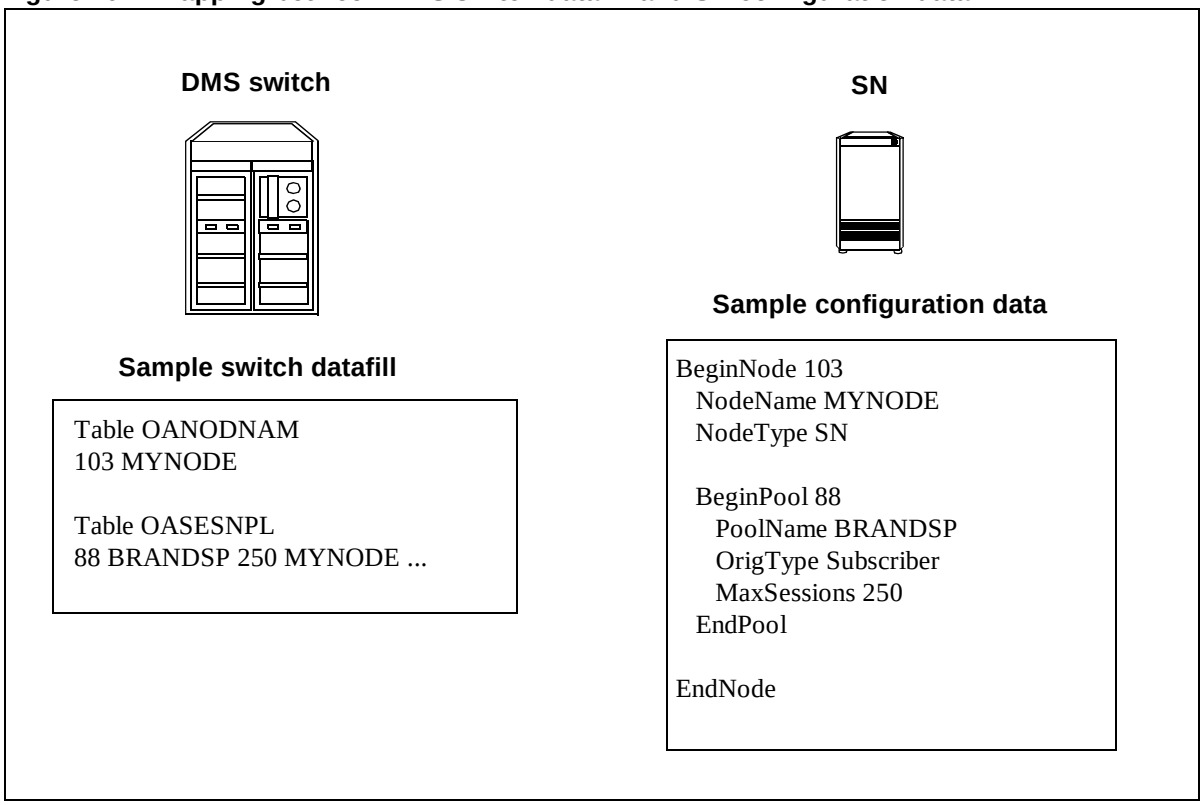
Datafill for node, session pools, and sessions

Figure 107 shows a high-level view of the mapping between the DMS switch datafill and the SN configuration data for the node, session pool, and sessions. Sample OSSAIN switch data is from the following tables:

- OANODNAM lists the service node identifier (ID) and name.
- OASESNPL defines all OSSAIN session pools.

The example shows an SN called MYNODE with an ID of 103. Its branding session pool, called BRANDSP, has an ID of 88 and can handle a maximum of 250 sessions.

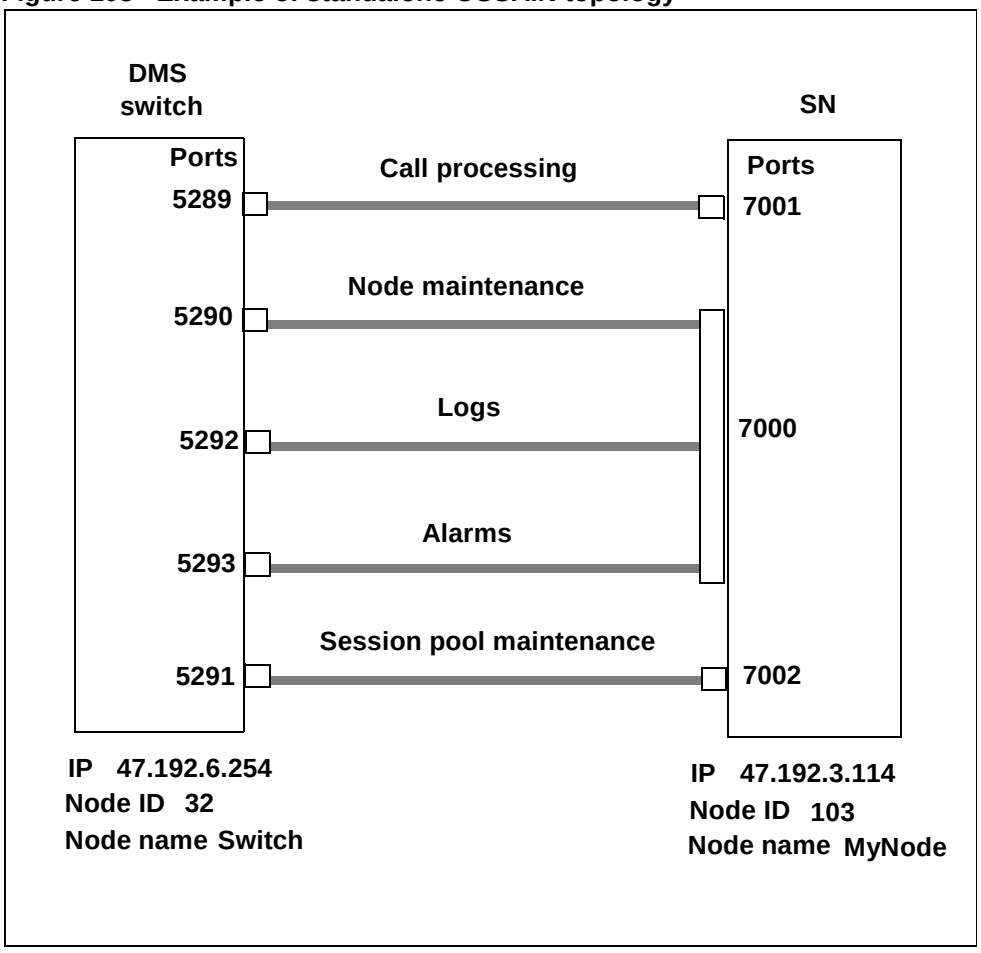
Figure 107 Mapping between DMS switch datafill and SN configuration data



Datafill for standalone OSSAIN configuration

Figure 108 shows sample data for a standalone configuration, including IP addresses and port numbers. (The example IP addresses are for illustration only.)

Figure 108 Example of standalone OSSAIN topology



The switch uses the following pre-defined port numbers for maintenance and call processing:

- 5289 for call processing
- 5290 for node maintenance
- 5291 for session pool maintenance
- 5292 for logs
- 5293 for alarms

Note: Figure 108 shows only the logical (not physical) relationship between ports on the DMS switch and the SN. In the figure, the SN uses port 7001 to originate and receive all node and session pool maintenance conversations. This is not a restriction. However, all node-level maintenance ports should communicate with the same port on the SN. Likewise, all session pool-level maintenance ports should communicate with the same port on the SN.

Figure 109 shows sample DMS switch datafill for the MYNODE SN and BRANDSP session pool in tables OANODNAM, OANODINV, and OASESNPL.

Figure 109 Switch datafill example for SN and session pool

OANODNAM

```
NODEID     NODENAME
-------------------------------------------------------------------------------
32         SWITCH
103        MYNODE
```

OANODINV

```
NODENAME    NODEAREA
-------------------------------------------------------------------------------
SWITCH   OSAC SELF 8 6 30
MYNODE   OSNM 1 UDP IPV4 47 192 3 114 7000 Y 2 60 240 60 SN 4 BB 3 CITYA BRANDING1
   8 6 30
```

OASESNPL

```
SESNPLID  SESNPLNM  MAXSESN  NODENAME  ORIGAREA
-------------------------------------------------------------------------------
88       BRANDSP   250     MYNODE  SUBSCRIBER USEDEFLT USEDEFLT N 3 UDP 7001 7002
```

Figure 110 shows sample configuration data for the SN.

Figure 110 Sample configuration data for standalone SN

```
BeginNode 103
  NodeName MYNODE
  NodeType SN

  BeginPool 88
    PoolName BRANDSP
    OrigType SUBSCRIBER
    MaxSessions 250
  EndPool

  BeginExternalNode 32
    NodeType Host
    IpAddr 47.192.6.254
  EndExternalNode
EndNode
```

Datafill for OSAC configuration

Figure 111 shows sample data for an OSAC configuration, including IP addresses and port numbers. (The example IP addresses are for illustration only.)

Figure 111 Example of simple OSAC topology

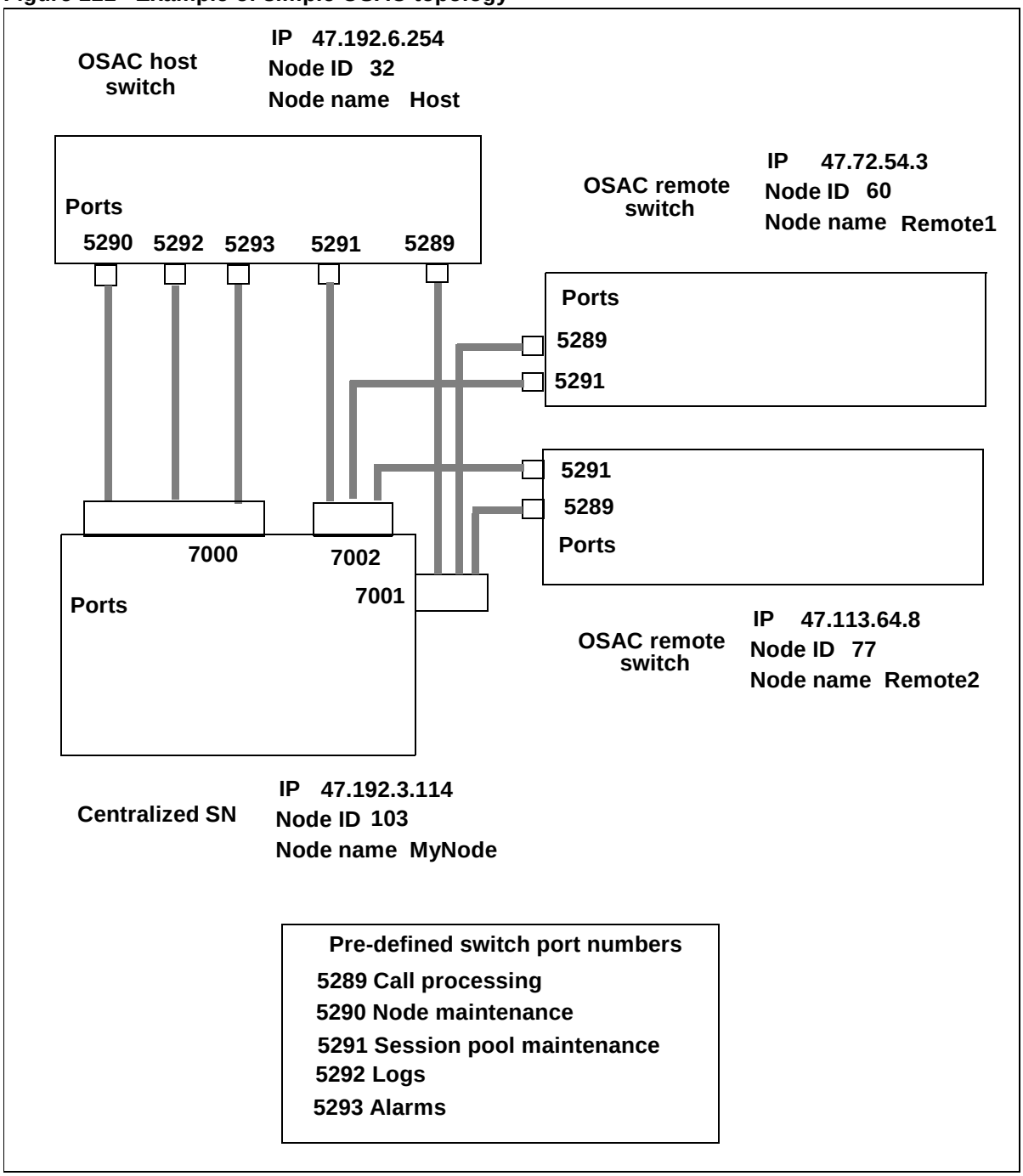


Figure 112 shows sample datafill for the node in tables OANODNAM, OANODINV, and OASESNPL at the OSAC host switch.

Figure 112 OSAC host switch datafill example for SN and session pool

OANODNAM

```
NODEID     NODENAME
-------------------------------------------------------------------------------
32         HOST
60         REMOTE1
77         REMOTE2
103         MYNODE
```

OANODINV

```
NODENAME    NODEAREA
-------------------------------------------------------------------------------
HOST      OSAC SELF 8 6 30
REMOTE1   OSAC OTHER 0 UDP IPV4 47 72 54 3 SWITCH 4 C 12 CITYB REMOTE1OSAC 8 6 30
REMOTE2   OSAC OTHER 0 UDP IPV4 47 113 64 8 SWITCH 2 B 11 CITYC REMOTE2OSAC 8 6 30
MYNODE   OSNM 1 UDP IPV4 47 192 3 114 7000 Y 2 60 240 60 SN 4 BB 3 CITYA BRANDING
   8 6 301
```

OASESNPL

```
SESNPLID  SESNPLNM  MAXSESN  NODENAME  ORIGAREA
-------------------------------------------------------------------------------
88       BRANDSP   250     MYNODE  SUBSCRIBER USEDEFLT USEDEFLT N 3 UDP 7001 7002
```

Figure 113 shows the sample configuration data for the SN in an OSAC configuration.

Figure 113 Sample configuration data for SN—OSAC

```
BeginNode 103
  NodeName MyNode
  NodeType SN

  BeginPool 88
    PoolName BrandSP
    OrigType Subscriber
    MaxSessions 250
  EndPool

  BeginExternalNode 32
    NodeType Host
    IpAddr 47.192.6.254
  EndExternalNode

  BeginExternalNode 60
    NodeType Remote
    IpAddr 47.72.96.3
  EndExternalNode

  BeginExternalNode 77
    NodeType Remote
    IpAddr 47.113.64.8
  EndExternalNode
EndNode
```

Additional datafill

The application developer is responsible for mapping the correct identifiers for OSSAIN functions, control lists, and logical voice channels. These identifiers are datafilled in the following DMS switch tables:

- OAFUNDEF (OSSAIN Function Definition)
- OACTLDEF (OSSAIN Control List Definition)
- OAVLMAP (OSSAIN Voice Link Mapping)

Note: For complete datafill information on these and other OSSAIN switch tables, refer to *OSSAIN User's Guide*, 297-8403-901.

Function identifiers

Table OAFUNDEF defines the functions (services) provided by the SN application. Each function has an identifier (FUNCID field). The application must map the function identifier to the correct service provided by the SN. This identifier is used in the header of all call processing operations.

Figure 114 shows two function IDs, one for an SN function (branding) and one for a team of operators.

Figure 114 Switch datafill example for functions

```
OAFUNDEF
FUNCID    FUNCNAME          FUNCAREA
-------------------------------------------------------------------------------------------
1         BRANDING          SN  TASERV  N N Y  CQ0
2         OPERATOR          TOPSOPER  OPER_TEAM
```

Control list identifiers

Table OACTLDEF defines all the control lists used by OSSAIN. The SN can use a control list to transfer a call to another node or to an operator. Each control list number (OACTLNUM field) is associated with a single function (datafilled in table OAFUNDEF). The application must map the correct control list number to a function it needs to transfer to. The control list number is specified in the Transfer to Control List Request operation.

Figure 115 shows two control list numbers. The switch uses control list 0 to route the call to the branding application. (The application does *not* use this control list number.) The application specifies control list 1 in the Transfer to Control List request to transfer the call to an operator.

Figure 115 Switch datafill example for control lists

```
OACTLDEF
OACTLNUM   OACTLNAM      NETWRKID    OAFUNCTS
-----------------------------------------------------------------
0          BRAND_CTL     3           (   BRANDING) $
1          OPER_CTL      4           (   OPERATOR) $
```

Voice link identifiers

Table OAVLMAP maps the logical voice channel numbers (NDANDCH field) to the actual voice circuits (CLLI field) associated with the SN. The application must supply the correct logical voice channel number in voice-related operations sent to the switch.

Figure 116 shows two logical voice channel numbers associated with the SN.

Figure 116 Switch datafill example for voice links

```
OAVLMAP
NDANCH            CLLI        EXTRKNUM   BCSTAREA
-----------------------------------------------------------------
MYNODE 1          OSSAINVL1   10         N
MYNODE 2          OSSAINVL2   11         N
```

Sample configuration files

This section shows examples of the following two configuration files:

- a simple configuration file that contains only the mandatory key-value pairs
- a full configuration file that contains all the optional key-value pairs as well as the mandatory ones

Simple configuration file

Figure 117 shows an example of a simple configuration file.

Figure 117 Example of simple configuration file

```
BeginNode 103

  BeginPool 88
    MaxSessions 250
    OrigType Subscriber
  EndPool

  BeginExternalNode 32
    IpAddr 47.192.6.254
  EndExternalNode

  BeginLogDevice Console
    BeginType Alarm
    EndType

    BeginType Report
      Level Detailed
    EndType

    BeginType Status
      Level Detailed
    EndType

    BeginType Swerr
    EndType
  EndLogDevice

EndNode
```

Note: Although the BeginLogDevice block is not mandatory, it is recommended that the SN always generate ALARM logs and SWERR logs.

Full configuration file

Figure 118 through Figure 124 show an example of a configuration file that contains all possible key-value pairs. The example includes code comments.

Figure 118 Example of full configuration file

```
# Configure an SN with node ID 10.
#
# The processing priority is set to 20, which means a maximum of 20
# messages are processed before checking the sockets for more messages.
# A maximum of 1000 messages can wait in the mailbox queue for
# processing.
#
# Node maintenance messages for this node use port 8000, and
# maintenance requests wait 15,000 milliseconds (15 seconds) before
# they time out. An MIS data port value is commented out and not used.
#
# Outgoing logs to the DMS are disabled, and only Major and Critical
# alarms are sent to the DMS.
#
# OAP message versions 2 and 3 are supported for node maintenance, and
# message validation is restricted to class headers and datablocks. No
# checking of operation headers or overall structure will be performed.
#
  BeginNode 10
    NodeName MyNode
    NodeType SN

    Priority 20
    MailboxSize 1000

    NodeMtcPort 8000
    RequestTimeout 15000
    # MISDataPort 6000

    DMSLog Off
    DMSAlarm On
    DMSAlarmLevel Major

    MinVersion 2.0
    MaxVersion 3.0

    Validation On
    ClassHeaderValidation On
    OpHeaderValidation    Off
    DatablockValidation   On
    OpStructureValidation Off
```

Figure 119 Example of full configuration file (continued)

```
# Configure a subscriber-originated session pool with up to 150
# simulataneous sessions. The session pool accepts OAP versions 3 and
# 4, and any session pool maintenance requests wait 15 seconds before
# timing out. No DMS logs or alarms are enabled.

    BeginPool 110
      PoolName "My Pool"

      MaxSessions 150
      OrigType Subscriber

      CallpPort   8001
      PoolMtcPort 8002
      RequestTimeout 15000

      DMSLog Off
      DMSAlarm Off

      MinVersion 3
      MaxVersion 4

# Call processing requests time out after only 3 seconds

      BeginCallp "My Application"
        RequestTimeout 3000
      EndCallp

# Configure user-defined data relevant to session pool 110.
# The application code can access the data in this way:
#
# OA_AbstractContext* cxt;
# char* favorite;
# int numQuestions;
#
# cxt->userData->getField( favorite, "FavoriteColor" );
# cxt->userData->getField( numQuestions, "NumberOfQuestions" );
#
# The value of 'favorite' and 'numQuestions' will be set to 'Blue'
# and 3, respectively, using this configuration file.

      BeginUserData
        FavoriteColor Blue
        NumberOfQuestions 3
      EndUserData

    EndPool  # End of data about session pool 110
```

Figure 120 Example of full configuration file (continued)

```
# Configure a service node-originated session pool (#120) with up to 24
# simulataneous sessions. The session pool accepts any OAP versions
# supported by the API, and any session pool maintenance requests
# wait the default 20 seconds before timing out. All DMS logs and
# alarms are enabled.
#
# Maintenance messages will use the same port as pool 110 (8002), but
# call processing messages use a different port (9000). Note that
# this is not required; the same local port can be used for all
# messaging, or a different port can be used for each function.

    BeginPool 120
      MaxSessions 24
      OrigType SN

      CallpPort 9000
      PoolMtcPort 8002

    EndPool

# Configure local ports for sending and receiving IP messages.
# The following ports are defined:
#
#   8000  Node maintenance with OSAC host
#   8001  Call processing with OSAC host
#   8002  Session pool maintenance with both OSAC host & remote
#   9000  Call processing with OSAC remote
#  10000  Communications with another SN
#  11000  Communications with another SN
#
# Callp with the OSAC host is given the highest priority (50). This
# means that as many as 50 messages will pulled off that socket before
# checking other sockets or processing the messages.
#
# Only port 8001 is set to block; the port will wait for as long as
# 500 milliseconds (.5 seconds) for a message to arrive before checking
# other ports or request timeouts.
```

Figure 121 Example of full configuration file (continued)

```
BeginPort 8000
  PortType NodeMtc
EndPort

BeginPort 8001
  PortType Callp
  Priority 50
  TimedBlocking 500
EndPort

BeginPort 8002
  PortType PoolMtc
EndPort

BeginPort 9000
  PortType Callp
EndPort

BeginPort 10000
  TransProtocol UDP
  AppProtocol   Text
EndPort

BeginPort 11000
  TransProtocol TCP
  AppProtocol   MyProtocol

  IsPassive     False
  NoDelay       True
  KeepAlive     True

  InMessageLength 212
  OutMessageLength 39
EndPort
```

Figure 122 Example of full configuration file (continued)

```
# Configure data for external nodes; OSAC host and 1 remote,
# other service node, and an administrative node (for OA&M).

  BeginExternalNode 200
    NodeType Host
    IpAddr 47.192.6.254
  EndExternalNode

  BeginExternalNode 300
    NodeType Remote
    IpAddr 47.72.96.3
  EndExternalNode

  BeginExternalNode 400
    NodeType SN
    IpAddr 47.192.113.8

    # If node goes down, all session pool are updated
    #
    StatusPerPool No

    BeginAddr 20
      Port 888
      TransProtocol UDP
      AppProtocol   Text
    EndAddr

    BeginAddr 21
      Port 9000
      TransProtocol TCP
      AppProtocol   MyProtocol
    EndAddr

  EndExternalNode

  BeginExternalNode 500
    NodeType Admin
    IpAddr  47.192.12.56
  EndExternalNode
```

Figure 123 Example of full configuration file (continued)

```
# Configure logs so that critical Alarms and Swerrs go to standard
# error, and Brief Status logs go to standard out. Alarms, Swerrs,
# and all Status logs are copied to a 'trouble' file, and all Report
# logs are copied to a 'report' file.
#
# Also, application-specific logs go to an application-specific
# log device.

  BeginLogDevice Console
    Stream stderr

    BeginLogType Alarm
      Level Critical
    EndLogType

    BeginLogType Swerr
    EndLogType
  EndLogDevice

  BeginLogDevice Console
    Stream stdout

    BeginLogType Status
      Level Brief
    EndLogType
  EndLogDevice

  BeginLogDevice File
    FileName "trouble.txt"

    BeginLogType Alarm
    EndLogType

    BeginLogType Swerr
    EndLogType

    BeginLogType Status
      Level Detailed
    EndLogType
  EndLogDevice

  BeginLogDevice File
    FileName "report.txt"
    MaxLogsInFile 20
    NumFiles 10

    BeginLogType Report
      Level Detailed
    EndLogType
  EndLogDevice
```

Figure 124 Example of full configuration file (continued)

```
# Configure user-defined data available to the node and all session
# pools:

  BeginUserData
    TypeOfSwallow European
    SwallowMigration False
    CoconutWeight 10
  EndUserData
EndNode
```

Engineering considerations

This section discusses several considerations for engineering the application.

Node processing priority and transport priority

For node processing, the priority value (in the BeginNode block) determines how many messages the node processes before polling the transport for more messages. If this value is too low, the time spent polling the transport increases. If the value is too high, the risk of transport buffer overflow increases. The proper value for node priority varies across platforms and types of transports. Typical values are 10 to 100. The default value is 50.

For transports, the priority value (in the BeginPort block) determines how many messages are received before processing them. If this value is too low, the risk of overflowing the transport buffer increases. If this value is too high, the time spent polling the transport increases. The proper value for transport priority varies across platforms and types of transports. Typical values are 10 to 100 for call processing ports and 1 to 10 for maintenance ports. The default value is 10.

Request timeout for call processing and maintenance

The request timeout value (in the BeginNode and BeginPool blocks) determine how long the node or session pool should wait for the response to a maintenance request before assuming that a message is lost. The default value is 20000 milliseconds (20 seconds).

The request timeout value (in the BeginCallp block) determines how long a session should wait for the response to a call processing request before assuming that a message is lost. The default value is 5000 milliseconds (5 seconds).

Timed blocking

The timed blocking value specifies how much time is spent polling a transport while no messages are available. A value of 0 specifies no blocking. A positive value prevents inactive nodes from spending excessive CPU cycles frequently checking empty transports. No more than one transport should have a non-zero blocking value. The default value is 0.

Scaling the application

While services can be deployed with a single DMS switch and a single SN, it may be necessary or cost-effective to deploy the service with multiple SNs or switches. The SN should be tested with heavy messaging throughput to determine its capacity. If a single SN cannot support the anticipated call traffic, more than one SN should be deployed for the service. Or, a hardware platform with more processing power should be selected for the SN application.

When a service is deployed to a large region, it may be desirable to use a centralized (OSAC) configuration. OSAC creates a network with multiple DMS switches that can access the same SN. The OSAC configuration is transparent to the SN application.

Starting the application

For a standalone application that does not change the configure() method of the framework, the application is started on the command line with the configuration filename as the only argument. Figure 125 shows an example.

Figure 125 Starting the application

```
myapp myapp.cfg
```

Note: There is no restriction on the name of the configuration file.

Appendix: OSSAIN OAP Simulator

The OAP Simulator is a multi-function tool that provides interactive and call traffic testing capabilities using the Open Automated Protocol (OAP). Its purpose is to facilitate the testing of OSSAIN service nodes (SN) and TOPS/ OSSAIN switches. The simulator can be configured to simulate an SN or a switch; and it can be configured to handle maintenance processing automatically or interactively.

This appendix provides guidelines on using the OAP Simulator. It is intended for two types of users: developers of OSSAIN SN applications and personnel involved in OSSAIN switch design and testing.

The OAP Simulator is driven by a database of OAP data blocks and operation scenario files. The user of the simulator must have a working knowledge of the OAP protocol in order to correctly process OAP messages. Complete information on the OAP protocol is provided in *OSSAIN Open Automated Protocol Specification*, Q235-1.

The discussion of the OAP Simulator focuses on the following areas:

Functionality overview

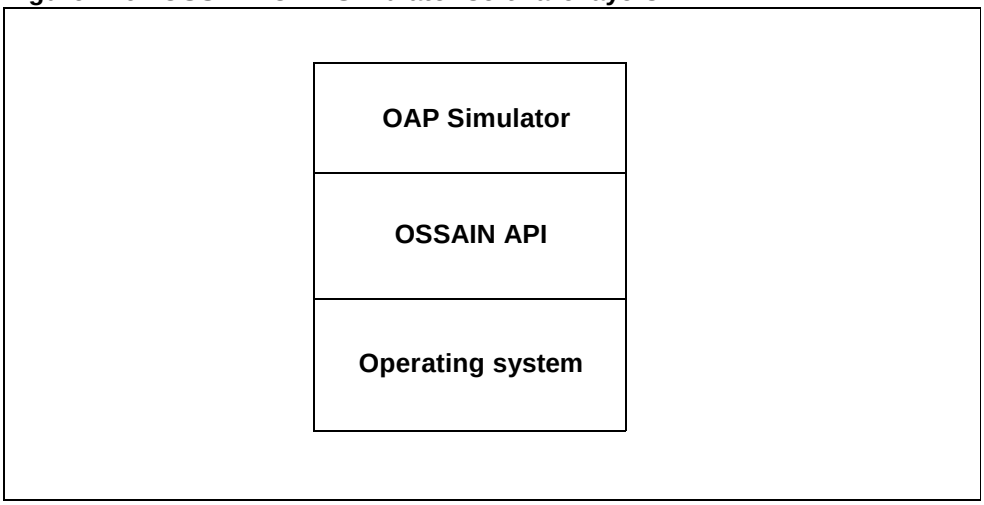
This section describes the following OAP Simulator functions:

- software architecture
- modes of operation
- context
- process model
- data model
- installation

Software architecture

The OAP Simulator is an OSSAIN API application. The simulator is built on top of the OSSAIN API and uses API utilities and interfaces to simulate OSSAIN SNs and OSSAIN DMS switches. The simulator is portable to any operating system supported by the OSSAIN API. Figure 126 shows the software layers.

Figure 126 OSSAIN OAP Simulator software layers



The OAP Simulator is provided as C++ source code with required make files. This allows the user to re-build the simulator after incorporating the required custom changes to the API (such as adding user-defined operations for node to node communications).

Modes of operation

The OAP Simulator can operate in three different modes, depending on the testing requirements of the user. The modes are described in the following paragraphs.

Interactive call processing mode

In this mode, the simulator provides a series of interactive menus from which the user controls OAP call control messaging. This messaging may be from a TOPS/OSSAIN switch perspective or from an SN perspective. The simulator supports all OAP operations so that the role of a switch or an SN can be fully simulated.

When simulating an SN in this mode, the simulator automatically responds to all OAP maintenance request messages. This allows the user to focus on call control messaging and not be interrupted by OAP maintenance request messaging.

Interactive call processing and maintenance mode

In this mode, the simulator provides a series of interactive menus from which the user controls OAP call control and maintenance messaging. The user controls all OAP messaging from the interactive menus. Refer to "Menus" on page 230 for details on the interactive menus.

The simulator can be configured so that only a portion of maintenance messaging is handled interactively. For example, it can be configured so that node maintenance messaging is handled interactively, while session pool maintenance is handled automatically.

The method by which OAP maintenance messages are processed is determined by the simulator configuration file. Refer to "Configuration files" on page 217 for a description.

Scripting mode

In this mode, the simulator allows multiple script files to be loaded. With script files, call traffic can be simulated by indicating that a script is to be executed multiple times, which simulates the processing of many calls. Scripts provide a way to execute entire call processing and/or maintenance message flows. Refer to "Scripting" on page 243 for a complete description of simulator scripting capabilities.

Context

Understanding the concept of *context* is important when working with the OAP Simulator. Context refers to the unique set of data associated with a particular message flow. When an OAP message is processed by the simulator, it is done so under a specific context.

Three levels of context are provided by the simulator. They model the OSSAIN concepts of node, session pool, and session.

Node context

All OAP node class messages are processed under the node context. A *single* node context is provided by the simulator, because only one node can be simulated at a time.

Session pool context

OAP session pool class messages are processed under a session pool context. *Multiple* session pool contexts can be configured.

When processing session pool maintenance messages interactively, the session pool context used to process a session pool class message is determined by setting the simulator's foreground session pool. The foreground session pool can be changed at any time using the simulator's main menu. This allows the user to set the session pool context used when working with the interactive session pool menus.

The foreground session pool is automatically changed when an incoming session pool class message is received that has a different session pool identifier than what is currently set for the foreground session pool. This allows the user to quickly process the maintenance message, which facilitates responding to timed activities such as an audit request.

Session context

OAP call processing class messages are processed under a *specific* session context. A session context is uniquely identified by the foreground session pool and session identifier.

Note: The combination of session pool and session identifier uniquely identifies a call processing session.

Like the session pool context, the call processing session context can be changed at any time using the simulator's main menu. Also, the foreground session can be changed when a call processing message for a session other than the one being currently worked on is received. The user has the option of not switching the session context when this occurs. If the user chooses not to switch the session context when an incoming message is received, the context associated with the incoming message is updated based on the incoming message, but the foreground session remains unchanged.

Process model

The OAP Simulator allows for a variety of testing options. It provides an interactive process (SIMINT) as well as an automated maintenance process (SIMAUTO).

Interactive process

The interactive process provides menus that allow a user to drive call processing and maintenance actions interactively. All OAP operations are supported so that a user can simulate an SN or a DMS switch.

The simulator keeps track of the context under which an OAP message is processed. The interactive process can be configured with one node, multiple session pools, and multiple session contexts.

When an incoming message is received, it is routed to the correct interactive context for processing. Users can switch between different contexts at any time. By doing so, the user has complete control of whether they work with node level maintenance, session pool level maintenance, or one of many call sessions.

Note: Incoming maintenance messages cause the simulator to automatically switch to the context required to process the maintenance message. This facilitates the processing of the maintenance message in a timely way, which is necessary under certain conditions (such as responding to an audit message).

The simulator also has the ability to operate in a centralized OSSAIN (OSAC) environment. When simulating an SN, the simulator is able to process calls originating from multiple DMS switches.

Automated maintenance process

The automated maintenance process simulates node and session pool maintenance by *automatically* responding to node and session pool maintenance requests originated from a DMS switch. This process frees the user from having to periodically interrupt call processing testing to deal with maintenance messaging. Instead, the user can use the interactive process to focus on call processing testing.

Note: SN developers can achieve a similar functionality by using scripting to simulate a DMS switch. Refer to “Scripting” on page 243 for more information.

The use of multiple simulator processes to achieve multiple testing configurations requires the simulator to be highly configurable. This high degree of configuration is achieved through the use of simulator configuration files. Refer to “Configuration files” on page 217 for an overview of the configuration file format.

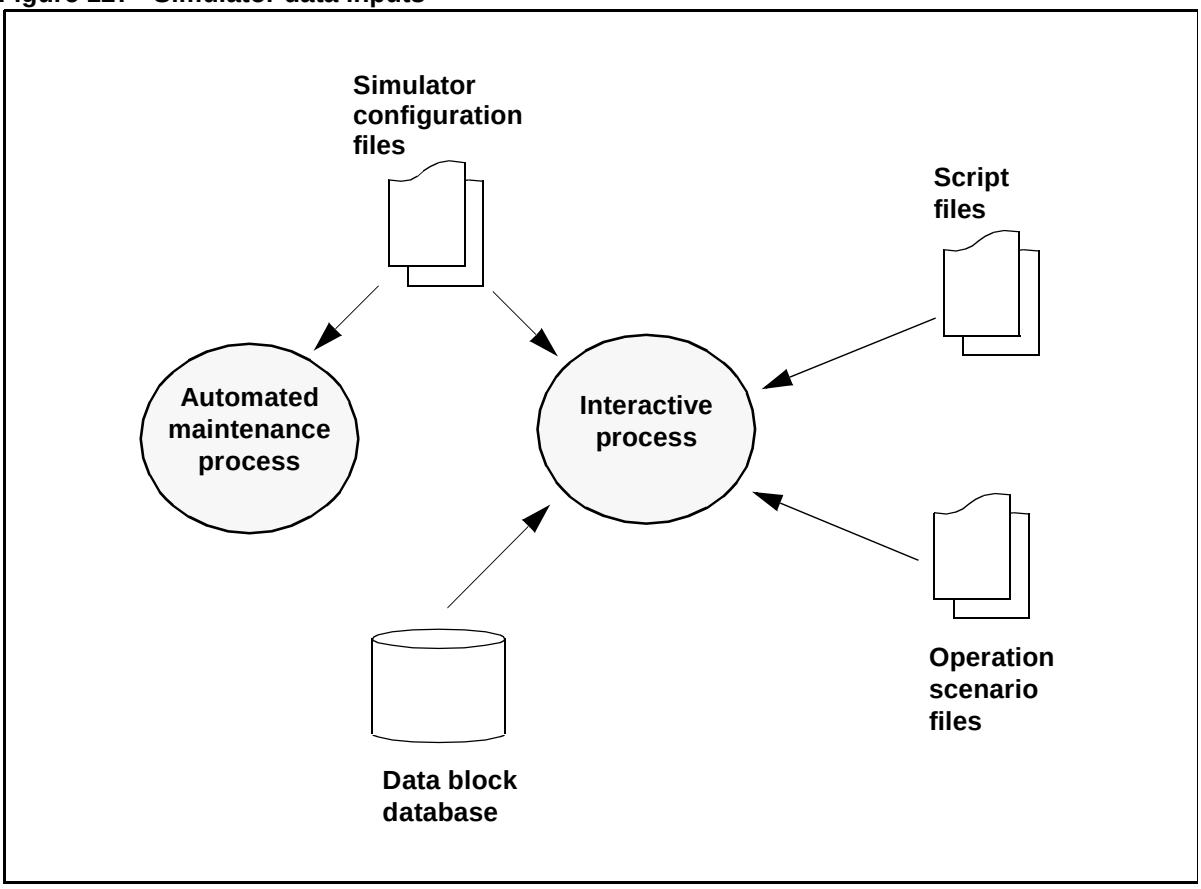
Data model

Several data inputs are used by the OAP Simulator to achieve a flexible, adaptable testing environment. This section provides an overview of these data inputs:

- configuration files
- operation scenario files
- data block database
- script files

Figure 127 shows the data inputs.

Figure 127 Simulator data inputs



Configuration files

The OAP Simulator can use a configuration file to facilitate a variety of testing options. The configuration file defines the node and session pools to be simulated. This definition includes node and session pool identifiers, data communications port numbers, and number of sessions to be supported.

The configuration file also defines the external nodes the simulator interacts with, and the type of alarms and logs the simulator generates during execution. The simulator's interactive and automated maintenance processes use the same configuration file.

Simulator configuration files follow the same format as OSSAIN API configuration files, with the addition of simulator-specific fields. Refer to "Configuration files" on page 217 for details on configuration file format.

Operation scenario files

Operation scenario files provide data needed by the simulator to build OAP messages. One scenario file exists for each OAP operation (such as SessionBeginInform, VoiceConnectRequest, and SessionPoolRtsRetres).

Each file contains a list of scenarios, which is a collection of data block names to be included in the OAP message that is built for the operation scenario. The data block information is found in the data block database as a hexadecimal data stream.

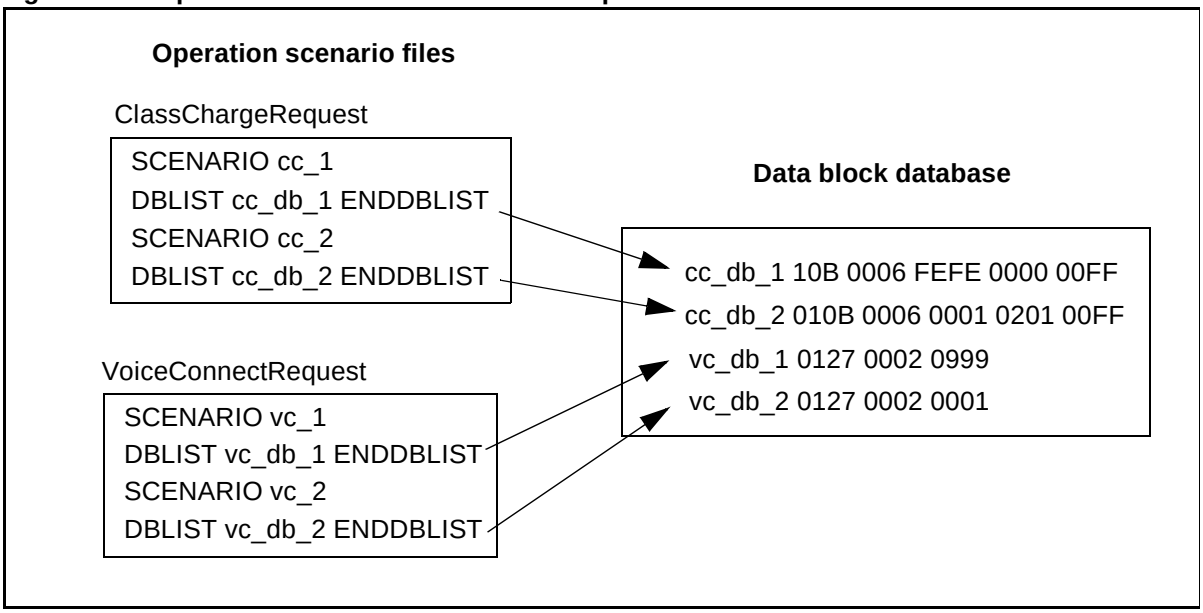
Users create operation scenario file entries by using any text editor or by using the operation editor found in the interactive simulator. Refer to "Database files" on page 225 for details on operation scenario files.

Data block database

The data block database is a collection of OAP data block names followed by the hexadecimal data bytestream that makes up the individual data block. There is one entry in the database for each data block identified in the collection of operation scenario files.

Figure 128 shows the relationship between operation scenario files and the data block database.

Figure 128 Operation scenario files relationship to the data block database



Data block information is read by the simulator when the user requests the simulator to load an existing data block or operation.

Users create data block database entries by using any text editor or by using the data block editor found in the interactive simulator. Refer to "Database files" on page 225 for details on the data block database.

Script files

Script files provide message flow scripts for use in script testing. Use of scripts allows for simulation of an entire message flow without the need for user intervention. Message flows can be repeated any number of times on multiple contexts. Through use of scripts, a user can simulate multiple calls on multiple sessions or maintenance message flows.

Users create script files by using a scripting language defined for the simulator. Script files are created using any text editor. Refer to "Scripting" on page 243 for details on script files.

Installation

The following files must be installed to run the OAP Simulator.

- SIMINT, which is the simulator interactive executable.
- SIMAUTO, which is the automated maintenance executable.
- SIMDATA.TXT, which is the default simulator configuration file. This file must be located in the working directory from which the simulator is executed. The default configuration filename can be specified using a command line argument when invoking the simulator.

- Operation scenario files, which provide the operation scenarios to be used during simulator testing. These files must be located in the working directory specified in the configuration file.
- Data block database, which provides the data block bytestream of data blocks to be used during the simulator testing. The data block database must be located in the working directory specified in the configuration file.

Configuration files

The OAP Simulator requires a configuration file to define the parameters within which the simulator operates. The information provided in the configuration file must match the datafill provided to nodes that interact with the simulator. This information includes:

- node identifier
- session pool identifiers
- session identifiers
- communication port definitions
- log configuration
- external node definitions

The OAP Simulator uses the standard configuration file format supported by the OSSAIN API. For details, refer to Chapter 9: "Configuration and administration."

This section discusses additional simulator-specific information provided in the configuration file.

Simulator-specific data fields

Additional simulator-specific fields are required to provide data file path information and interactive maintenance behavior. This information is defined in simulator configuration files using BeginUserData and EndUserData tags.

The following simulator-specific fields are supported:

- SIMULATOR_WORKING_DIRECTORY

  Defined in the node block, the simulator working directory tag specifies the directory path where operation scenario files and data block database files are located.

  Note: When working with multiple versions of the OAP protocol, it is recommended that a separate working directory be set up for each version of the protocol.

- SIMULATOR_DATABLOCK_FILE

  Defined in the node block, the simulator data block file tag specifies the name of the data block database file to be used. The file must be located in the directory specified by the simulator working directory tag.

- INTERACTIVE_MTC

  Defined in the node and/or session pool block, the interactive maintenance tag indicates that the interactive process will handle the maintenance messages for the node or session pool where the interactive maintenance tag is defined.

- FUNCTION_IDENTIFIER

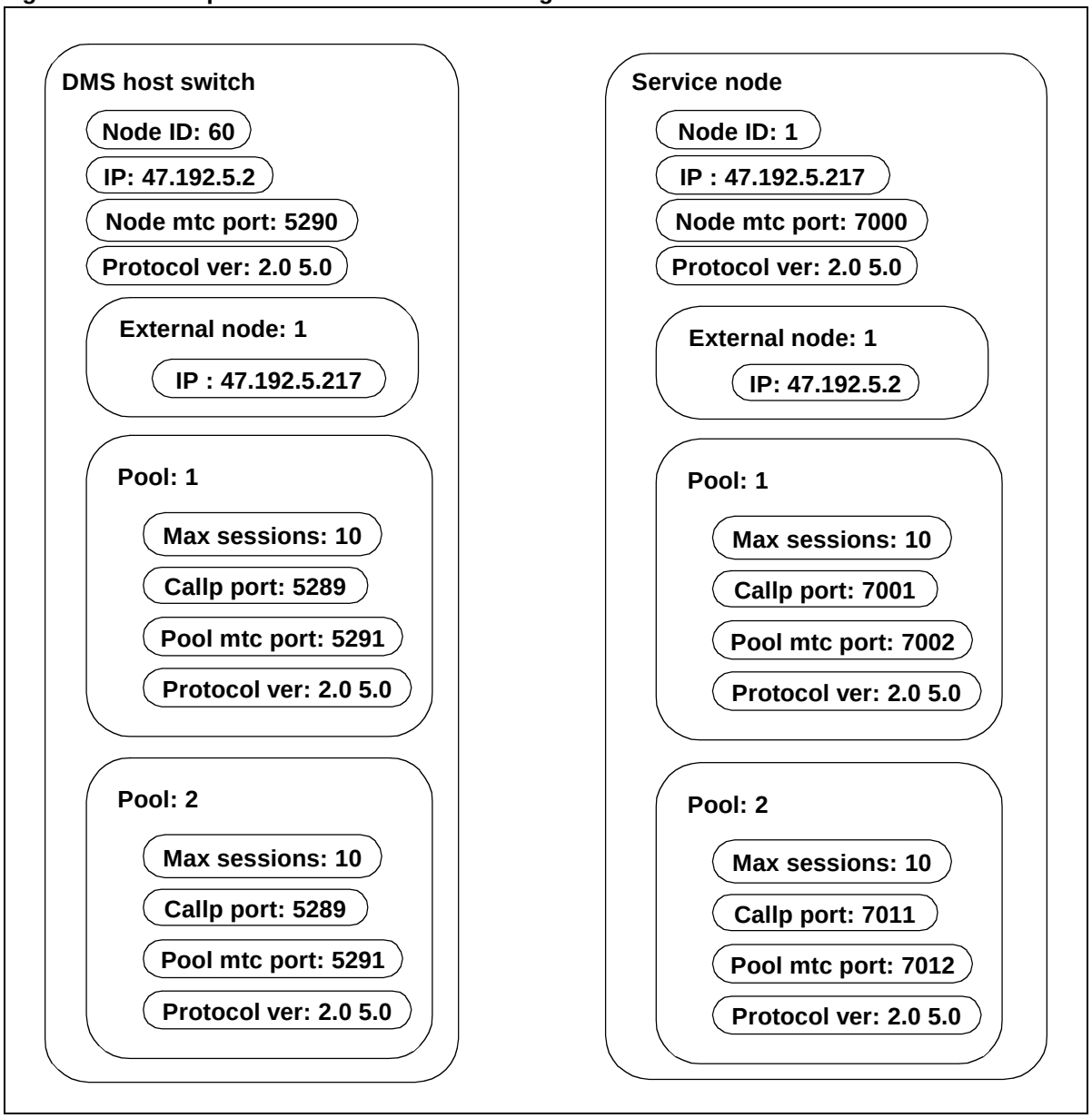
  Defined in the session pool block, the function identifier tag specifies the default function identifier to be used by the simulator for the session pool where the function identifier tag is defined.

Example configuration files

This section provides examples of simulator configuration files from a switch perspective and from an SN perspective. It is important to note that datafill on the DMS switch and SN *must match* for the OAP Simulator to operate correctly.

Figure 129 shows the switch and SN configuration used in the example configuration files.

Figure 129 Example DMS switch and SN configuration



Switch configuration file

Figure 130 and Figure 131 show an example of a configuration file used to simulate a DMS host switch. A description of the file follows the figure.

Figure 130 Simulator configuration file example—DMS switch

```
##################################################
#
# OAP Simulator configuration file
#
# DMS Switch
#
##################################################

BeginNode 60
  NodeName DMS Switch
  NodeType Host
  NodeMtcPort 5290
  Validation On

  BeginUserData
    Simulator_Working_Directory /fullpath/working/dir/
    Simulator_Datablock_File oap.dbs
  EndUserData

  BeginPool 1
    PoolName Application1
    OrigType Subscriber
    MaxSessions 10
    MinVersion 2.0
    MaxVersion 5.0
    CallpPort 5289
    PoolMtcPort 5291

    BeginUserData
      Function_Identifier 1
    EndUserData
  EndPool

  BeginPool 2
    PoolName Application2
    OrigType SN
    MaxSessions 10
    MinVersion 2.0
    MaxVersion 5.0
    CallpPort 5289
    PoolMtcPort 5291

    BeginUserData
      Function_Identifier 2
    EndUserData
  EndPool

  BeginExternalNode 1
    NodeName Service_Node_1
    NodeType SN
    IpAddr 47.192.5.217

  EndExternalNode
```

Figure 131 Simulator configuration file example—DMS switch (continued)

```
    BeginLogDevice Console
     Stream stdout

     BeginLogType Alarm
       Level Minor
     EndLogType

     BeginLogType Report
       Level Detailed
     EndLogType

     BeginLogType Status
       Level Detailed
     EndLogType

     BeginLogType Swerr
     EndLogType

   EndLogDevice

EndNode
```

- BeginNode block

  As with all OSSAIN API configuration files, this file begins with a BeginNode block. This block defines the node that is being simulated. In this example, the node identifier is set to 60 and the node type is Host for the DMS host switch. The node maintenance UDP port is set to 5290 and OAP protocol validation is enabled.

- BeginUserData block

  A BeginUserData block defines the working directory file path and the filename of the data block database. This block also defines whether the interactive maintenance process is used by the simulator. In this example, node maintenance is handled by the simulator's automated process instead, because the Interactive_Mtc line has been omitted.

- BeginPool blocks

  Two session pools are defined using BeginPool blocks. The first session pool, Application1, is a subscriber-originated session pool with 10 sessions. A default function identifier of 1 is defined in a BeginUserData block for the Application1 session pool.

  The second session pool, Application2, is an SN-originated session pool with 10 sessions. A default function identifier of 2 is defined in a BeginUserData block for the Application2 session pool.

  Both session pools can handle OAP versions 2.0 to 5.0 and both use UDP ports 5289 for call processing messaging and ports 5291 for session pool maintenance messaging.

- BeginExternalNode block

  A single external node is defined using a BeginExternalNode block. The node identifier of the external node is 1, the node type is SN, and the IP address of the node is 47.192.5.217.

- BeginLogDevice block

  In a BeginLogDevice block, logs are configured to send alarms, report logs, status logs, and swerr logs to standard output.

SN configuration file

Figure 132 and Figure 133 shows an example of a configuration file used to simulate an SN. A description of the file follows the figure.

Figure 132 Simulator configuration file example—SN

```
################################################
#
# OAP Simulator configuration file
#
# Service Node
#
################################################

BeginNode 1
  NodeName Service_Node_1
  NodeType SN
  NodeMtcPort 7000
  Validation On

  BeginUserData
    Interactive_Mtc
    Simulator_Working_Directory /fullpath/working/dir/
    Simulator_Datablock_File oap.dbs
  EndUserData

  BeginPool 1
    PoolName Application1
    OrigType Subscriber
    MaxSessions 10
    MinVersion 2.0
    MaxVersion 5.0
    CallpPort 7001
    PoolMtcPort 7002

    BeginUserData
      Interactive_Mtc
      Function_Identifier 1
    EndUserData
  EndPool

  BeginPool 2
    PoolName Application2
    OrigType SN
    MaxSessions 10
    MinVersion 2.0
    MaxVersion 5.0
    CallpPort 7011
    PoolMtcPort 7012

    BeginUserData
      Interactive_Mtc
      Function_Identifier 2
    EndUserData
  EndPool

  BeginExternalNode 60
    NodeName DMS Switch
    NodeType Host
    IpAddr 47.192.5.2

 EndExternalNode
```

Figure 133 Simulator configuration file example—SN 9continued)

```
  BeginLogDevice Console
    Stream stdout

    BeginLogType Alarm
      Level Minor
    EndLogType

    BeginLogType Report
      Level Detailed
    EndLogType

    BeginLogType Status
      Level Detailed
    EndLogType

    BeginLogType Swerr
    EndLogType

  EndLogDevice

EndNode
```

- BeginNode block

  In this example, the SN is the node being simulated. A BeginNode block defines the node identifier as 1 and the node type as SN. The node maintenance UDP port is set to 7000 and OAP protocol validation is enabled.

- BeginUserData block

  A BeginUserData block is used to define the working directory file path and the filename of the data block database. In this example, the interactive maintenance process is used by the simulator, because the Interactive_Mtc line is included. With interactive maintenance, the user is required to respond to all node maintenance requests by the simulator's interactive user interface.

- BeginPool blocks

  Two session pools are defined using BeginPool blocks. The first session pool, Application1, is a subscriber-originated session pool with 10 sessions. A default function identifier of 1 is defined in a BeginUserData block for the Application1 session pool. Application1 uses UDP port 7001 for call processing messaging and port 7002 for session pool maintenance messaging.

  The second session pool is an SN-originated session pool with 10 sessions. A default function identifier of 2 is defined in a BeginUserData block for the Application2 session pool. Application2 uses UDP port 7011 for call processing messaging and port 7012 for session pool maintenance messaging.

  Both session pools can handle OAP versions 2.0 to 5.0.

- BeginExternalNode

  A single external node is defined using a BeginExternalNode block. The node identifier of the external node is 60, the node type is Host for the DMS switch, and the IP address of the node is 47.192.5.2.

- BeginLogDevice block

  In a BeginLogDevice block, logs are configured to send alarms, report logs, status logs, and swerr logs to standard output.

Database files

The OAP Simulator can create OAP operations from the OAP specification with default values or it can use pre-built operations that are loaded from disk. The data used to load operations from disk are broken into two sources: the data block database and operation scenario files.

Data block database

The data block database contains a collection of OAP data blocks in hexadecimal format created by the user. Each line in the data block database file specifies a data block name and the hexadecimal data bytestream that makes up the data block. Figure 134 shows an example of a voice channel data block from a voice connect request operation.

Figure 134 Example data block from oap.dbs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;      vc1_VoiceChannel
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


VoiceChannel_vc1   0127 0002 0000
```

The hexadecimal data specification for a data block can be added to the data block database file either manually using a text editor, or using the simulator's data block editor. The data block editor allows the user to create a data block from the OAP specification initialized to default values. The user can then modify the individual fields of the data block, validate the data block, and save the data block to disk.

Note: Data block names must be *unique* within the data block database. When creating data blocks manually or from the simulator's data block editor, be sure the data block name is unique. If not, the simulator will discard redundant data blocks when loading the data block database.

The simulator performs file maintenance on the data block database while the interactive process is used. At the time the data block database is loaded into the simulator, any redundant data blocks are resolved.

When the user exits the interactive menu, the data block database file is rewritten from the simulator's core memory. This includes any modifications the user has made to existing data blocks and any new data blocks. A destructive write is performed on the existing data block database with the new data block data.

Operation scenario files

Operation scenario files provide pre-built OAP operations. A file exists for each OAP operation (for example, SessionBeginInform, VoiceConnectRequest). A list of supported operation names can be displayed using the simulator operation editor.

Each operation scenario file has a collection of scenario blocks for the OAP operation. A scenario block begins with the tag field SCENARIO and a unique scenario name. The next line is the tag field DBLIST followed by the data block names that are to be added to the operation. The data block list ends with the tag field ENDDBLIST. Figure 135 shows an example entry from an operation scenario file.

Figure 135 Example operation scenario file entry

```
#
SCENARIO vc1
DBLIST   VoiceChannel_vc1 END_DBLIST
#
SCENARIO vc2
DBLIST   VoiceChannel_vc2 ConfId_vc2 END_DBLIST
#
SCENARIO vc3
DBLIST   END_DBLIST
#
SCENARIO vc4
DBLIST   VoiceChannel_vc4 ConfId_vc4  END_DBLIST
#
SCENARIO vc5
DBLIST   VoiceChannel_vc5 ConfId_vc5  END_DBLIST
```

When directed to load an operation from disk, the simulator displays the list of supported OAP operations for the user to choose from. The simulator displays the scenario file for the selected operation, which displays the scenario names and included data block names. Once the user selects a scenario name, the simulator builds the operation from the specified data block names using the data from the data block database. Operation scenario file entries can by created using a text editor or using the simulator's operation editor.

As with data block names, the scenario names in an operation scenario file must be unique. When the simulator's operation editor is used to save an operation to file, an entry is made in the appropriate operation scenario file. If the operation scenario file does not exist, it is created.

Entries are also made in the data block database for each data block found in the operation. The data block name is a concatenation of the OAP data block name and the scenario name.

In Figure 135, scenario vc1 has a single voice channel data block. The data block name built for this scenario is VoiceChannel_vc1 (VoiceChannel for the OAP data block name and vc1 for the scenario name). If a data block already exists in the data block database with the same name as the one being saved, the existing data block is *overwritten*.

Multiple user environment

The OAP simulator currently does not support sharing of the data block database and operation scenario files. This is primarily due to the destructive write performed on the data block database by the simulator during normal operation. Although a central copy of these files can be used by many users for read-only purposes, write protection is not provided by the simulator.

If a multiple user environment is required, it is recommended that users access private copies of simulator operation and data block files for writing. New operation scenarios and data blocks can then be manually merged into the groups master copies of the simulator files for read-only access.

File location

The default name for the data block database is oap.dbs unless otherwise specified in the simulator configuration file. The default path for the data block database and operation scenario files is the current working directory unless otherwise specified in the simulator configuration file.

Invoking the simulator

This section describes the two methods for starting the OAP Simulator:

- invoking the simulator executable files directly
- invoking the simulator by script

Invoking the executable files

Two simulator executable files are provided: SIMINT is the simulator interactive process and SIMAUTO is the simulator automated maintenance process.

Depending on your testing needs, you may invoke one or both of these executables. Each file is invoked by entering the executable's filename (for example, SIMINT) at a command line.

Each executable can take an optional command line argument. This argument specifies the configuration filename for the simulator process being invoked. The working directory is assumed for the location of the configuration file. If the configuration file is not provided in the same directory where the executable is invoked, the full path name of the configuration file must be specified. If the configuration filename is not provided as an argument to each of the simulator executables, the default configuration filename, simdata, is used.

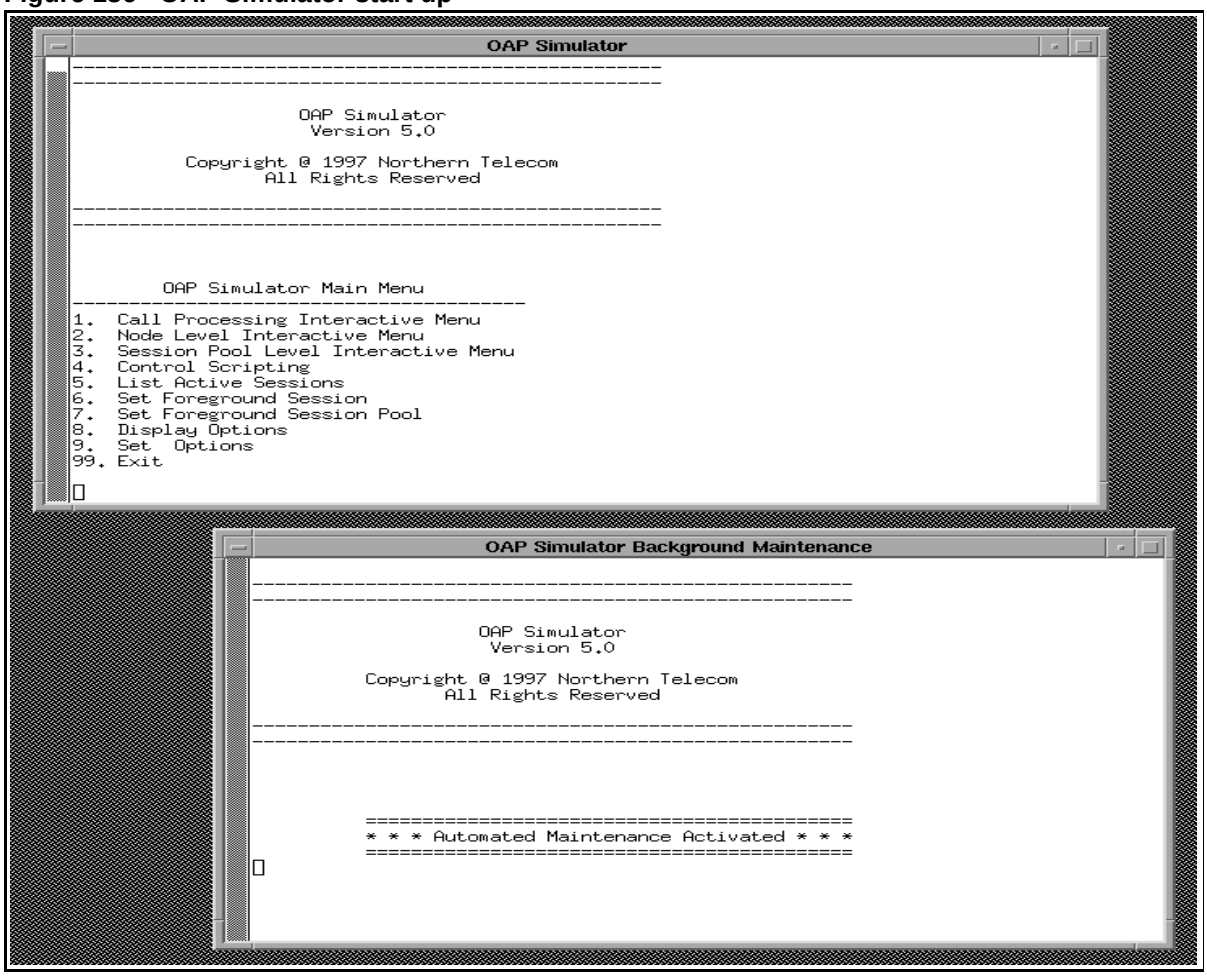
Invoking by script

A script file is provided for each supported platform (HP-UX 9.5 and NT 4.0) that spins off the SIMINT and SIMAUTO processes. An xterm window is created for each process and titled appropriately. This script file is intended primarily for users who want to simulate an SN and prefer to have DMS switch maintenance requests responded to automatically.

The default script file does not specify the configuration filename as a command line argument to the SIMINT and SIMAUTO executables, so the default configuration filename (simdata) is used. Users can edit the script file to specify their own configuration file as an argument to the SIMINT and SIMAUTO executables. If the full path name of the configuration file is not specified, the configuration file must be located in the directory from which the script is invoked.

Figure 136 shows an example of the two processes started by the simulator script file. The interactive process runs under the window titled "OAP Simulator." This is where the user performs interactive testing. The automatic maintenance process runs under the window titled "OAP Simulator Background Maintenance." No user actions are performed from this window because it automatically handles maintenance messages for the simulation. If message tracing is enabled in the configuration file, this window displays incoming and outgoing maintenance messages.

Figure 136 OAP Simulator start up



Which method to use

Consider what type of simulation you are attempting. If you want to simulate an SN and do not want to respond to switch-originated maintenance requests, start the simulator using the script file. If you want to simulate a DMS switch or do not want automated maintenance responses, invoke the interactive process directly.

There are several ways to configure and use the simulator, including running multiple instances of a simulator process. The simulator is designed to be configurable for a variety of testing needs.

Menus

The OAP simulator's interactive process is driven by the following five text-based menus:

- Main Menu
- Interactive Menu
- Operation Editor Menu
- Datablock Menu
- Scripting Control Menu

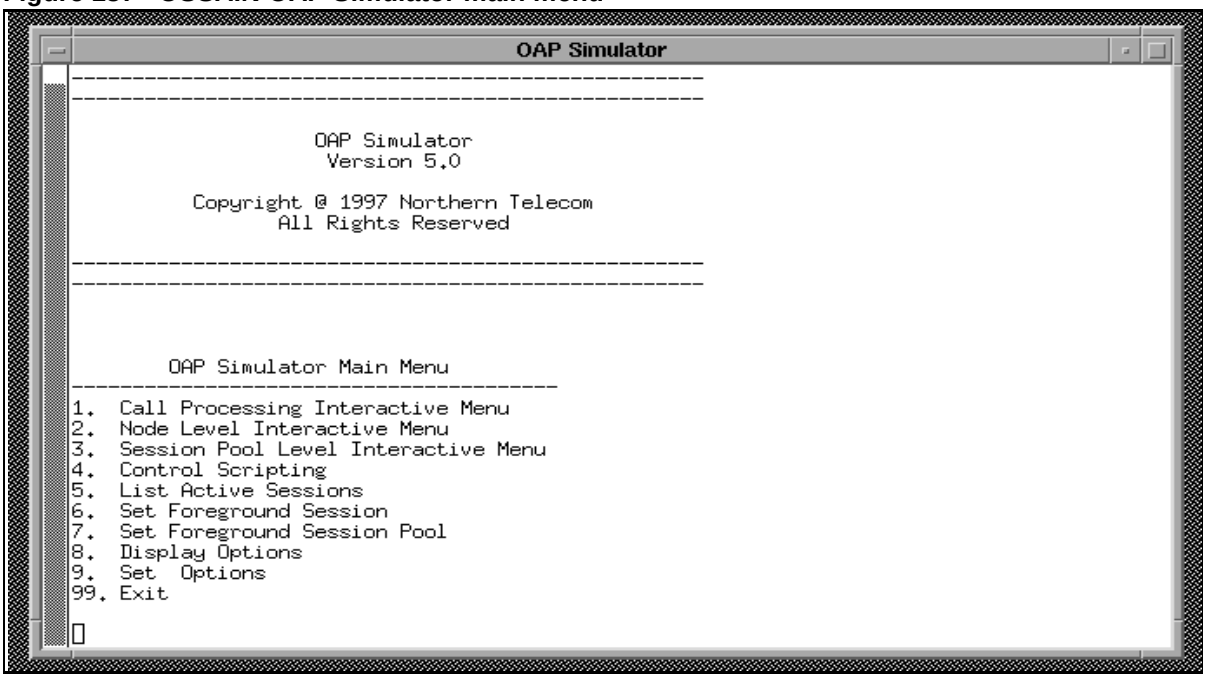
This section describes the functions of each menu.

Main Menu

The OAP Simulator Main Menu provides links to the interactive menus for interactive call processing, interactive node maintenance, interactive session pool maintenance, and scripting control. The Main Menu also lists sessions that are active currently (sessions that are processing calls). The Main Menu is used to set the foreground session and session pool, and to set general options.

Figure 137 shows an example of the Main Menu.

Figure 137 OSSAIN OAP Simulator main menu



Context

The OAP simulator maintains a separate context for each session, session pool, and node configured for the simulator. A context provides data specific to the call, session pool, or node for which it belongs. At a session level, the context can be thought of as a callp process.

The simulator provides the ability to switch between different contexts. This allows a user to service multiple calls at virtually the same time. These calls can be a mix of subscriber-originated calls or SN-originated calls. Calls can be with multiple switches or nodes. The user can also switch to the node maintenance context or any of the configured session pool maintenance contexts at any time.

If the user is waiting for an incoming message and receives a message that must be processed under another context, the user will be prompted to switch context. If the incoming message is a node or session pool maintenance message, the context is switched *automatically*. This facilitates the processing of maintenance messages.

The user can switch context back to the original session context and resume working where the interruption occurred. Maintenance messages are *only* received by the interactive process if the configuration file indicates that interactive maintenance processing is desired. Otherwise, all node and session pool messages are handled by the automated maintenance process.

The main menu provides selections for setting foreground session pool and session context. These specify the context to use when interactive call processing or interactive session pool maintenance options are selected from the main menu.

Setting Options

The following general options can be set from the main menu:

- Perform protocol negotiation—This option specifies whether OAP protocol negotiation is performed or not. In general, this option should be set to yes (Y). If not set to Y, the user is required to send the correct series of OAP reject messages to negotiate the OAP protocol version to use during testing. The default for this option is Y.
- Wait for OAP message after send—This option specifies whether the simulator automatically waits for a message after sending a message. Generally, this option should be set to yes (Y). By doing so, the simulator automatically waits for a response to any request sent. This facilitates SN simulations. Likewise, the simulator automatically waits for another request after sending a response. This facilitates DMS switch simulations.

Interactive Menu

The Interactive Menu provides most of the interactive capabilities of the OAP Simulator. There is an Interactive Menu for each context:

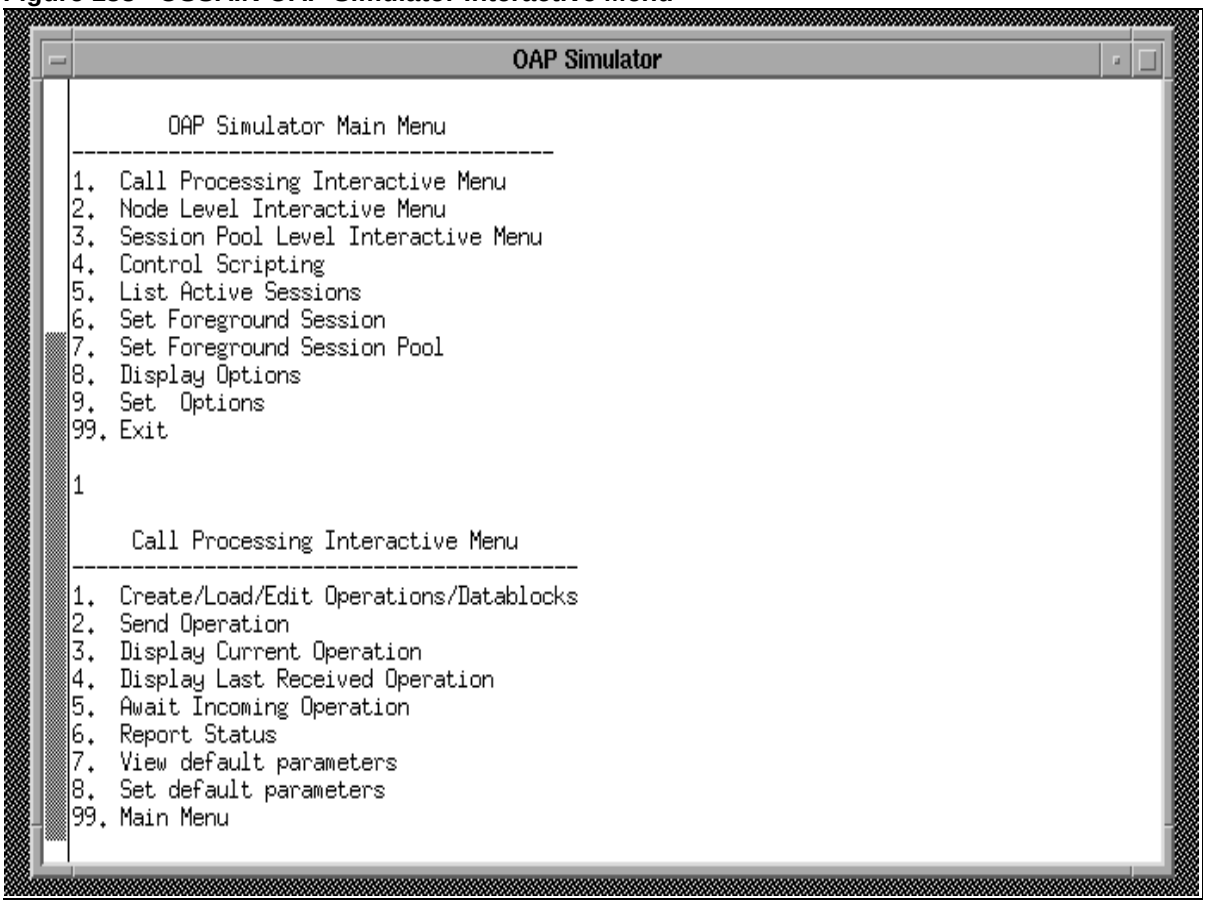
- call processing
- node maintenance
- session pool maintenance

The only difference in these menus is the context in which they operate.

The Interactive Menu allows the user to create, load, and send OAP operations. It also allows the user to display the current or last received operation, wait for an incoming operation, report the current status of the context, and view and modify interactive parameters.

Figure 138 shows an example of navigating from the Main Menu to the Call Processing Interactive Menu.

Figure 138 OSSAIN OAP Simulator Interactive Menu



Create/Load/Edit Operations/Datablocks

This option selects the operation and data block editors, where the user can create, load, and edit operations and data blocks. Refer to "Operation Editor Menu" on page 234 for details on editing operations. Refer to "Datablock Editor Menu" on page 238 for details on editing data blocks.

Send Operation

This option sends the current operation, if one exists. If configured to do so with option settings on the Main Menu, the simulator automatically waits for an incoming message after sending the current operation.

Display Current Operation

If a working operation exists, this option displays the high-level operation data and message bytestream of the current operation.

Display Last Received Operation

If a previous incoming message has been received, this option displays the high-level operation data and message bytestream of the last received message.

Await Incoming Operation

This option places the simulator in a wait state for an incoming message. If a message is received for the current context, the message is displayed followed by the interactive menu.

If the incoming message is for a *different* context, the message is displayed and the user is prompted to switch to the other context. If the user chooses to switch context, the interactive menu is displayed under the new call context. If the user chooses not to switch context, the simulator resumes waiting for an incoming message.

Note: If the message received is a node or session pool maintenance message, the context is automatically switched so that the maintenance message can be quickly processed.

The user can interrupt the wait state by entering a control break key sequence: Ctrl + c. This returns the user to the Interactive Menu.

Report Status

This option displays the current status of the current context. The information displayed depends on the current context, as follows:

- for the node context:
  - node identifier
- for the session pool context:
  - node identifier
  - session pool identifier
- for the session context:
  - node identifier
  - session pool identifier
  - session identifier
  - call status (active or idle)

View default parameters

This option displays the current settings for default parameters.

Set default parameters

This option allows the user to modify default parameters. Parameters include the current OAP version and the request timeout value. These parameters are initialized based on data provided by the configuration file and can be modified "on the fly" with this option.

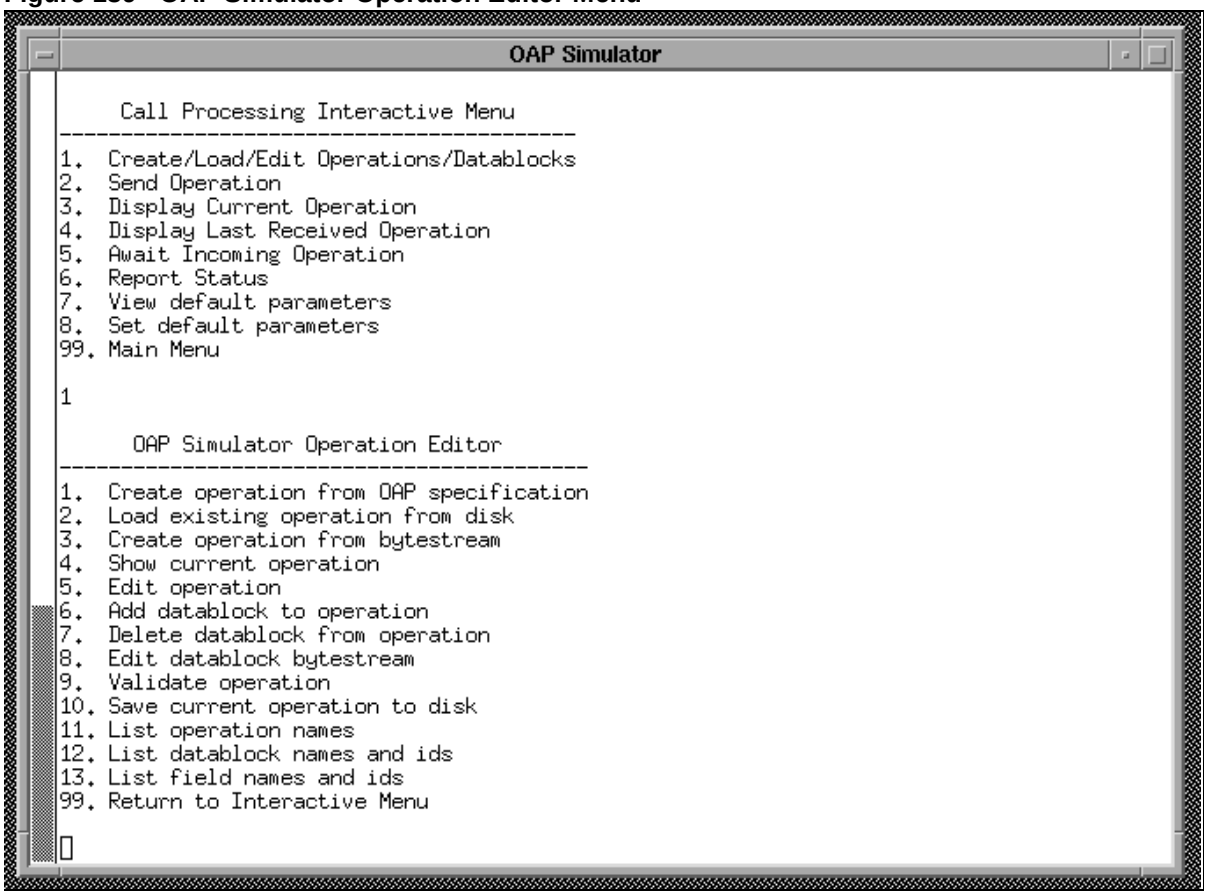
Main Menu

This option returns the user to the OAP Simulator Main Menu.

Operation Editor Menu

The Operation Editor Menu allows the user to create, load, validate, and send OAP operations. It also allows the user to add data blocks to operations or remove data blocks from operations. Figure 139 shows an example of navigating from an Interactive Menu to the Operation Editor Menu.

Figure 139 OAP Simulator Operation Editor Menu



Create operation from OAP specification

This option creates a default operation based on the OAP specification. When this option is selected, an indexed list of all supported OAP operation names is displayed ordered by operation type (such as requests, informs, return results). The user enters the index of the desired operation, and the simulator creates the default operation from the OAP specification.

After the operation is created, it is displayed and the user is prompted to send or edit the operation. If the response is yes, the operation is sent to its destination. If the response is edit, the user is prompted to edit each data block in the operation. This includes the class and operation header data blocks. If the response is no, the Operation Editor Menu is re-displayed.

Load existing operation from disk

This option loads an operation from disk. When this option is selected, an indexed list of all supported OAP operation names is displayed ordered by operation type. The user enters the index of the desired operation, and the simulator locates the operation scenario file for the operation and displays the list of selectable scenarios.

The user is prompted for the desired scenario name. The simulator builds the operation using the list of data blocks specified in the selected scenario.

After the operation is built, it is displayed and the user is prompted to send or edit the operation. If the response is yes, the operation is sent to its destination. If the response is edit, the user is prompted to edit each data block in the operation. This includes the class and operation header data blocks. If the response is no, the Operation Editor Menu is re-displayed.

Create operation from bytestream

This option builds an operation from a bytestream. The user is prompted for the bytestream to be used as the basis of the operation. The user is continually prompted for data until a blank entry is provided and the simulator builds an operation using the provided data.

After the operation is built, it is displayed and the user is prompted to send or edit the operation. If the response is yes, the operation is sent to its destination. If the response is edit, the user is prompted to edit each data block in the operation. This includes the class and operation header data blocks. If the response is no, the Operation Editor Menu is re-displayed.

Show current operation

This option displays the current working operation. The high-level operation data is displayed along with the individual bytestream for each data block, followed by the overall message bytestream.

Edit operation

This option edits the current working operation. When this option is selected, the current working operation is displayed and the user is prompted to edit *each* data block in the operation. This includes the class and operation headers.

The user has the option to respond with yes, no, or quit to each prompt. If the response is yes, the high-level data for the data block is displayed and the user is prompted to edit each field in the data block. If the user presses enter without entering any data for a field prompt, the current setting for that field remains the same. If a new entry is made, it is updated for the data block.

Note: Values are not validated at this time. If validation is required, the user should select the validate option after completing the editing of the operation.

If the response is no, the data block is skipped and the user is prompted to edit the next data block. If the response is quit, the user is prompted to send or edit the operation. If the response is yes, the current operation is sent to its destination. If the response is edit, the operation edit process begins again starting at the message class header. If the response is no, the user is returned to the Operation Editor Menu.

Add datablock to operation

This option adds data blocks to the current working operation. Selecting this option places the user in the Datablock Editor Menu, where data blocks can be created, loaded, and edited. Refer to “Datablock Editor Menu” on page 238 for details.

When exiting the Datablock Editor Menu, the user has the option of saving the current working data block into the current operation or not. If the user chooses to add the operation, it is appended to the end of the current working operation. The message class and operation headers of the operation are updated appropriately.

Delete datablock from operation

This option removes a data block from the current working operation. Selecting this option displays the name of each data block in the current working operation. The user selects a data block to delete, and the data block is removed from the operation. If the user presses enter without providing a data block name, or if the data block name is misspelled so that it is not found in the operation, the operation is not modified.

Edit datablock bytestream

This option edits bytestreams of data blocks found in the current working operation. Selecting this option displays the name of each data block in the current working operation. The user selects a data block, and the simulator displays the field information and the data block bytestream. The user is prompted to provide the new bytestream for the data block. The user is continually prompted for data until a blank entry is provided, and the simulator displays the updated data block. The user is prompted to save the data block into the operation.

Validate operation

This option validates the current working operation against the OAP specification. The level of validation performed depends on the configuration file settings for OAP validation.

Save current operation to disk

This option saves the current working operation to file. The user is prompted for a scenario name. The scenario name must be unique.

The simulator appends the operation to the operation scenario file associated with that operation. Each data block in the operation is added to the data block database. The data block name is a concatenation of the OAP data block name and the scenario name. If the data block name is not unique, the existing data block found in the database file is overwritten with the new bytestream.

List operation names

This option lists the supported OAP operation names grouped by operation type.

List datablock names and ids

This option lists data block IDs and their associated data block name sorted by data block name.

List field names and ids

This options lists all of the OAP field names.

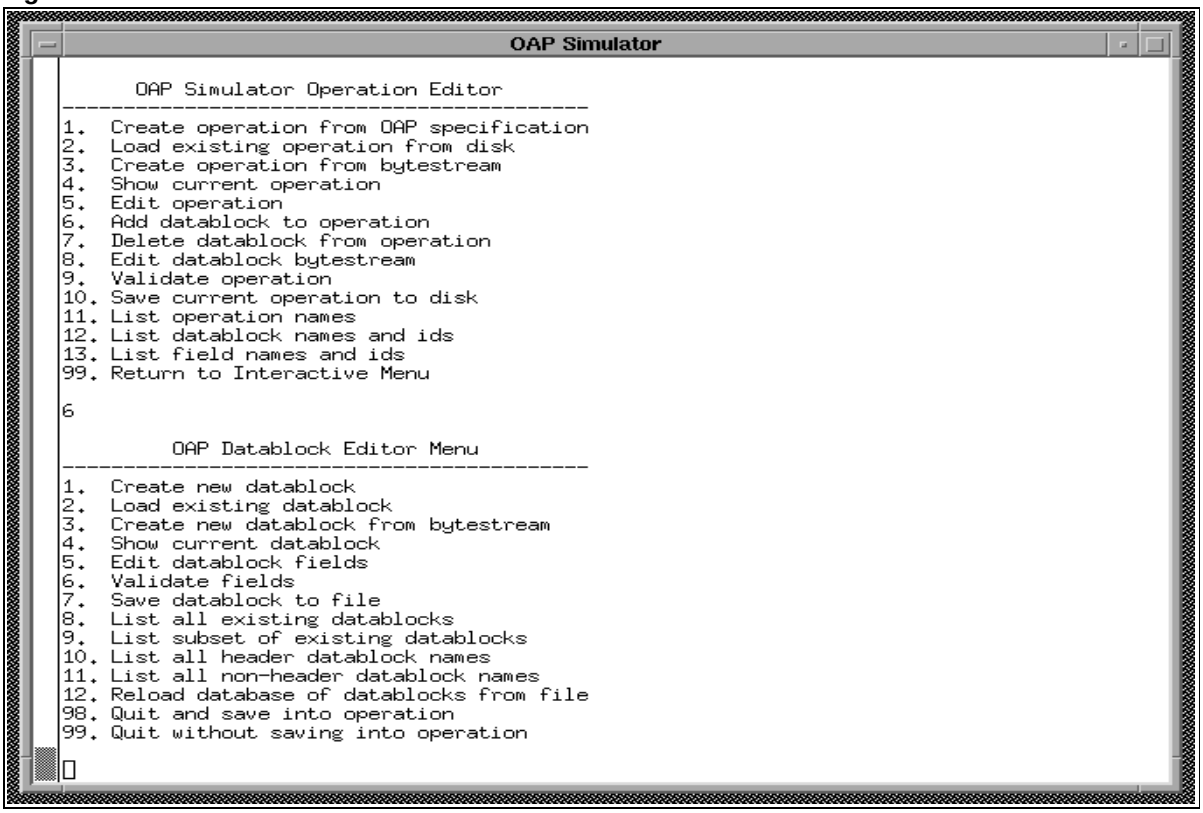
Return to Interactive Menu

This option returns the user to the OAP Simulator Main Menu.

Datablock Editor Menu

The Datablock Editor Menu allows users to create, load, and validate OAP data blocks. Figure 140 shows an example of navigating from the Operation Editor Menu to the Datablock Editor Menu.

Figure 140 OAP Simulator Datablock Editor Menu



Create new datablock

This option creates a default data block based on the OAP specification. The user is prompted for a the name of a data block. Users can list supported data block names by selecting menu options "List all header datablock names" and "List all non-header datablock names" (see page 240).

After the user enters the data block name, the simulator creates the default data block from the OAP specification. If there is an existing working data block in the data block editor, it is overwritten by the new data block. After the data block is created, it is displayed and the user is returned to the Datablock Editor Menu.

Load existing datablock

This option loads an existing data block from the data block database. The user is prompted for the name of an existing data block. Users can list existing data block names by selecting menu options "List all existing datablocks" and "List subset of existing datablocks" (see page 240).

After the user enters the name of the existing data block, the simulator builds a data block from the data provided by the database. If there is an existing working data block in the data block editor, it is overwritten by the new data block. After the data block is built, it is displayed and the user is returned to the Datablock Editor Menu.

Create new datablock from bytestream

This option creates a data block from a bytestream. The user is continually prompted for data until a blank entry is provided, and the simulator builds a data block from the provided data.

If there is an existing working data block in the data block editor, it is overwritten by the new data block. After the data block is created, it is displayed and the user is returned to the Datablock Editor Menu.

Show current datablock

This option displays the current working data block. The high-level data for each field is displayed along with the overall data block bytestream.

Edit datablock fields

This option edits the fields of the current working data block. Selecting this option displays the data block. The user is prompted to update values for each field in the data block. If a value is not provided, the field remains unchanged. After all fields have been processed, the user is returned to the Datablock Editor Menu.

Validate fields

This option validates the field data of the current working data block against the OAP specification.

Save datablock to file

This option saves the current working data block to file. The user is prompted for a data block name. If the specified name already exists, the user is prompted to overwrite the existing data block.

List all existing datablocks

This option lists all existing data blocks (found in the data block database) in alphabetical order.

List subset of existing datablocks

This options lists a subset of existing data blocks (found in the data block database). The user is prompted for a substring to search for. Any data block names meeting the regular expression built from the specified substring are listed. For example, all data block names starting with Voice can be found by specifying Voice as the search string.

List all header datablock names

This option lists all header data block names supported by the OSSAIN API (such as CallpClshdr, ClassHeader).

List all non-header datablock names

This option lists all non-header data block names supported by the OSSAIN API (such as AmaModule, ConfPartyConnectionBParty).

Reload database of datablocks from file

This option forces the simulator to reload the data block database into core memory. This helps when merging another user's data block database changes with your own. The two data block database files can be merged and then reloaded with this option.

Quit and save into operation

This option returns the user to the Operation Editor Menu and saves (adds) the current working data block into the current working operation.

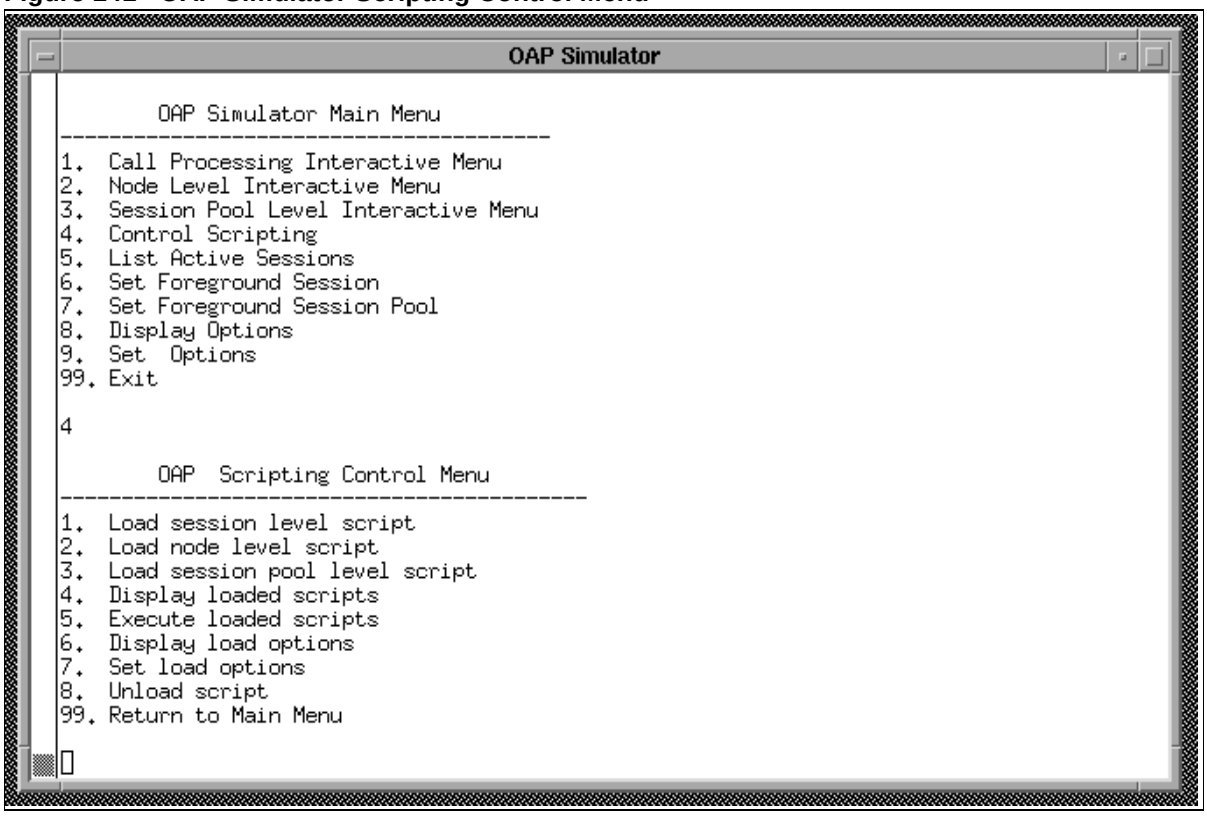
Quit without saving into operation

This option returns the user to the Operation Editor Menu with out saving the current working data block.

Scripting Control Menu

The scripting control menu allows users to load, unload, and execute simulator scripts. Figure 141 shows an example of navigating from the Main Menu to the Scripting Control Menu.

Figure 141 OAP Simulator Scripting Control Menu



Load session level script

This option loads a script at the session level. The user is prompted for the filename containing the script. The simulator attempts to load the script into the session level contexts currently configured for loading. Refer to "Set load options" on page 242 for a description of setting load contexts.

During the loading process, the script file is parsed to verify that it is formatted correctly. If errors are found, error logs are produced and the script file will fail to load.

If the script passes the parsing phase of loading, each operation specified in the script file is built by the simulator. The simulator accesses the appropriate operation scenario files and the data block database file to obtain the data needed to create each OAP message used by the script. If errors are found, (such as not locating an operation scenario or data block), the script will fail to load. Refer to "Scripting" on page 243 for details about the script language and script file format.

Load node level script

This option loads a script at the node level. The user is prompted for the filename containing the script to be loaded. The simulator attempts to load the script into the node context currently configured for loading. Refer to “Set load options” on page 242 for a description of setting load contexts.

Load session pool level script

This option loads a script at the session pool level. The user is prompted for the filename containing the script to be loaded. The simulator attempts to load the script into the session pool context currently configured for loading. Refer to “Set load options” on page 242 for a description of setting load contexts.

Display loaded scripts

This option displays all loaded scripts. The display consists of the script name (if one was provided by the script file), a script index (assigned by the simulator during the load process), and the script statements that make up the script.

Execute loaded scripts

This option begins execution of all scripts that are currently loaded. The order of execution is determined by the assigned script index. The script with index 1 will be executed before the script with index 2 and so on.

The simulator examines each script to determine the contexts that have been loaded with the script. Then the simulator starts the script flow for the associated contexts. The simulator performs a staggered start of scripts. Five contexts are started and then the simulator attempts to process any received messages. Then the next five contexts are started.

Execution continues until all scripts have completed. The execution process can be interrupted by entering a control break key sequence: Ctrl + c. This halts all scripts and the simulator returns the user to the Scripting Control Menu.

Display load options

This option displays the scripting load options. These parameters determine which contexts the scripts are loaded into, and whether or not OAP messages are displayed during the execution of scripts.

Set load options

This option allows users to modify script load options. These parameters determine which contexts a script is loaded into. The node, session pool, and session range can be specified with this option. By varying these parameters, the user can dictate which contexts are loaded when the load options (session level, node level, and session pool level) are selected.

By specifying a range for sessions, a session pool can be set up to execute multiple call flows. For example, sessions 0 through 5 on session pool 1 can be loaded with one script, while sessions 6 through 10 on the same session pool can be loaded with another script. As calls are sent to this session pool, those arriving on sessions 0 through 5 will be processed with one call flow, while calls arriving on sessions 6 through 10 will be processed with another.

Unload script

This option unloads a previously loaded script. The user is prompted for the script index of the script to be unloaded. Users can display the script index by selecting menu option "Display loaded scripts" (see page 242). After entering the script index, the simulator unloads the specified script.

Return to Main Menu

This option returns the user to the OAP Simulator Main Menu.

Scripting

The OAP Simulator supports a programming-like language that allows users to build entire call processing and maintenance message flows called *scripts*. Scripts can specify the number of times flows are to be executed, the success and error paths to be taken, and statistics to be reported. Scripts are loaded and executed by the OAP Simulator to simulate single or multiple calls and node/session pool maintenance scenarios.

This section discusses scripting, focusing on the following areas:

- script files
- script language
- special field handling

Script files

A message flow (that is, script) is written in the OAP Simulator script language and placed into a text file. Script files are created using any available text editor.

When the OAP Simulator loads a script file, it parses the file to verify that the syntax of the file meets the scripting language specification. If an error is found, error messages are generated that indicate the specific problem. Refer to "Scripting Control Menu" on page 241 for information on loading script files.

Script language

Script files are written in the OAP Simulator script language, which is a block-oriented language. Most language statements are in a Begin . . . End block format.

The script language is flexible and allows the user much control of call and maintenance message flows. Key to this flexibility is the ability to analyze incoming OAP messages to determine success and error paths within the script.

This subsection provides a reference for the simulator script language. The syntax of each statement is shown, followed by a detailed description and example of its use.

COMMENT statement

comment_statement ::= # *comment_text* $_1$

The comment statement begins with the pound sign followed by a text string that may include white spaces. The pound sign may be preceded by white space. The newline character signifies the end of the comment statement. Comment statements may appear anywhere in the script file.

Figure 142 Example COMMENT statement

```
# My comment line
# My second comment line
```

SCRIPT block statement

script_block_statement ::= BEGINSCRIPT *ScriptName*

statement $_{0+}$

ENDSCRIPT

The script block statement defines the beginning and end of a script. An optional string argument, ScriptName, may be provided as an argument to the beginscript key word. If provided, this string is displayed when the script is displayed by the OAP Simulator and when script statistics are displayed.

Zero or more statements may be provided within the beginscript and endscript key words. Only one script block statement may appear in a script file.

Figure 143 Example SCRIPT block statement

```
# Example use of script block statement
BEGINSCRIPT “This is a sample script”
      statement_1
      statement_2
ENDSCRIPT
```

FLOW block statement

flow_block_statement ::= BEGINFLOW *FlowCount*

statement $_{0+}$

ENDFLOW

The flow block statement identifies the beginning and ending of executable statements of the script. The flow block must appear nested within a script block statement. Zero or more statements may be provided within the beginflow and endflow key words. The flow control block is required for a script to be loaded successfully by the simulator. Only one flow block statement may appear within a script block.

An optional integer value, FlowCount, may be provided as an argument to the beginflow key word. This argument specifies the number of times the flow is to be executed. If this argument is not provided, the flow will be executed once.

Figure 144 Example FLOW block statement

```
# Example use of flow block statement
BEGINSCRIPT “This is a sample script”
      BEGINFLOW 5
            # This flow will be executed 5 times
            statement_1
            statement_2
      ENDFLOW
ENDSCRIPT
```

DCLDNMGR assignment statement

dcldnmgr_statement ::= DCLDNMGR *MgrName*
DirNumStart
DirNumCount

The DCLDNMGR statement defines a directory number resource manager to be used during the scripted simulation. This statement must appear nested within the script block statement. There is no limit to the number of directory number resource managers that may be defined.

The mandatory string argument, MgrName, specifies the name of the directory number manager being defined. This must be a unique alphanumeric string.

A directory number resource manager manages a specified range of directory numbers. The mandatory string argument, DirNumStart, specifies the starting directory number. Only ten-digit directory numbers are supported. The mandatory integer argument, DirNumCount, specifies the number of directory numbers to be managed.

Directory numbers are requested from the directory number manager with the GETDN statement and returned with the RELDN statement.

Figure 145 Example DCLDNMGR assignment statement

```
# Example definition of a directory number manager
BEGINSCRIPT
      # Starting DN = 2012200010
      # Number of DNs = 10
     DCLDNMGR dirMgr1 2012200010 10
ENDSCRIPT
```

DCLSTAT assignment statement

dclstat_statement ::= DCLSTAT *RegisterName*
RegisterDescription

The DCLSTAT statement defines a statistical register to be used for the scripted simulation. This statement must appear nested within the script block statement. There is no limit to the number of statistical counters that may be defined.

The mandatory string argument, RegisterName, specifies the name of the statistics register. This must be a unique alphanumeric string. The mandatory string argument, RegisterDescription, is a character string describing the statistics register being defined.

Figure 146 Example DCLSTAT assignment statement

```
# Example definition of statistics registers
BEGINSCRIPT
     DCLSTAT callAttempt “Calls Attempted”
     DCLSTAT callSuccess “Call Success”
ENDSCRIPT
```

DCLVOICEMGR assignment statement

dclvoicemgr_statement ::= DCLVOICEMGR *MgrName*
ChanStart
ChanCount
CallsPerChannel

The DCLVOICEMGR statement defines a voice channel resource manager to be used during the scripted simulation. This statement must appear nested within the script block statement. There is no limit to the number of voice channel resource managers that may be defined.

The mandatory string argument, MgrName, specifies the name of the voice channel manager being defined. This must be a unique alphanumeric string.

A voice channel resource manager manages a specified range of logical voice channels. The mandatory string argument, ChanStart, specifies the starting voice channel. The mandatory integer argument, ChanCount, specifies the number of voice channels to be managed. The mandatory integer argument, CallsPerChannel, specifies the number of calls that can use the same channel. For non-broadcast voice links, this argument should be set to 1.

Voice channels are requested from the voice channel manager with the GETCHAN statement and returned with the RELCHAN statement.

Figure 147 Example DCLVOICEMGR assignment statement

```
# Example definition of a voice channel manager
BEGINSCRIPT
      # Starting channel = 0
      # Number of channels = 10
      # Calls per channel = 1
     DCLVOICEMGR voiceMgr1 0 10 1
ENDSCRIPT
```

CANCELWAIT statement

cancelwait_statement ::= CANCELWAIT

The CANCELWAIT statement causes all timers for the given context to be canceled. This includes timers that are set when a request message is sent and timers that are set by the WAIT statement. If no timers are currently set when the CANCELWAIT statement is executed, there is no effect.

The CANCELWAIT statement must appear nested within a flow block statement. There is no limit to the number of CANCELWAIT statements that may appear within a flow block.

Figure 148 Example CANCELWAIT statement

```
# Example use of cancel wait statement
BEGINSCRIPT
   BEGINFLOW
      # Wait for 30 seconds or for an incoming message
      WAIT 30000

      # We are here because the simulator has returned control
      # to the script after being in the wait state. We either
      # received a message or the wait timeout expired. Cancel
      # the wait timeout in case this is an incoming message.
      CANCELWAIT

      # Process the incoming message
   ENDFLOW
ENDSCRIPT
```

CLEAR statement

clear_statement ::= CLEAR *RegisterName*

The CLEAR statement sets the user statistic specified in the mandatory string argument, RegisterName, to zero. The CLEAR statement must appear nested within a flow block statement. There is no limit to the number of CLEAR statements that may appear within a flow block.

Figure 149 Example CLEAR statement

```
# Example use of clear statement
BEGINSCRIPT
    DCLSTAT callAttempts “Call Attempts”
    BEGINFLOW
        # Set the call attempt register to 0
        CLEAR callAttempts
    ENDFLOW
ENDSCRIPT
```

DISPLAYSTATS statement

displaystats_statement ::= DISPLAYSTATS *DisplayFrequency*

The DISPLAYSTATS statement causes the scripting simulation to display the user defined statistics. The mandatory integer argument, DisplayFrequency, specifies how often the user defined statistics are to be displayed. The simulator tracks the number of times the DISPLAYSTATS statement is processed. When the number of passes equals the value of the DisplayFrequency argument, the user defined statistics are displayed.

The statistics display statement must appear nested within a flow block statement. There is no limit to the number of DISPLAYSTATS statements that may appear within a flow block.

Figure 150 Example DISPLAYSTATS statement

```
# Example use of display statistics statement
BEGINSCRIPT
DCLSTAT callAttempts “Call Attempts”
    BEGINFLOW 100
        # Increment the register 100 times
        INCR callAttempts
        # Display stats every 10th time
        DISPLAYSTATS 10
    ENDFLOW
ENDSCRIPT
```

FLOWEXIT statement

flow_exit_statement ::= FLOWEXIT

The FLOWEXIT statement causes the script to end execution. Other contexts executing the same script as the one being terminated are allowed to complete execution before the script terminates.

The FLOWEXIT statement must appear nested within a flow block statement. There is no limit to the number of FLOWEXIT statements that may appear within a flow block.

Figure 151 Example FLOWEXIT statement

```
# Example use of flow exit statement
BEGINSCRIPT
    DCLSTAT callAttempts "Call Attempts"
    BEGINFLOW
        FLOWEXIT
        # This statement will never execute
        CLEAR callAttempts
    ENDFLOW
ENDSCRIPT
```

FLOWSTART statement

flow_exit_statement ::= FLOWSTART

The FLOWSTART statement causes the script to return to the beginning of the flow block.

The FLOWSTART statement must appear nested within a flow block statement. There is no limit to the number of FLOWSTART statements that may appear within a flow block.

Figure 152 Example FLOWSTART statement

```
# Example use of flow start statement
BEGINSCRIPT
    DCLSTAT callAttempts "Call Attempts"
    BEGINFLOW
        FLOWSTART
        # This statement will never execute
        CLEAR callAttempts
    ENDFLOW
ENDSCRIPT
```

GETDN statement

getdn_statement ::= GETDN *Party MgrName*

The GETDN statement obtains a directory number from the specified directory number manager, MgrName, and saves it for the specified party, Party. The directory manager specified must have been previously defined with the DCLDNMGR statement. The party specified may be APARTY, BPARTY, or ALTPARTY.

The directory number will be set for the appropriate data blocks by the simulator in subsequent message processing. Refer to "Special field handling" on page 261 for details on automated directory number updates.

If there are no directory numbers available when the GETDN statement is executed, a minor alarm is generated. Directory numbers are returned to the directory number manager with the RELDN statement.

The GETDN statement must appear nested within a flow block statement. There is no limit to the number of GETDN statements that may appear within a flow block.

Figure 153 Example GETDN statement

```
# Example use of getdn statement
BEGINSCRIPT
    # Define a dn mgr starting at 2012200010 with a
    # total of 10 directory numbers
    DCLDNMGR dnmgr1 2012200015 10
    BEGINFLOW
        # Obtain a directory number for the BParty
        GETDN BPARTY dnmgr1
    ENDFLOW
ENDSCRIPT
```

GETCHAN statement

getchan_statement ::= GETCHAN *MgrName*

The GETCHAN statement obtains a logical voice channel identifier from the specified voice channel manager, MgrName. The voice channel manager specified must have been previously defined with the DCLVOICEMGR statement.

The logical voice channel will be set for the appropriate data blocks by the simulator in subsequent message processing. Refer to "Special field handling" on page 261 for details on automated logical voice channel identifier updates.

If there are no voice channels available when the GETCHAN statement is executed, a minor alarm is generated. Voice channels are returned to the voice channel manager with the RELCHAN statement.

The GETCHAN statement must appear nested within a flow block statement. There is no limit to the number of GETCHAN statements that may appear within a flow block.

Figure 154 Example GETCHAN statement

```
# Example use of getchan statement
BEGINSCRIPT
    # define voice chan mgr starting at channel 1,
    # with 30 channels, and 1 call per channel
    DCLVOICEMGR voicemgr1 1 30 1
    BEGINFLOW
        # Obtain a logical voice channel
        GETCHAN voicemgr1
    ENDFLOW
ENDSCRIPT
```

INCR statement

incr_statement ::= INCR *RegisterName*

The INCR statement increments the statistics register specified in the mandatory string argument, RegisterName. The INCR statement must appear nested within a flow block statement. There is no limit to the number of INCR statements that may appear within a flow block.

Figure 155 Example INCR statement

```
# Example use of incr statement
BEGINSCRIPT
    DCLSTAT callAttempts "Call Attempts"
    BEGINFLOW
        # Increment the call attempts register
        INCR callAttempts
    ENDFLOW
ENDSCRIPT
```

LOOP block statement

loop_block_statement ::= LOOP *LoopCount*

statement $_{0+}$

ENDLOOP

The loop block statement provides repetition of statements nested within the loop block. Loop blocks must appear nested within a flow block statement. There is no limit to the number of loop blocks that may appear within a flow block. Loop blocks may be nested within a loop block.

The statements nested within the loop block are repeated the number of times specified by the optional integer argument, LoopCount. An infinite loop can be created by excluding the LoopCount argument.

Figure 156 Example LOOP block statement

```
# Example use of loop block statement
BEGINSCRIPT
      BEGINFLOW
           # Repeat 1000 times
           LOOP 1000
                statement_N
           ENDLOOP

           # Repeat forever
           LOOP
                statement_M
           ENDLOOP
      ENDFLOW
ENDSCRIPT
```

LOOPEXIT statement

loop_exit_statement := LOOPEXIT

The LOOPEXIT statement causes the script flow to jump out of the controlling loop block statement. The statement following the loop block statement will be the next to be executed.

The LOOPEXIT statement must appear nested within a loop block statement. There is no limit to the number of LOOPEXIT statements that may appear within a loop block.

Figure 157 Example LOOPEXIT statement

```
# Example use of loop exit statement
BEGINSCRIPT
      DCLSTAT callAttempts “Call Attempts”
      BEGINFLOW
        LOOP 100
             # jump out of the controling loop block
             LOOPEXIT
             # This statement will never execute
             INCR callAttempts
        ENDLOOP
      ENDFLOW
ENDSCRIPT
```

LOOPSTART statement

loop_start_statement ::= LOOPSTART

The LOOPSTART statement causes the script flow to jump to the top of the controlling loop block statement.

The LOOPSTART statement must appear nested within a loop block statement. There is no limit to the number of LOOPSTART statements that may appear within a loop block.

Figure 158 Example LOOPSTART statement

```
# Example use of the loop start statement
BEGINSCRIPT
      DCLSTAT callAttempts "Call Attempts"
      BEGINFLOW
         LOOP 100
              # jump to the top of the controlling loop block
              LOOPSTART
              # This statement will never execute
              INCR callAttempts
         ENDLOOP
      ENDFLOW
ENDSCRIPT
```

PRINT statement

print_statement ::= PRINT *PrintString*

The PRINT statement causes the scripting simulation to display the character string specified in the mandatory string argument, PrintString. The PrintString argument is limited to 80 characters.

The PRINT statement must appear nested within a flow block statement. There is no limit to the number of PRINT statements that may appear within a flow block.

Figure 159 Example PRINT statement

```
# Example use of print statement
BEGINSCRIPT
      BEGINFLOW
         LOOP 100
              # Print the standard greeting 100 times
              PRINT "Hello world"
         ENDLOOP
      ENDFLOW
ENDSCRIPT
```

RELDN statement

reldn_statement := RELDN *Party MgrName*

The RELDN statement releases the directory number of the party specified by Party to the directory number manager specified by MgrName. The directory number manager specified must have been previously defined with the DCLDNMGR statement. The party specified may be APARTY, BPARTY, or ALTPARTY.

If the specified party does not have a directory number set, or the directory number was set by an incoming OAP message, the RELDN statement is ignored.

The RELDN statement must appear nested within a flow block statement. There is no limit to the number of RELDN statements that may appear within a flow block.

Figure 160 Example RELDN statement

```
# Example use of reldn statement
BEGINSCRIPT
    # Define a dn mgr starting at 2012200010 with a
    # total 10 directory numbers
    DCLDNMGR dnmgr1 2012200015 10
    BEGINFLOW
        # Obtain a directory number for the BParty
        GETDN BPARTY dnmgr1

        # Now release the BPParty directory number
        RELDN BPARTY dnmgr1
    ENDFLOW
ENDSCRIPT
```

RELCHAN statement

relchan_statement := RELCHAN *MgrName*

The RELCHAN statement releases the current logical voice channel identifier to the specified voice channel manager, MgrName. The voice channel manager specified must have been previously defined with the DCLVOICEMGR statement.

If there is not a logical voice channel identifier currently set, the RELCHAN statement is ignored.

The RELCHAN statement must appear nested within a flow block statement. There is no limit to the number of RELCHAN statements that may appear within a flow block.

Figure 161 Example RELCHAN statement

```
# Example use of relchan statement
BEGINSCRIPT
    # define voice chan mgr starting at channel 1,
    # with 30 channels, and 1 call per channel
    DCLVOICEMGR voicemgr1 1 30 1
    BEGINFLOW
        # Obtain a logical voice channel
        GETCHAN voicemgr1

        # Now release the voice channel
        RELCHAN voicemgr1
    ENDFLOW
ENDSCRIPT
```

SEND statement

send_statement ::= SEND *OperationName ScenarioName*

The SEND statement causes the scripting simulation to send the OAP operation specified by OperationName and ScenarioName arguments. The SEND statement must appear nested within a flow block statement. There is no limit to the number of send statements that may appear within a flow block.

The OperationName argument specifies the operation to be sent (such as SessionInitiationRequest, SessionBeginInform, or SessionInitiationRetres). The ScenarioName argument specifies which scenario from the operation scenario file to use to build the OAP message. At the time the script is loaded by the simulator, the OperationName and ScenarioName arguments are verified and the OAP message is constructed. If verification or message construction fails, the simulator will fail to load the script file.

When the SEND statement is executed, the message class and operation headers are constructed and fields such as the message sequence number, directory numbers, and voice channel identifiers are updated. Refer to "Special field handling" on page 261 for details on automated field updating.

If the message being sent is a request message, a request timeout is automatically set. If the request timer expires before a response is received, a timeout message is received. The timeout message is a return error message with an error id value of 0xFD00. The amount of time to wait for a response message is configurable with the RequestTimeout field in the simulator configuration file. The RequestTimeout field may be specified at the node, session pool, or call processing level.

Figure 162 Example SEND statement

```
# Example use of send statement
BEGINSCRIPT
    BEGINFLOW
        # Send a session begin inform
        SEND SessionBeginInform scenario_1
    ENDFLOW
ENDSCRIPT
```

SWITCH statement

switch_block_statement ::= SWITCH (OPTYPE |
OPNAME |
DBNAME |
FIELDVALUE |
ERRORID)

optype_case_statement $_{0+}$ |
opname_case_statement $_{0+}$ |
dbname_case_statement $_{0+}$ |
fieldvalue_case_statement $_{0+}$ |
errorid_case_statement $_{0+}$ |
default_statement $_{opt}$

ENDSWITCH

optype_case_statement ::= CASE (INVOKE |
RETRES |
RETERR |
REJECT)

statement $_{0+}$

ENDCASE

opname_case_statement ::= CASE *OperationName*

statement $_{0+}$

ENDCASE

dbname_case_statement ::= CASE *DatablockName*

statement $_{0+}$

ENDCASE

fieldvalue_case_statement ::= CASE *FieldName DatablockName FieldValue*

statement $_{0+}$

ENDCASE

errorid_case_statement ::= CASE *ErrorId*

statement $_{0+}$

ENDCASE

default_statement ::= DEFAULT

statement $_{0+}$

ENDDEFAULT

The SWITCH statement is a multiway conditional statement performed on the last OAP message received by the context. The SWITCH statement must appear nested within a flow block statement. There is no limit to the number of switch block statements that may appear within a flow block.

There is no limit on the number of CASE block statements that may be nested within the switch block statement. Any statement, except the flow block and script block statements, may be nested within the CASE block statement. Only one DEFAULT block statement can be associated with a switch block statement.

The controlling key word of the SWITCH statement determines the type of evaluation performed on the incoming OAP message. The arguments to the enclosed CASE statements are evaluated against the incoming OAP message to determine if a match exists. If a match is found, the statements enclosed by the case statement are executed. Upon completion of the last statement enclosed by the case statement, the flow of control jumps to the statement following the SWITCH block as the next statement to be executed. If no matches are found among the included CASE statements, the DEFAULT block is executed (if included).

If there are multiple CASE statements that evaluate to a match with the incoming message (for example, two field value comparisons on different fields), the first CASE statement encountered will be executed.

The controlling key word of the SWITCH statement determines the supported arguments of the enclosed CASE statements. The SWITCH statement supports the following controlling key words:

- OPTYPE—Operation Type

  The operation type of the incoming OAP message is examined to determine if it matches any of the operation types indicated by the enclosed CASE statements. Valid operation types are INVOKE, RETRES, RETERR, and REJECT.

- OPNAME—Operation Name

  The operation name of the incoming OAP message is examined to determine if it matches any of the operation names indicated by the enclosed CASE statements. Valid operation names can be listed from the OAP Simulator operation editor.

- DBNAME—Datablock Name

  The incoming OAP message is examined to determine if any of the data block names indicated by the enclosed CASE statements are included in the OAP message. Valid data block names can be listed from the OAP Simulator operation editor.

- FIELDVALUE—Field Value

  The incoming OAP message is examined to determine if any of the field and data block names indicated by the enclosed CASE statements are included in the OAP message. If so, the value of the field from the message is compared to the value included in the CASE statement. If a match is found, the enclosed statements of the matching CASE statement are executed. Note that the field value in the CASE statement argument list is a string field. The data value to be compared should be represented as a character string.

- ERRORID—Error Identifier

  The error identifier of the incoming OAP message is examined to determine if it matches any of the error id values indicated by the enclosed CASE statements. If the operation type of the OAP message is not RETERR or if a matching error id value is not found, the comparison will fail.

Figure 163 Example SWITCH statement

```
# Example use of switch block statement
BEGINSCRIPT
   # Perform this call flow 1000 times
   BEGINFLOW 1000

   # Display stats every 100 calls
   DISPLAYSTATS 100

   # Wait 10 mins for a session begin. This will cause this
   # call to enter a wait state. Always cancel the wait upon
   # return from the wait state.
   WAIT 600000
   CANCELWAIT

   # We have some sort of message or a time out,
   # check the OPTYPE
   SWITCH OPTYPE
      CASE invoke
         # This is an invoke, is it a session begin?
         SWITCH OPNAME
            CASE SessionBeginInform
               # This is a session begin inform
               #
               PRINT "Received session begin"
            ENDCASE
            DEFAULT
               # Not a session begin. Is it a timeout
               SWITCH FIELDVALUE
                  CASE OperationIdValue OperationHeader fc00
                     # This is a timeout from the wait
                     PRINT "Timeout while waiting"
                  ENDCASE
                  DEFAULT
                     PRINT "Unexpected invoke"
                  ENDDEFAULT
               ENDSWITCH
               FLOWSTART
            ENDDEFAULT
         ENDSWITCH
      ENDCASE
      DEFAULT
         # Not an invoke...error
         PRINT "Unexpected optype received"
         FLOWSTART
      ENDDEFAULT
   ENDSWITCH

   ENDFLOW
ENDSCRIPT
```

WAIT statement

wait_statement ::= WAIT *timeout*

The WAIT statement causes the current context to enter a wait state. The WAIT statement must appear nested within a flow block statement. There is no limit to the number of WAIT statements that may appear within a flow block.

The WAIT statement optional argument, Timeout, specifies a time-out value (in milliseconds) to wait. The simulator will wait for an incoming OAP message to the waiting context or for the timeout to expire before executing any mores script statements for the context.

If the wait times out before an incoming message is received, an INVOKE operation is received with an operation identifier value of fc00. If the timeout argument is not supplied, the WAIT statement will wait forever for an incoming message.

Figure 164 Example WAIT statement

```
# Example use of wait statement
BEGINSCRIPT
    BEGINFLOW
      # Wait for 30 seconds or for an incoming message
      WAIT 30000
      # Now wait forever for an incoming message
      WAIT
    ENDFLOW
ENDSCRIPT
```

Special field handling

Several fields of incoming and outgoing OAP messages receive special handling by the OAP Simulator. The following sections describe this special handling.

Sequence number

The sequence number is a class header field common to all OAP message classes. This field indicates the sequence ordering of a message within a message flow.

- Incoming messages

  The expected sequence number is updated with each OAP message received. The expected sequence number is compared with the sequence number found in the incoming message. If the sequence numbers are out of sync, the incoming message is ignored.

  The expected sequence number is reset when the following OAP messages are received.

  — SessionBeginInform

  — SessionInitiationRequest

— SessionRecallRequest

— TriggerEventInform

- Outgoing messages

  Sequence numbers in outgoing messages are automatically updated by the simulator. The sequence number is reset when the following OAP messages are sent:

  — SessionBeginInform

  — SessionInitiationRequest

  — SessionRecallRequest

  — TriggerEventInform

Call identifier

The call identifier is a callp class header field that uniquely identifies a call. Call identifiers are managed by the DMS switch presenting the call to the SN. The simulator saves call identifiers provided by the switch when a call arrives at the simulator. This is done so that request messages originating from the simulator can be updated with the correct call identifier.

The simulator is capable of allocating call identifiers in order to fully simulate a DMS switch. This is done when the OAP messages being originated from the simulator indicate an attempt to originate a call from the DMS switch perspective (that is, originating a SessionBeginInform, TriggerEventInform, and so on).

- Incoming messages

  The call identifier is saved when an OAP message is received that signals a new call arrival. The simulator uses the saved call identifier in subsequent messages during the call. The call identifier is saved when the following messages are received:

  — SessionBeginInform

  — TriggerEventInform

  — SessionInitiationReturnResult

  — SessionRecallReturnResult

- Outgoing messages

  In most cases, the current call identifier that has been saved by the simulator is used for all outgoing messages. When the outgoing message indicates that the switch is simulating a DMS switch originating a call, a new call identifier is created for use during the simulated call. This is done when sending the following OAP operations.

  — SessionBeginInform

  — TriggerEventInform

— SessionInitiationReturnResult

— SessionRecallReturnResult

In some cases, it is necessary to include a NULL call identifier in the message. These are cases where an SN is requesting the origination of a call. Affected messages are:

— SessionInitiationRequest

— SessionRecallRequest

Function identifier

The function identifier is a callp class header field which specifies the function being referenced in the OAP message.

- Incoming messages

  The function identifier is saved on all incoming callp messages. This function identifier is then used in all subsequent outgoing messages.

- Outgoing messages

  In cases where the node is originating a call, the function identifier is set to a default function identifier that is configurable through the simulator configuration file. The function identifier is configured using the FUNCTION_IDENTIFIER key word in the simulator configuration file. The following messages use the default function identifier:

  — SessionBeginInform

  — TriggerEventInform

  — SessionRecallRequest

  — SessionInitiationRequest

Solicitor number

The solicitor number field is a maintenance class header field which identifies the maintenance task that is running at the switch or SN.

- Incoming messages

  The simulator saves the solicitor number from incoming maintenance messages and uses them to update outgoing response messages.

- Outgoing messages

  If the outgoing message is a response to a maintenance request, the previously saved solicitor number is used in the outgoing response. If the message is an unsolicited request, a default solicitor number (300) is used.

Invoke identifier

The invoke identifier (invoke id) is an OAP operation header field used to match an OAP response message to the original request message.

- Incoming messages

  The invoke identifier of every incoming invoke message is saved for use in updating the outgoing response message.

- Outgoing messages

  If the outgoing message is a response message, the invoke identifier previously saved is used to update the outgoing message. If the outgoing message is a request message, a new invoke id value is created and used to update the outgoing message.

Directory number

The directory number is a data block specific field which specifies the directory number of the specified party. The simulator is capable of saving directory numbers on a per party basis and updating outgoing messages with a directory number when appropriate.

Note: Automatic directory number updating is only supported in the simulator scripting mode.

- Incoming messages

  The OAP simulator analyzes incoming OAP operations to determine if they contain a directory number field. If so, the appropriate party is updated with the directory number from the incoming message. Table 27 on page 265 provides a matrix of incoming operations and data blocks that cause directory numbers to be saved by the simulator for the specified party.

  To interpret the table for incoming operations, begin by finding the operation of interest under the operation column. Once located, scan to the right to locate the data blocks that have been included with the incoming operation. Once an included data block is located, reference the column headers of the table to determine which party will be updated with the directory number being provided by the data block.

Table 27 Operation to party directory number matrix

Operation	AParty	BParty	AltParty
Accept Control Inform	Directory Number db	Directory Number db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Billing Number Request	n/a	Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Call Details Return Result	Directory Number db	Directory Number db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Call Merge Inform	Directory Number db	n/a	n/a
Directory Number Request	Directory Number Update db	Directory Number Update db	n/a
ISPU CGPN Update Request	ISUP Calling Party Number db	n/a	n/a
Session Begin Inform	Directory Number db	Directory Number db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Session Initiation Request	Directory Number Update db	Directory Number Update db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Session Recall Return Result	Directory Number db	Directory Number db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)
Trigger Event Inform	Directory Number db	Directory Number db Alternate Billing Directory Number db (collect)	Alternate Billing Directory Number db (alternate)

- Outgoing messages

  Outgoing messages are examined to determine if they contain a directory number field in any of the included data blocks. If so, the party is determined and the context is referenced to determine if a directory number has been set for the party referenced by the outgoing message. If the referenced party has a directory number set, it is used to update the directory number in the outgoing message. If a directory number is not set, the directory number from the data block database file, oap.dbs, is used.

  There are two ways that a party's directory number can be set. The first is via an incoming message as mentioned above. The other is through use of a script directory number manager. When a call is made to obtain a directory number from a directory number manager, GETDN, the party is specified. The obtained directory number is then saved for the specified party.

  In addition to illustrating how directory numbers are saved when incoming OAP messages are received, Table 27 on page 265 also can be interpreted to show which data blocks are updated with directory number information for outgoing messages. To determine this, begin by locating the party column of interest, for example AParty. Scan down the table until the operation of interest is located. The intersection of the operation row and the party column contains the data blocks that will be updated with the party directory number if those data blocks are included in the outgoing operations.

  Note: If the scenario being used to build the outgoing operation does not include a data block that would be updated with directory number information, the simulator does not automatically add the data block. All data blocks that are to be included in an outgoing operation must be listed in the operation scenario file.

Voice channel identifier

The voice channel identifier field specifies the logical voice channel number for the voice connection between the switch and the service node. The simulator is able to automatically allocate logical voice channel identifiers for use in the voice connect request operation.

Note: Automatic logical voice channel identifier updating is only supported in the simulator scripting mode.

- Incoming messages

  When an incoming voice connect request is received. The simulator examines the incoming message to determine the logical voice channel identifier. If found, the logical channel is saved for later use.

- Outgoing messages

  Outgoing messages are examined to determine if they contain a voice channel identifier field in any of the included data blocks. If so, the context is referenced to determine if a logical channel identifier has been set. If a channel identifier has been set, it is used to update the voice channel identifier in the outgoing message. If a channel identifier is not set, the channel identifier from the data block database file, oap.dbs, is used.

  There are two ways that a voice channel identifier can be set. The first is via an incoming message as mentioned above. The other is through use of a script voice channel manager. When a call is made to obtain a voice channel from a voice channel manager, GETCHAN, the obtained channel id is saved into the context.

  Only two operations include data blocks that contain the voice channel identifier. They are the VoiceConnectRequest and the return error in response to the VoiceConnectRequest.

Appendix: OAP Corba Server

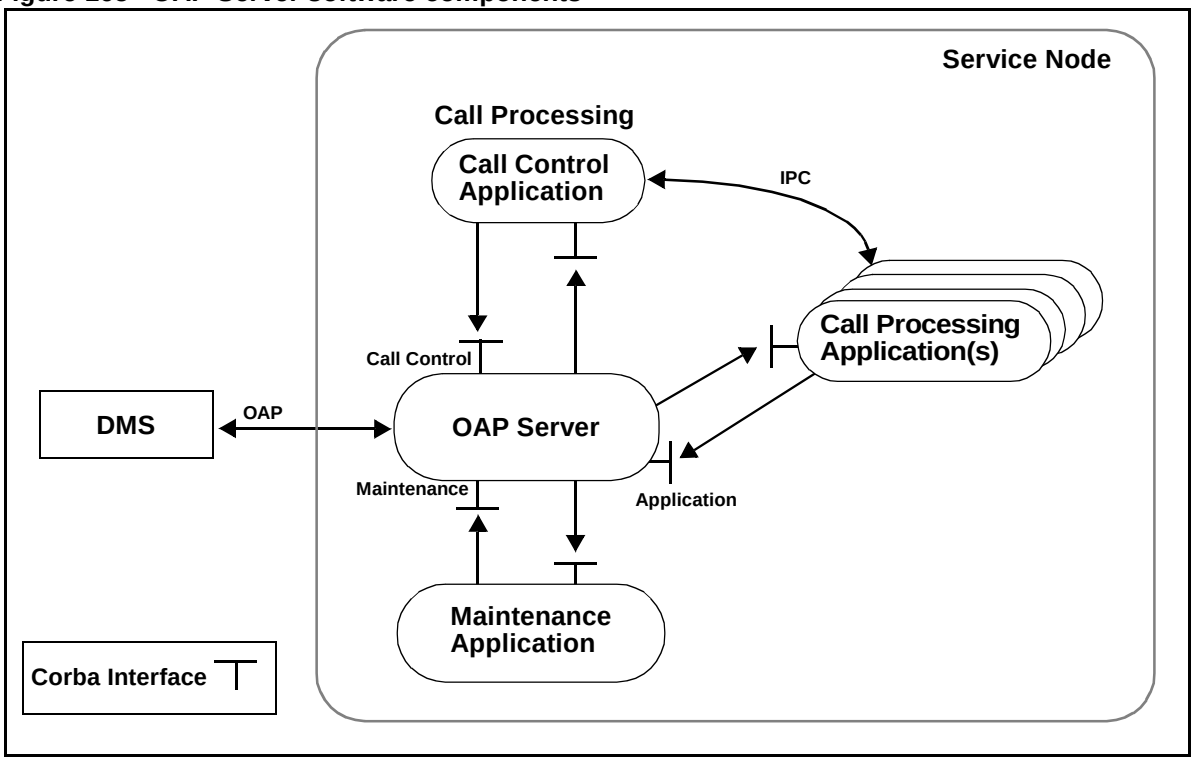
Overview

The OAP Server is a Corba based application built on the OAP API framework. The purpose of the OAP Server is to provide an intermediary between call processing applications and the OSSAIN DMS. A set of well defined Corba interfaces provide call processing applications with the full set of OAP operations and capabilities.

Note: The OAP Server utilizes Iona Orbix 2.3c as its source for Corba software support. Some Orbix specific extensions were utilized in order to improve real time performance of OAP Server/client application interactions.

OAP Server components

Figure 165 OAP Server software components



OAP Server

The OAP Server application acts as the intermediary between the DMS and call processing and maintenance applications. It is responsible for routing OAP message events from the DMS to the correct application and for routing OAP message events from applications to the DMS.

Call Control Application

The call control application is a call processing application that controls an OSSAIN session pool. The call control application receives all OAP inform messages and is responsible for taking action for those OAP events. There is only one call control application per session pool.

Call Processing Application

The call processing application contains the bulk of the call processing logic. It makes decisions on when and how to use the resources of the service node to provide service to a call. There can be multiple call processing applications that take part in the servicing of a call.

Note: The call control and call processing applications can be combined into a single application. The OAP Server design allows for these applications to be split into separate application processes. If call control and call processing applications are developed as separate processes, an inter process communication (IPC) device is required to coordinate call processing between the applications.

Maintenance Application

The maintenance application is responsible for managing and monitoring the maintenance state or OAP Server resources (node and session pool objects). Only one maintenance application may control OAP Server resources. Multiple applications may monitor OAP server resources.

Call Control Interface

The call control interface is the OAP Server Corba interface utilized by a call control application to register for control of an OSSAIN session pool. Additionally, this interface is used by call control and call processing applications to initiate service node originated calls.

Application Interface

The application interface is the OAP Server Corba interface utilized by call processing applications to send and receive OAP message events.

Maintenance Interface

The maintenance interface is the OAP Server Corba interface utilized by maintenance application to control and monitor the OAP Server. Only one maintenance application may register for control of the OAP Server. Multiple applications, including call processing applications, can register to monitor the OAP Server.

Call Processing Call Back Reference

The call processing call back reference is a reference to a Corba object provided by either the call control application or a call processing application. The call back Corba object is utilized by the OAP Server to send OAP message events to call control and call processing applications.

Maintenance Call Back Reference

The maintenance call back reference is a reference to a Corba object provided by a registered maintenance application. The call back Corba object is utilized by the OAP Server to inform the maintenance application of OAP maintenance events.

Application processing

The OAP Server architecture facilitates a distributive application processing environment. The use of three distinct Corba interfaces allows the use of one or multiple call processing and maintenance applications. The following sections describe the three Corba interfaces provided by the OAP Server.

OAP message byte streams

Several of the methods found in the OAP Server Corba interfaces contain a parameter of the type OA_OapBuffer. This type is defined to more efficiently send OAP message byte streams between the OAP Server and client applications. The OA_OapBuffer type is defined on Iona's Orbix opaque type which means the implementation of the OA_OapBuffer class is hidden from Orbix. This technique provides the capability to customize how OAP message byte streams are marshalled and unmarshalled when Corba interface methods are invoked.

A new OA_OapBuffer object is created by invoking the constructor of the OA_OapBuffer class. This is illustrated in Figure 166 "Creating an OA_OapBuffer object'.

Figure 166 Creating an OA_OapBuffer object

```
OA_UBYTE* buffer;            // Initialized to oap message buffer
OA_UWORD bufferSize;         // Initialized to oap message buffer size
OA_OapBuffer* oapBuffer = new OA_OapBuffer(buffer, bufferSize);
```

The buffer is an unsigned character string that contains the OAP message byte stream being passed via a Corba interface method. The application utilizes the PI layer of the OAP API to build the OAP message byte stream.

The contents of the OA_OapBuffer object are obtained by invoking the getBuffer() method of the OA_OAPBuffer class. This is illustrated in Figure 167 "Retrieving OAP byte stream from OA_OapBuffer object'.

Figure 167 Retrieving OAP byte stream from OA_OapBuffer object

```
const OA_OapBuffer* buffer;      // Initialized elsewhere
unsigned char* inBuffer = OA_NULL;
unsigned short inBufferSize = 0;

buffer->getBuffer(inBuffer, inBufferSize);
```

Once the buffer is retrieved from the OA_OapBuffer object. The PI layer of the OAP API can be used to decode the message byte stream into a high level operation.

Call control processing

Call control application

The call control interface provides the ability to register for control of an OAP Server session pool. A single application registers with the OAP Server for control of a session pool. An application may register for control of multiple session pools but a session pool can only be controlled by one application. An application that utilizes the call control interface to control a session pool is referred to as a **call control** application.

The call control application is responsible for setting up and tearing down calls based on notification events received from the OAP Server. The call control application is the recipient of all OAP inform messages and determines how the incoming messages are to be processed. The OAP inform messages may be routed to other call processing applications or they may be processed by the call control application itself. For example, when a session begin inform message is received by the OAP Server, the call control application is notified of the new call arrival. The call control application can then gather required resources for the call and makes a new OAP request or it may opt to route the session begin message to a call processing application for handling.

When call control and call processing applications are developed as separate processes, it is possible to configure the call control application to receive the successful responses to OAP requests made by call processing applications. This is typically done for OAP requests that result in a call being releases from the service node. By routing OAP success responses to the call control application, the call control application is able to free call processing resources prior to a new call arrival. Call processing application s specify this behavior when utilizing the OAP Server application interface.

Call control interface

The OAP Server call control interface is used by client application to register as call control applications and to initiate calls to an OSSAIN switch. The call control interface is fully defined in the oapserver.idl file which distributed with the OAP Server application. Figure 168 “Call control interface IDL' shows the call control interface as it appears in the IDL file. Each of the methods shown are described below.

Figure 168 Call control interface IDL

```
interface OA_CallControl
{
  unsigned short registerApplication(in unsigned short poolId,
                        in OA_CallpCallBack callControlClient,
                        in boolean sendMessage);

  unsigned short unregisterApplication(in unsigned short poolId);

  oneway void initOapSession(in unsigned short nodeId,
                    in unsigned short poolId,
                    in unsigned long transactionId,
                    in OA_OapBuffer buffer,
                    in unsigned short bufferSize,
                    in OA_CallpCallBack callControlClient,
                    in boolean sendMessage);

};
```

registerApplication()

The registerApplication() method is used to register as the call control application of an OAP Server provisioned session pool. This is typically done during initialization of the service node and during restarts. The following parameters are required.

- poolId - The client application specifies the session pool that the application will be controlling.
- callControlClient - When registering with the OAP Server, the client application provides a Corba based call back reference object, OA_CallpCallBack, that the OAP Server utilizes to notify the call control application of incoming OAP inform messages.
- sendMessage - The application can opt to have the OAP Server notification be in the from of a simple notify method call or the application can specify that the OAP message byte stream be passed to it. This is indicated by the sendMessage boolean value. A value of true indicates that a byte stream be sent. False indicates that only a notify method be invoked.

The registerApplication returns a return code indicating whether the registration request succeeded or failed. This value maps to the OAP API return code enumerated type, OA_RCODE, found in the header file rcodes.h. A value of zero indicates that the registration request succeeded.

unregisterApplication()

The unregisterApplication() method is used by a call control application to relinquish control of a session pool. This is typically done during the shutdown or restart of a service node. The following parameter is required.

- poolId - The client specifies the session pool whose control is being relinquished.

The unregisterApplication() method returns a return code indicating whether the registration request succeeded or failed. This value maps to the OAP API return code enumerated type, OA_RCODE, found in the header file rcodes.h. A value of zero indicates that the registration request succeeded.

initOapSession()

The initOAPSession() method is used by a call control application to originate a call to a specified switch. When the initOapSession() is invoked the OAP Server attempts to assign an idle session to the call being originated. If a session is found, the session initiation request message is forwarded to the DMS. The originating application is sent the response once it is received from the DMS. The following parameters are required.

- nodeId - The node identifier of the OSSIAN switch where the call is to be originated.
- poolId - The session pool identifier of the session pool to be used for the call. The session pool must be provisioned on the DMS as a service node originated type session pool.
- transactionId - Unique identifier whose value range is managed by the application. The transaction identifier is echoed back to the application when the response to the session initiation request returned by the OAP Server. The transaction identifier can be used by the application to synchronize the response with the origination request.
- buffer - Contains the session initiation request message byte stream built using the PI layer of the OAP API.
- bufferSize - The length of the message byte stream contained in the buffer parameter.
- sendMessage - The application can opt to have the OAP Server notification be in the form of a simple notify method call or the application can specify that the OAP message byte stream be passed to it. This is indicated by the sendMessage boolean value. A value of true indicates that a byte stream be sent. False indicates that only a notify method be invoked.

Call processing call back interface

The call processing call back interface, OA_CallpCallBack, is the Corba interface used by the OAP Server to return Corba request results to the originating applications. The call back interface is used by the call control application and the call processing applications.

When the call control application registers with the OAP Server, it provides a reference to its call back interface object. The OAP Server uses this as the default call back interface for the session pool specified during the application registration process.

Call processing applications are able to override the call back interface when they make a request of the OAP Server. The OAP Server sends request responses to the application that made the most recent request of the OAP Server. In this way, multiple call processing applications may take part in the processing of a call.

The IDL of the call processing call back interface is shown in Figure 169 "Call processing call back interface IDL'. A description of each method follows.

Figure 169 Call processing call back interface IDL

```
interface OA_CallpCallBack
{
oneway void sendToClient(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                         in unsigned short oapVersion, in unsigned short operationId,
                         in unsigned long transactionId, in OA_OapBuffer buffer,
                         in unsigned short bufferSize);

oneway void informNotify(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                        in unsigned short oapVersion, in unsigned long invokeId,
                        in unsigned short operationId, in unsigned short functionId, in unsigned long callId,
                        in unsigned short channelId, in unsigned short trunkGroupId,
                        in unsigned short trunkMemberId);

oneway void returnResultNotify(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                          in unsigned short oapVersion, in unsigned long transactionId,
                          in unsigned short operationId, in unsigned long callId,
                          in unsigned short channelId, in unsigned short trunkGroupId,
                          in unsigned short trunkMemberId);

oneway void returnErrorNotify(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                         in unsigned short oapVersion, in unsigned long transactionId,
                         in unsigned short operationId, in unsigned long callId,
                         in unsigned short errorId);

oneway void returnRejectNotify(in unsigned short nodeId,in unsigned short poolId, in unsigned short sessionId,
                          in unsigned short oapVersion, in unsigned long transactionId,
                          in unsigned short operationId, in unsigned long callId,
                          in unsigned short problemType, in unsigned long rejectReason);
```

```
 oneway void timeOut(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                     in unsigned long transactionId, in unsigned short operationId,
                     in unsigned long callId);

oneway void processError(in unsigned short nodeId, in unsigned short poolId, in unsigned short sessionId,
                         in unsigned long transactionId, in unsigned short operationId,
                         in unsigned long callId, in CallpErrorCodes errorCode);

  oneway void restart();

  oneway void shutdown();

}; // interface OA_CallpCallBack
```

sendToClient()

The OAP Server invokes the sendToClient() method when sending an OAP message byte stream to a client application. The following parameters are provided.

- nodeId - The source node identifier. This is the node identifier of the switch that originated the message being sent to the client application.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- oapVersion - The OAP Version of the message contained in buffer.
- operationId - The operation identifier of the message contained in buffer.
- transactionId - If this is a response to a request initiated by the application, the transaction identifier is set to the value received from the application when the request was made. Otherwise, the transaction identifier is set to NULL_TRANSACTION_ID (0xFFFFFFFF).
- buffer - Contains the OAP message byte stream being sent to the application. The application utilizes the PI layer of the OAP API to decode the message byte stream into a high level operation.
- bufferSize - The size of the message byte steam contained in buffer.

informNotify()

The OAP Server invokes the informNotify() method when notifying an application that an inform message has been received from the DMS. The informNotify() method provides information about the inform message being received for the application to continue processing the call without having to retrieve the OAP message byte stream. The following parameters are provided.

- nodeId - The source node identifier. This is the node identifier of the switch that originated the inform message.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- oapVersion - The OAP Version of the inform message.
- invokeId - The OAP invoke identifier of the inform message. The invoke identifier can be used by the application retrieving the message byte stream of the inform message from the OAP Server.
- operationId - The operation identifier of the inform message.
- functionId - The function identifier of the inform message.
- callId - The call identifier of the inform message. This is the call identifier from the DMS switch that is hosting the call.
- channelId - Logical channel identifier of voice link currently in use for this call. The channelId parameter is set to NULL_VOICE_CHANNEL_ID if there is not a voice link currently connected to the call or if the switch picks voice link selection model is being used.
- trunkGroupId - The trunk group identifier of the voice link currently in use for this call. The trunkGroupId parameter is set to NULL_TRUNK_GROUP_ID if there is not a voice link currently connected to the call or if the service node picks voice link selection model is being used.
- trunkMemberId - The trunk member identifier of the voice link currently in use for this call. The trunkmemberId parameter is set to NULL_TRUNK_MEMBER_ID if there is not a voice link currently connected to the call or if the service node picks voice link selection model is being used.

returnResultNotify()

The OAP Server invokes the returnResultNotify() method when notifying an application that a return result message has been received from the DMS. The returnResultNotify() method provides information about the return result message being received for the application to continue processing the call without having to retrieve the OAP message byte stream. The following parameters are provided.

- nodeId - The source node identifier. This is the node identifier of the switch that originated the return result message.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.

- oapVersion - The OAP Version of the return result message.
- transactionId - The transaction identifier is set to the value received from the application when the request being responded to was made.
- operationId - The operation identifier of the return result message.
- callId - The call identifier of the return result message. This is the call identifier from the DMS switch that is hosting the call.
- channelId - Logical channel identifier of voice link currently in use for this call. The channelId parameter is set to NULL_VOICE_CHANNEL_ID if there is not a voice link currently connected to the call or if the switch picks voice link selection model is being used.
- trunkGroupId - The trunk group identifier of the voice link currently in use for this call. The trunkGroupId parameter is set to NULL_TRUNK_GROUP_ID if there is not a voice link currently connected to the call or if the service node picks voice link selection model is being used.
- trunkMemberId - The trunk member identifier of the voice link currently in use for this call. The trunkmemberId parameter is set to NULL_TRUNK_MEMBER_ID if there is not a voice link currently connected to the call or if the service node picks voice link selection model is being used.

returnErrorNotify()

The OAP Server invokes the returnErrorNotify() method when notifying an application that a return error message has been received from the DMS. The returnErrorNotify() method provides information about the return error message being received for the application to continue processing the call without having to retrieve the OAP message byte stream. The following parameters are provided.

- nodeId - The source node identifier. This is the node identifier of the switch that originated the return error message.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- oapVersion - The OAP Version of the return error message.
- transactionId - The transaction identifier is set to the value received from the application when the request being responded to was made.
- operationId - The operation identifier of the return error message.
- callId - The call identifier of the return error message. This is the call identifier from the DMS switch that is hosting the call.

- errorId - The error identifier received from the DMS in the return error OAP message.

returnRejectNotify()

The OAP Server invokes the returnRejectNotify() method when notifying an application that a return reject message has been received from the DMS. The returnRejectNotify() method provides information about the return reject message being received for the application to continue processing the call without having to retrieve the OAP message byte stream. The following parameters are provided.

- nodeId - The source node identifier. This is the node identifier of the switch that originated the return reject message.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- oapVersion - The OAP Version of the return error message.
- transactionId - The transaction identifier is set to the value received from the application when the request being responded to was made.
- operationId - The operation identifier of the return error message.
- callId - The call identifier of the return error message. This is the call identifier from the DMS switch that is hosting the call.
- problemType - The problem type received from the DMS in the return reject OAP message.
- rejectReason - The reject reason received from the DMS in the return reject OAP message.

timeOut()

The OAP Server invokes the timeOut() method when notifying an application that a outstanding request to the DMS has expired. The following parameters are provided.

- nodeId - This is the node identifier of the switch that was sent the request that has timed out.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- transactionId - The transaction identifier is set to the value received from the application when the request that has timed out was made.

- operationId - The operation identifier of the original request.
- callId - The call identifier of the current call.

processError()

The OAP Server invokes the processError() method when notifying an application that an error was encountered while processing the application's request.

Note: This is not the same as when a return error message is received from the DMS. See comments on sendToClient() and returnErrorNotify().

The following parameters are provided.

- nodeId - This is the node identifier of the switch that was the intended destination of the request.
- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- transactionId - The transaction identifier is set to the value received from the application when the request was made.
- operationId - The operation identifier of the original request.
- callId - The call identifier of the current call.
- errorCode - The error reason.

Application call processing

Call processing application

A call processing application performs the bulk of the call processing logic of a call. A call processing application may be separate from the call control application or they may be combined into a single application.

The call processing application uses the OAP Server Corba interfaces to send and receive OAP message byte streams to a DMS. When sending an OAP message byte stream, the call processing application provides a reference to a call processing call back object, OA_CallpCallBack, so that the OAP Server can send responses to the call processing application.

Application interface

The OAP Server provides an application interface so that call processing and call control applications can send and receive OAP message byte streams to the DMS. The application interface is fully defined in the oapserver.idl file which is distributed with the OAP Server application. Figure 170 "Application interface IDL' shows the application interface as it appears in the IDL file. Each of the methods shown are described below.

Figure 170 Application interface IDL

```
interface OA_Application
{
  oneway void sendToServer(in unsigned short poolId,
                           in unsigned short sessionId,
                           in unsigned long transactionId,
                           in OA_OapBuffer buffer,
                           in unsigned short bufferSize,
                           in OA_CallpCallBack callpClient,
                           in ReturnResultDisp retResDisp,
                           in boolean sendMessage,
                           in unsigned long requestTimeout);

  oneway void getOapMessage(in unsigned short poolId,
                            in unsigned short sessionId,
                            in unsigned long invokeId,
                            in unsigned short invokeType,
                            in unsigned long transactionId,
                            in OA_CallpCallBack callpClient);

  oneway void getLastOapMessage(in unsigned short poolId,
                                in unsigned short sessionId,
                                in unsigned long transactionId,
                                in OA_CallpCallBack callpClient);
};
```

sendToServer()

An application invokes the sendToServer() method when sending an OAP message byte stream to the OAP Server for forwarding on to the DMS switch. The following parameters are required.

- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- transactionId - Unique identifier whose value range is managed by the application. The transaction identifier is echoed back to the application when a response to the send to server request is returned by the OAP Server. The transaction identifier can be used by the application to synchronize the response with the original request.
- buffer - Contains the OAP message byte stream being sent to the application. The application utilizes the PI layer of the OAP API to build the message byte steam.
- bufferSize - The size of the message byte steam contained in buffer.

- callpClient - The application's call back reference object. When using the OAP Server application interface, the client application provides a Corba based call back reference object, OA_CallpCallBack, that the OAP Server utilizes to send responses to the application.
- retResDisp - Return result disposition. The application can set the return result disposition so that successful responses (return result) are sent to the call control application. In this way, the call control application and the call processing application can coordinate calls that are ending. Error and reject responses are always sent to the requesting application.
- sendMessage - The application can opt to have the OAP Server notification be in the form of a simple notify method call or the application can specify that the OAP message byte stream be passed to it. This is indicated by the sendMessage boolean value. A value of true indicates that a byte stream be sent. False indicates that only a notify method be invoked.
- requestTimeOut - The application can specify the amount of time the OAP Server waits for a response from the DMS to the request being made. If DEFAULT_REQUEST_TIMEOUT is specified, the timeout value found in the OAP Server's configuration file is used. If the request times out, the OAP Server invokes the timeOut method on the client's call back object.

getOapMessage()

An application can retrieve a previously received OAP message byte stream from the OAP Server using the getOapMessage() method. The following parameters are required.

- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- invokeId - The invoke identifier of the message being requested. When retrieving an inform message, the invoke identifier value is obtained when the OAP Server invokes the client's informNotify method. In all other cases, the invoke identifier is the value specified by the client application when it created the original request message.
- invokeType - The invoke type specifies whether the OAP Server should look in its list of inform messages received from the DMS, SWITCH_INVOKEID, or its list of responses from the DMS, SN_INVOKEID. Invoke identifiers in the inform list are managed by the DMS. Invoke identifiers in the result list are managed by the client application.
- transactionId - Unique identifier whose value range is managed by the application. The transaction identifier is echoed back to the application when a response to the get OAP message request is returned by the OAP Server. The transaction identifier can be used by the application to synchronize the response with the original request.

- callpClient - The application's call back reference object. When using the OAP Server application interface, the client application provides a Corba based call back reference object, OA_CallpCallBack, that the OAP Server utilizes to send responses to the application.

getLastOapMessage()

An application can retrieve the most recently received OAP message byte stream from the OAP Server using the getLastOapMessage() method. The following parameters are required.

- poolId - This is the session pool identifier of the session pool being used for this call.
- sessionId - This is the session identifier of the session being used for this call.
- transactionId - Unique identifier whose value range is managed by the application. The transaction identifier is echoed back to the application when a response to the get OAP message request is returned by the OAP Server. The transaction identifier can be used by the application to synchronize the response with the original request.
- callpClient - The application's call back reference object. When using the OAP Server application interface, the client application provides a Corba based call back reference object, OA_CallpCallBack, that the OAP Server utilizes to send responses to the application.

Maintenance processing

The OAP protocol supports a variety of maintenance class messages in order to coordinate availability of resources to handle calls. The OAP Server handles all DMS originated maintenance requests. A Corba based maintenance interface is provided so that an application can control when the OAP Server allows itself to be brought into service thus guaranteeing the availability of call processing resources prior to calls being routed to the service node.

The OAP Server contains a state machine for each contained OSSAIN resource (node, session pools, and sessions). The state of each resource is governed by interactions through the Corba maintenance interface and DMS originated OAP maintenance requests. This section describes the OAP Server maintenance state machines and how they are influenced by applications and the DMS.

Maintenance states

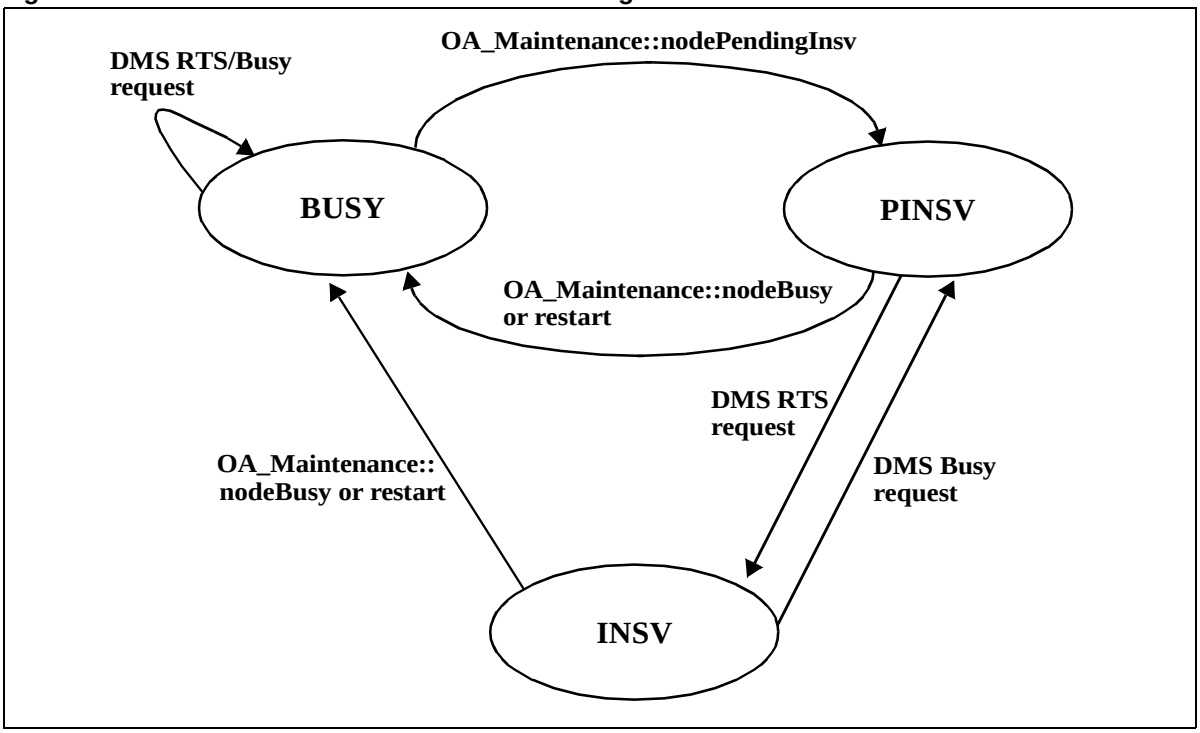
Each OSSAIN resource contained by the OAP Server has a maintenance state that is managed through a state machine. Descriptions of OAP Server node, session pool, and session state machines are described in the following sections.

OAP Server node states

OAP Server nodes have three states.

- BUSY - The node is disabled. DMS switch requests to RTS the node or subtending session pools are ignored by the OAP Server.
- PINSV (Pending in service) - The node has been made available by the maintenance control application but has not received an RTS request message from the DMS.
- INSV (In service) - The node has been made available by the maintenance control application and has successfully responded to a DMS RTS request.

Figure 171 OAP Server node state transition diagram



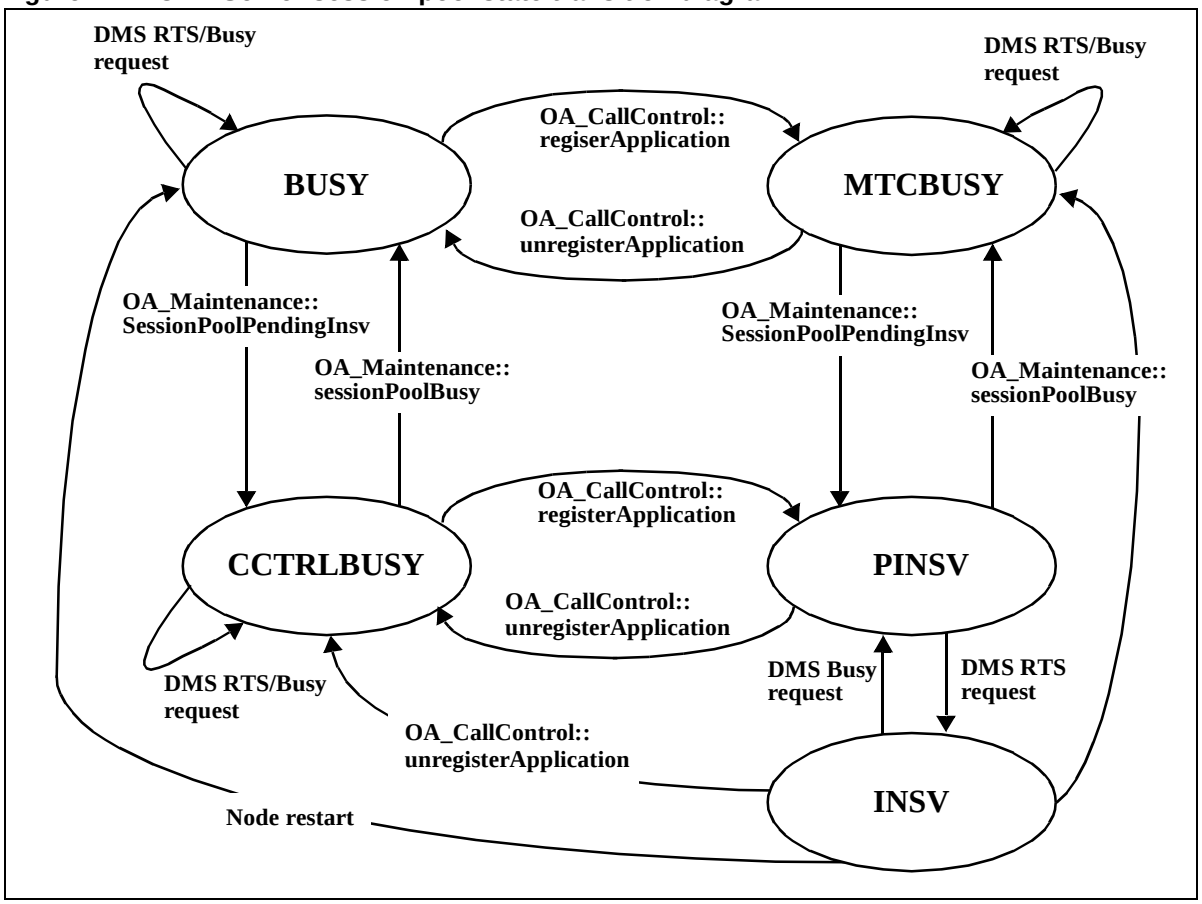
OAP Server session pool states

OAP Server session pools have five states.

- BUSY - The session pool is disabled. DMS switch requests to RTS the session pool are ignored by the OAP Server.
- MTCBUSY (Maintenance control busy) - This is an intermediate state to the pending in service state. A call control application has registered for control of the application but the maintenance control application has not invoked the sessionPoolPendingInsv() method.

- CCTRLBUSY (Call control busy) - This is an intermediate state to the pending in service state. The maintenance control application has invoked the sessionPoolPendingInsv() method but no call control application has registered for control of the session pool.
- PINSV (Pending in service) - A call control application has registered for control of the session pool and the maintenance control application has invoked the sessionPoolPendingInsv() method for the session pool. The session pool is ready to come into service.
- INSV (In service) - The session pool is in service.

Figure 172 OAP Server session pool state transition diagram

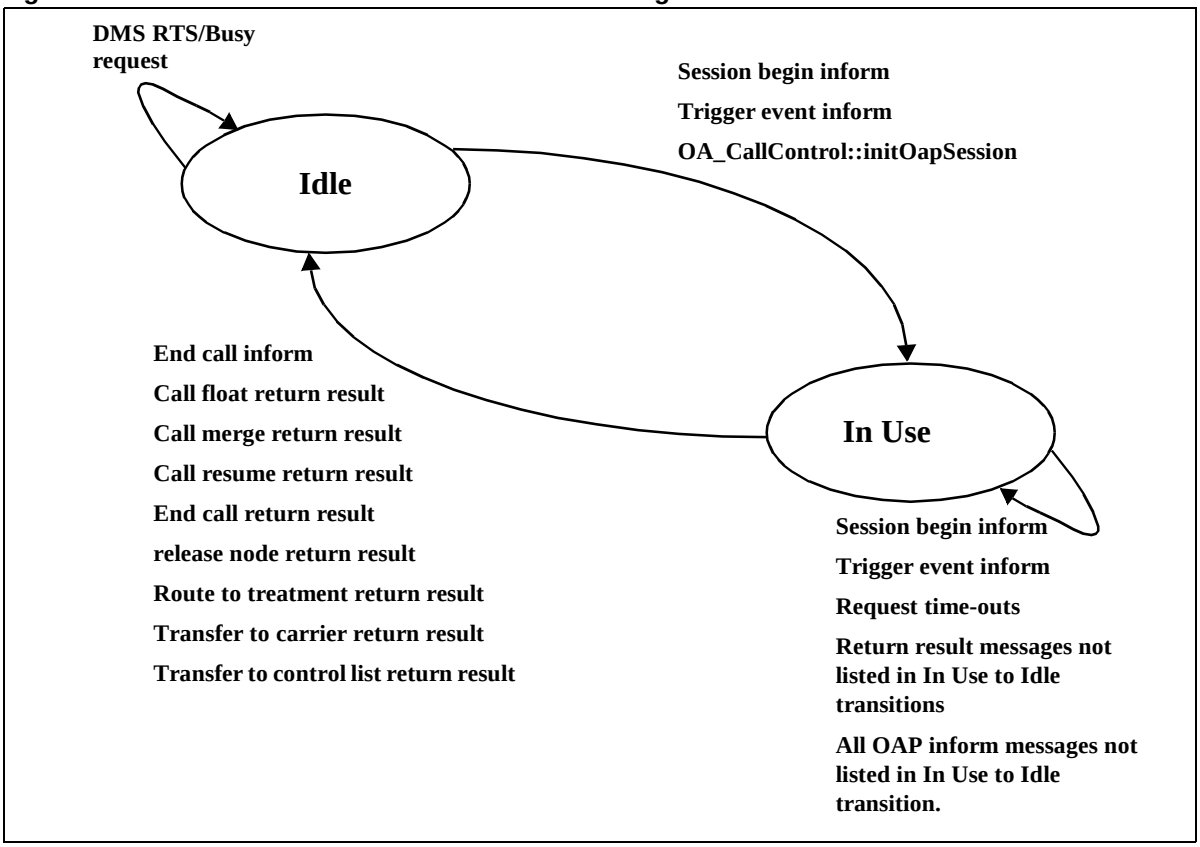


OAP Server session states

OAP Server sessions have two states.

- Idle - The session is free. There is not a call utilizing the session.
- In Use - There is a call currently using the session.

Figure 173 OAP Server session state transition diagram



Maintenance application

The OAP Server contains resource objects that model the node and session pools provisioned on the DMS switch. A maintenance application is required to control the maintenance behavior of the OAP Server and subsequently the state the contained resource objects. The maintenance application controls the OAP Server maintenance behavior through the OAP Server maintenance interface. The maintenance application and other applications may monitor the OAP Server for state changes of OAP Server resources.

Maintenance interface

The OAP Server maintenance interface provides a set of methods that mirror maintenance actions consistent with the OSSAIN domain. The maintenance interface is fully defined in the oapserver.idl file which is distributed with the OAP Server application. Figure 174 "Maintenance interface IDL' shows the application interface as it appears in the IDL file. Each of the methods shown are described below.

Note: Monitoring applications can only invoke the registerMtcApplication() and unregisterMtcApplication() methods.

Figure 174 Maintenance interface IDL

```
enum MaintenanceType {mtcControl, statusNotify};
typedef sequence<MaintenanceType> seq_maintenanceType;
typedef sequence<unsigned long> seq_ulong;

interface OA_Maintenance
{
  unsigned short registerMtcApplication(in seq_maintenanceType mtcTypes,
                                        in OA_MaintCallBack maintClient,
                                        in unsigned short nodeId,
                                        in seq_ulong poolIds,
                                       out unsigned short applicationId);

  unsigned short unregisterMtcApplication(in unsigned short applicationId);

  unsigned short shutdown(in unsigned short applicationId);

  unsigned short restart(in unsigned short applicationId);

  unsigned short nodeBusy(in unsigned short applicationId);

  unsigned short nodePendingInsv(in unsigned short applicationId);

  unsigned short sessionPoolBusy(in unsigned short applicationId,
                                 in unsigned short poolId);

  unsigned short sessionPoolPendingInsv(in unsigned short applicationId,
                                        in unsigned short poolId,
                                        in unsigned short throttleValue);

  unsigned short sessionPoolThrottle(in unsigned short applicationId,
                                     in unsigned short poolId,
                                     in unsigned short throttleValue);

};
```

registerMtcApplication()

The registerMtcApplication() method is used to register with the OAP Server as a maintenance application. An application may register as a maintenance control application or a monitoring application or both. The enumerated type, MaintenanceType is used to indicate the maintenance mode being registered for. Only one application may register as the maintenance control application for a session pool but multiple applications may register to monitor a session pool. The following list describes the parameters required by the registerMtcApplication() method.

- mtcTypes - A sequence of the maintenance types to use when registering for the specified session pools. mtcControl indicates that the application will be controlling the maintenance states of the session pool(s). statusNotify indicates that the application will be monitoring the session pool(s).

- maintClient - This is a reference to the application's maintenance call back object. The OAP Server invokes method's on the call back object when maintenance events occur.
- nodeId - The node identifier of the node where the OAP Server is resident.
- poolIds - A list of session pool identifiers being registered.
- applicationId - The application identifier is returned by the OAP Server. This is a unique identifier assigned to the maintenance application. The maintenance application must provide the application identifier in all subsequent OAP Server maintenance interface method calls.

unregisterMtcApplication()

The unregisterMtcApplication() method is used to unregister a maintenance application from the OAP Server. If the application is a maintenance control application for any its registered session pools, invoking this method will cause the state of the session pool to be modified.

A single parameter, applicationId, is required for this method. This is the application identifier that was assigned by the OAP Server when the registerMtcApplication() was invoked.

shutdown()

The shutdown() method directs the OAP Server to shutdown.

A single parameter, applicationId, is required for this method. This is the application identifier that was assigned by the OAP Server when the registerMtcApplication() was invoked.

restart()

The restart() method directs the OAP Server to go through its restart processing. All applications must re-register with the OAP Server once restart processing has completed.

A single parameter, applicationId, is required for this method. This is the application identifier that was assigned by the OAP Server when the registerMtcApplication() was invoked.

nodeBusy()

The nodeBusy() method is used to place the node into the busy state. Any calls that are in progress are terminated.

A single parameter, applicationId, is required for this method. This is the application identifier that was assigned by the OAP Server when the registerMtcApplication() was invoked.

nodePendingInsv()

The nodePendingInsv() method places the node in a pending in service state. Once in the pending in service state, the node will respond to RTS requests from the DMS.

A single parameter, applicationId, is required for this method. This is the application identifier that was assigned by the OAP Server when the registerMtcApplication() was invoked.

sessionPoolBusy()

The sessionPoolBusy() method causes the specified session pool to go to either the maintenance busy state (MTCBSY) or the busy state (BUSY) depending on the current state of the session pool. If going from the in service state (INSV) to the maintenance busy state, active calls on the session pool are terminated.

The application identifier of the maintenance application and the session pool identifier of the session pool whose state is being changed are provided as parameters to the sessionPoolBusy() method.

sessionPoolPendingInsv()

The sessionPoolPendingInsv() method is used to direct a session pool into the pending in service (PINSV) state. If a call control application has not yet registered for control of the specified session pool, the session pool is placed into the call control busy state (CCRTLBUSY). Once a session pool is in the pending in service state, the OAP Server will respond to DMS request to bring the session pool into service.

The application identifier of the maintenance application and the session pool identifier of the session pool whose state is being changed are provided as parameters to the sessionPoolPendingInsv() method. Additionally, the initial throttle value is specified. This value is sent by the OAP Server to the DMS in the RTS response message indicating the initial number of sessions available for call processing. This parameter only has significance when applied to a subscriber origination type session pool.

sessionPoolThrottle()

The sessionPoolThrottle() method is used to change the number of available sessions on a session pool.

The application identifier of the maintenance application and the session pool identifier of the session pool whose session throttle value is being modified are provided as parameters to the sessionPoolThorottle() method. The throttle value specifies the new session throttle value setting.

Maintenance call back interface

The maintenance call back reference, OA_MaintCallBack, object is utilized by the OAP Server to inform registered maintenance applications of maintenance events that affect OAP Server node and session pool resources. Maintenance applications provide a reference to their call back object when they register with the OAP Server using the OAP Server maintenance interface registerMtcApplication() method.

The IDL of the maintenance call back interface is shown in Figure 175 “ Maintenance call back interface IDL'. A description of each method follows.

Figure 175 Maintenance call back interface IDL

```
interface OA_MaintCallBack {

   boolean nodeStatusNotify(in NodeState oldState,
                            in NodeState newState);

   boolean poolStatusNotify(in unsigned short poolId,
                            in PoolState oldState,
                            in PoolState newState,
                            in unsigned short oapVersion);

   boolean osacPoolStateInform(in unsigned short switchId,
                               in unsigned short poolId,
                               in PoolState state,
                               in unsigned short oapVersion);

   boolean poolTestRequest(in unsigned short poolId,
                           out string mtcText);

   boolean poolAuditRequest(in unsigned short poolId,
                            out string mtcText);

   oneway void restart();

   oneway void shutdown();

}; // interface OA_MaintCallBack
```

nodeStatusNotify()

The nodeStatusNotify() method is invoked for all registered maintenance applications when the state of the OAP Server node resource changes. The previous state of the node and the new state of the node are provided.

This is a two way method that returns a boolean value. A value of true indicates the application successfully processed the notification event. A return value of false indicates that the application encountered a problem while processing the notification event.

poolStatusNotify()

The poolStatusNotify() method is invoked when the state of the OAP Server session pool resource changes. Only the maintenance applications that have registered to be informed of state changes for the specified session pool are notified.

The session pool identifier of the session pool changing state, the previous state of the session pool and the new state of the session pool are provided. The version of the OAP protocol currently negotiated with the host DMS switch is also provided.

This is a two way method that returns a boolean value. A value of true indicates the application successfully processed the notification event. A return value of false indicates that the application encountered a problem while processing the notification event.

osacPoolStateInform()

The osacPoolStateInform() method is invoked when the OAP Server receives a session pool state inform message from an OSAC remote switch. Only the maintenance applications that have registered to be informed of state changes for the specified session pool are notified.

The node identifier of the switch sending the session pool state inform message, affected session pool identifier, and the new state (in service or out of service from the switches perspective) are provided. The optimal OAP protocol version between the originating DMS switch and OAP Server is also provided. This is optimal OAP protocol version is useful when originating calls from the service node to an OSAC remote switch so that the OAP messages built by the service node are built at the optimal OAP protocol version.

This is a two way method that returns a boolean value. A value of true indicates the application successfully processed the notification event. A return value of false indicates that the application encountered a problem while processing the notification event.

poolTestRequest()

The poolTestRequest() method is invoked by the OAP Server when a session pool test request is received from the DMS. This method is only invoked for the application that is registered as the maintenance control application for the affected session pool.

This is a two way method that returns a boolean value. A value of true indicates the test was successful. A return value of false indicates that the test failed. The maintenance application may return a text string of up to thirty two characters in length which is forwarded to the DMS for display at the DMS map.

poolAuditRequest()

The poolAuditRequest() method is invoked by the OAP Server when a session pool audit request is received. This method is only invoked for the application that is registered as the maintenance control application for the affected session pool.

This is a two way method that returns a boolean value. A value of true indicates the audit was successful. A return value of false indicates that the audit failed. The maintenance application may return a text string of up to thirty two characters in length which is forwarded to the DMS for display at the DMS map.

restart()

The restart() method is invoked for all registered maintenance applications when the OAP Server is undergoing a restart. All applications must register with the OAP Server once the restart is complete.

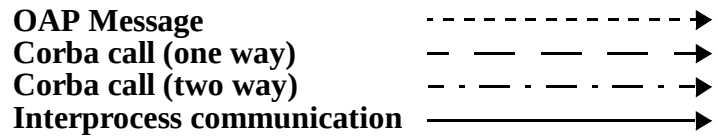
shutdown()

The shutdown() method is invoked for all registered maintenance applications when the OAP Server executes its shutdown process.

OAP Server event flows

This section describes the flow of events between the DMS, OAP Server, maintenance applications, call control application, and call processing applications. Events discussed include OAP Messages and use of server and client Corba interfaces.

The following notations are used to indicate the type of events flowing from one component to another.



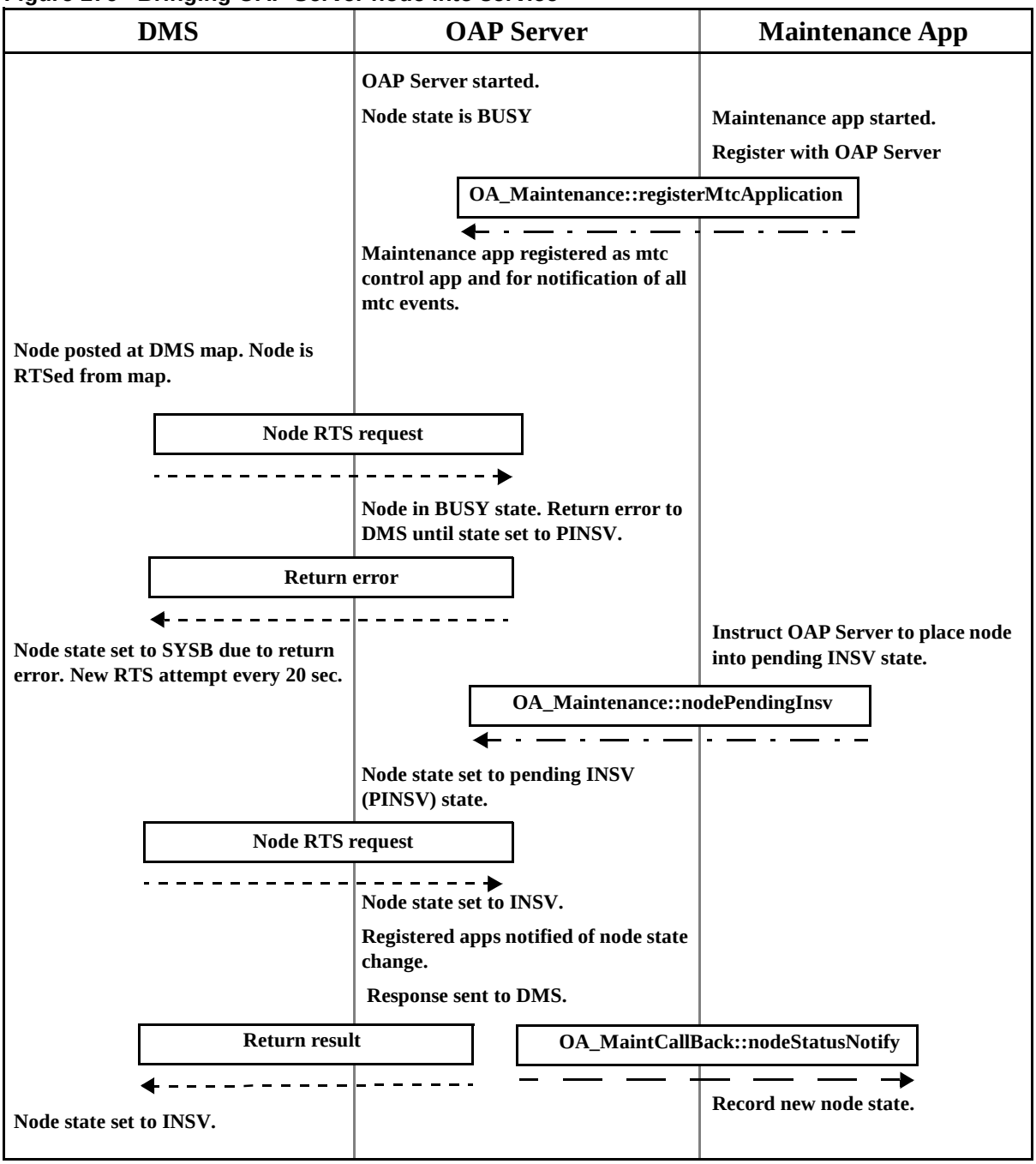
Maintenance event flows

Maintenance flows illustrate the series of events that occur when bringing an OAP Server node and session pool into service.

Node maintenance

Figure 176 “Bringing OAP Server node into service' shows the events that occur while bringing an OAP Server node into service. Once the OAP Server is started, a maintenance application must register with the OAP Server before the node can be brought into service from the DMS. Attempts to RTS the node prior to the maintenance application registering with the OAP Server cause the OAP Server to send a return error message to the DMS.

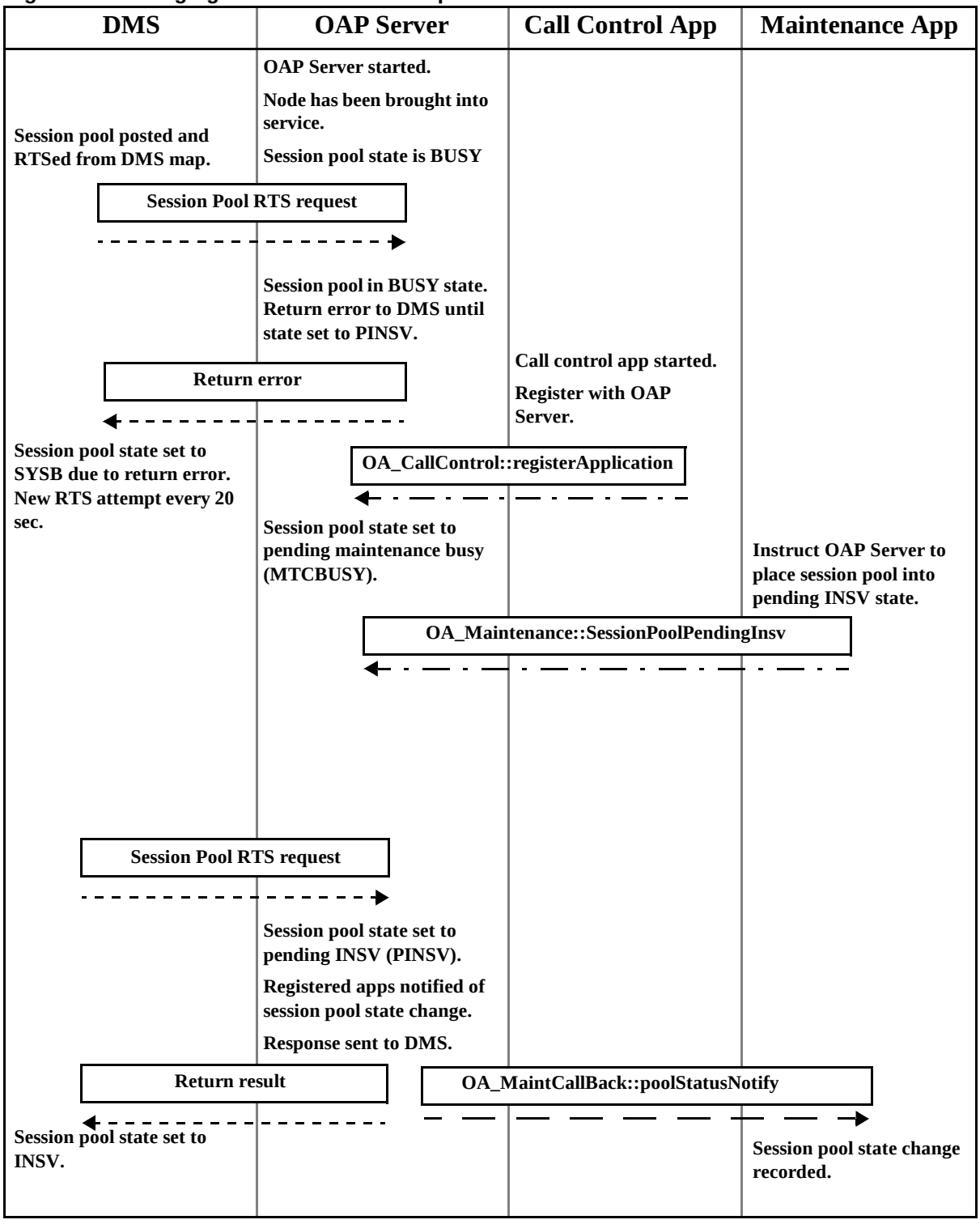
Figure 176 Bringing OAP Server node into service



Session pool maintenance

Figure 177 “Bringing OAP Server session pool into service' shows the events that occur while bringing an OAP Server session pool into service. This event flow assumes that the maintenance application has already registered with the OAP Server and the node has already been brought into service. This flow also includes the call control application registering with the OAP Server. The call control application must register for control of a session pool before the session pool can be brought into service.

Figure 177 Bringing OAP Server session pool into service



Call processing event flows

This section describes the flow of events between the OAP Server and other application components that occur during the processing of a call. This topic

is divided into three sub-sections.

- Call origination processing - Describes interactions that occur during the origination of a call.
- Mid-call processing - Describes interactions that occur as the call is processed at the service node.
- Release call processing - Describes how calls are released from the service node.

Call origination processing

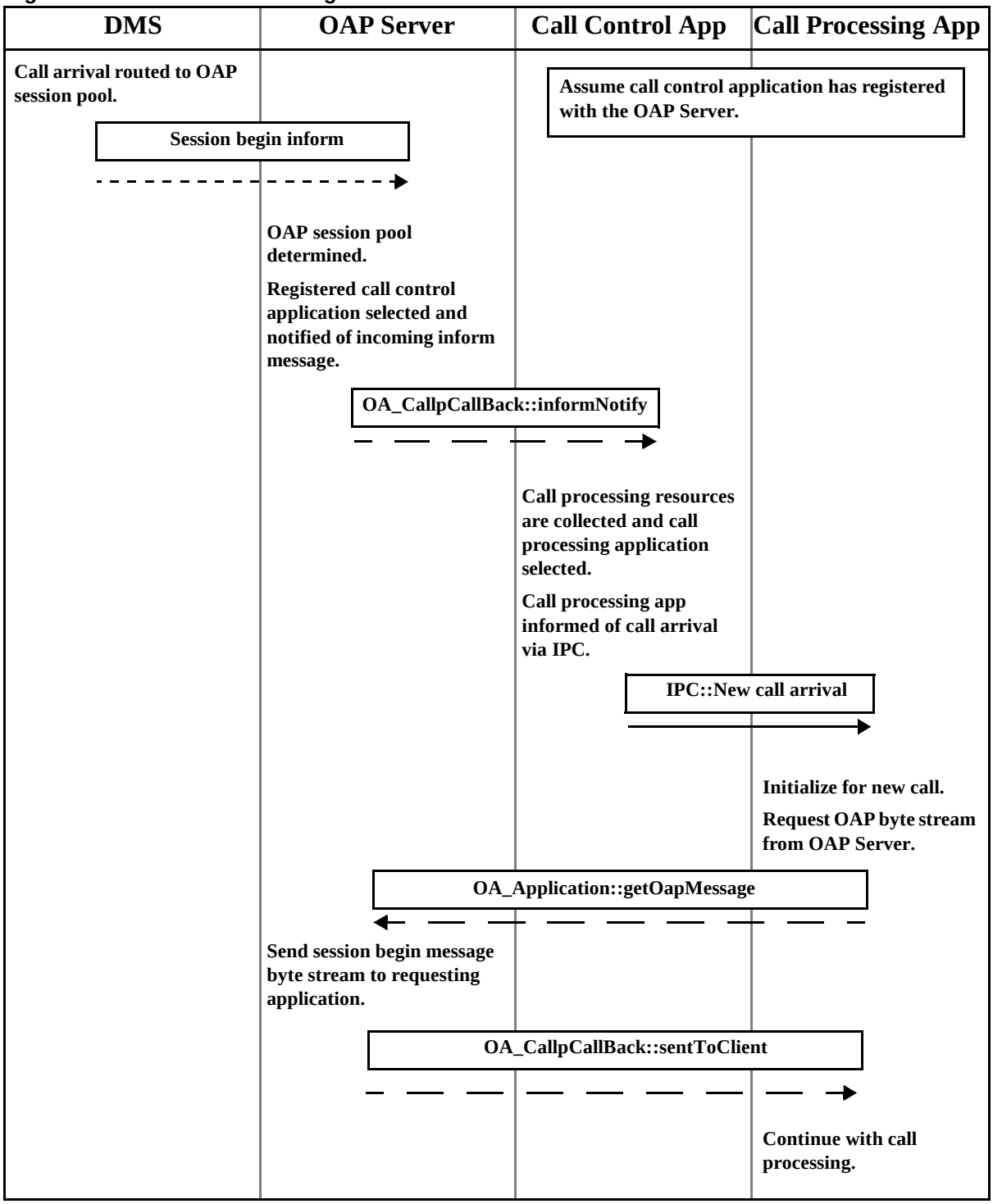
Calls can be originated from either the DMS switch (subscriber orig calls) or a call control application (SN orig calls). In the case of the subscriber orig call, the call control application waits to be notified by the OAP Server of a new call. This appears as either an invoke of the call control application's OA_CallpCallBack::informNotify() or OA_CallpCallBack::sentToClient() methods. The method selected is based on what was specified in the sendMessage parameter of the registerApplication() method when the call control application registered with the OAP Server.

Figure 178 "Subscriber call origination event flow' illustrates a subscriber originated call event flow. In this example, the call control application and the call processing application are separate processes. An IPC device (e.g. named pipe, shared memory, etc.) is used between the two application processes to exchange call processing events.

When the OAP Server receives a session begin inform message from the DMS the OAP Server notifies the call control application via the informNotify() method that an inform message has been received. The call control application examines the provided operation identifier to determine that the message received by the OAP Server is a session begin inform operation. The call control application gathers call processing resources and selects a call processing application to handle the call. The call control uses an IPC device to inform the call processing application that a new call has arrived.

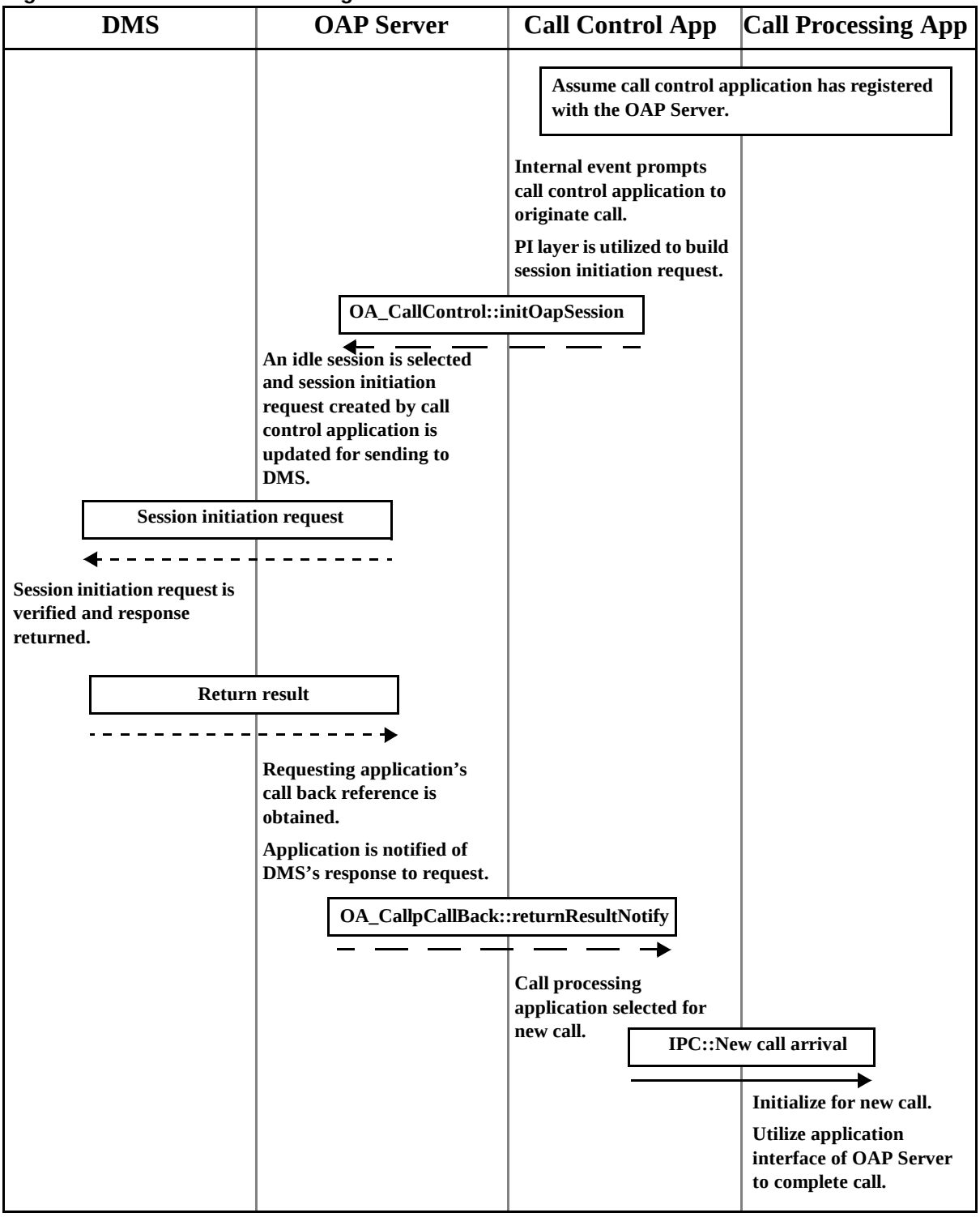
The call processing application may utilize the OAP Server application interface as needed to process the new call. In our example, the first task done is the retrieval of the session begin inform message byte stream from the OAP Server. Refer to Figure 178 for the illustration of these event flows.

Figure 178 Subscriber call origination event flow



In the case of SN orig calls, the call control application starts a new call by invoking the initOapSession() method. This is illustrated in Figure 179 “Service node call origination event flow'.

Figure 179 Service node call origination event flow



When the initOapSession() method is invoked, the OAP Server obtains an idle session to service the call. The session initiation request built by the application is updated to reflect the selected session and other contextual data. The request is then sent to the DMS.

Once the DMS responds to the request, the OAP Server utilizes the call back interface provided by the call control application to provide the DMS's response. In this example, the call control application selects a call processing application to handle the call and presents it with a new call via an IPC device. The call processing application then uses the OAP Server application interface to process the call.

Mid-call processing

Mid-call processing is the processing of events while the call is at the service node. This section describes normal call processing, timeout processing, and error processing.

Normal call processing

During a normal call flow, the call processing application does some work, sends an OAP request to the DMS, receives a result to the request, and performs some more work. The content of each request is dependent on the requirements of the call flow and is not of interest to the OAP Server. The OAP Server's role is to forward OAP requests originated by applications to the DMS and to forward DMS OAP responses and inform messages to applications.

The application interface supported by the OAP Server supports the exchange of OAP message byte streams between the OAP Server and applications. When sending an OAP operations, the application creates an OAP message byte stream that represents the OAP operation being sent. The OAP API PI layer is available for use by applications to facilitate the creation of OAP message byte streams.

Once an application has created an OAP message byte stream, it sends the message to the OAP Server via the OA_Application::sendToServer() method. The OAP Server takes the message byte stream provided by the application and updates message header attributes (e.g. sequence number). It then sends the OAP message to the DMS. If the OAP operation is a request, the OAP Server begins timing for a response to the request.

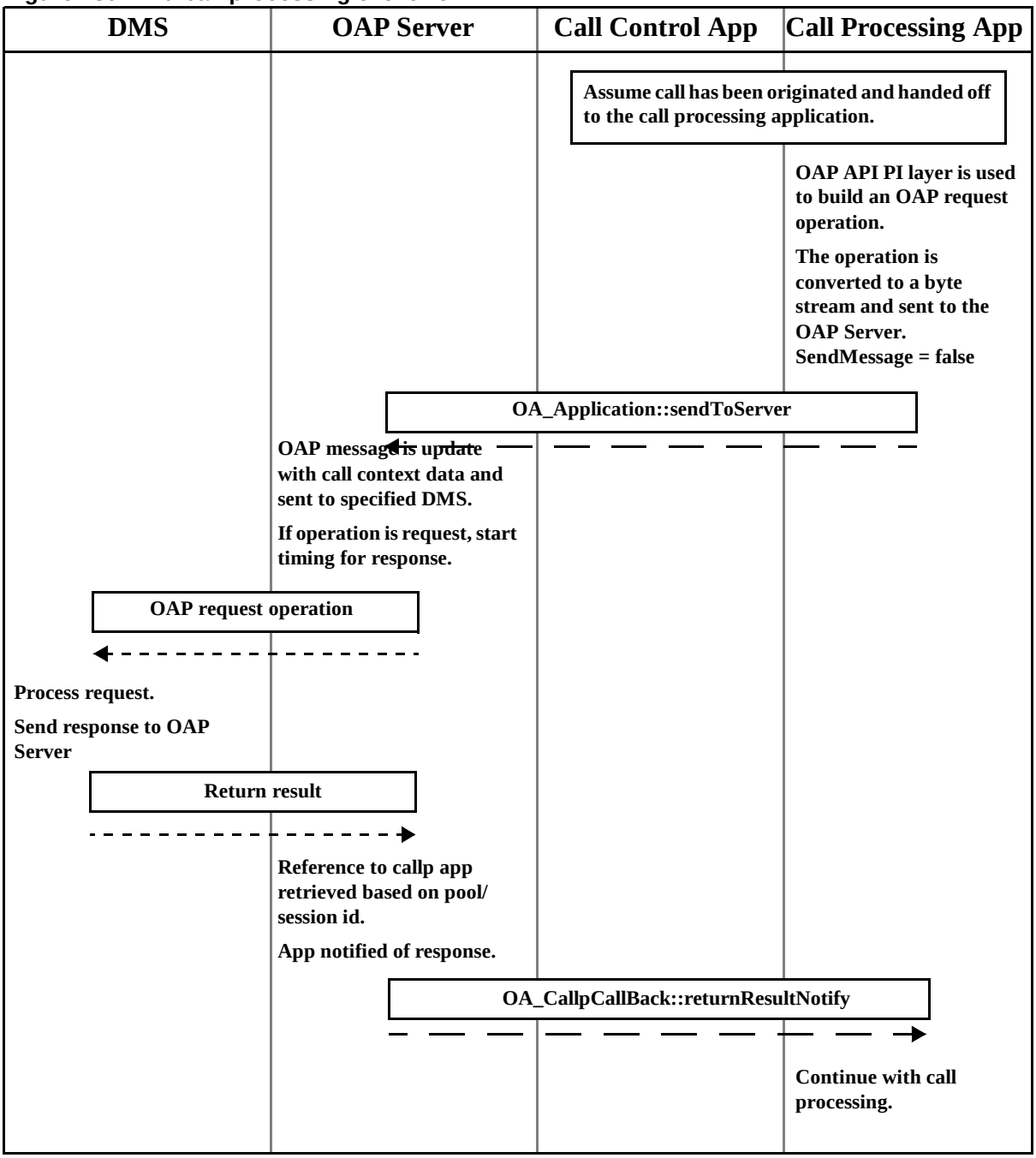
The DMS receives the requests, processes, and sends a response to the OAP Server. This can be a return result (success), return error, or return reject. If the requested operation was an inform operation, no response is expected.

The OAP Server takes the response and obtains a reference to the requesting application's call back interface based on the session pool and session identifiers. If the requesting application has requested that the OAP message byte steam of the response be sent to it, the OAP Server invokes the sendToClient() method of the application's call back interface. Otherwise, the appropriate notify method is executed.

- returnResultNotify()
- returnerrorNotify()
- returnRejectNotify()

Figure 180 “Mid-call processing event flow' illustrates the flow of events that occur during mid-call processing.

Figure 180 Mid-call processing event flow



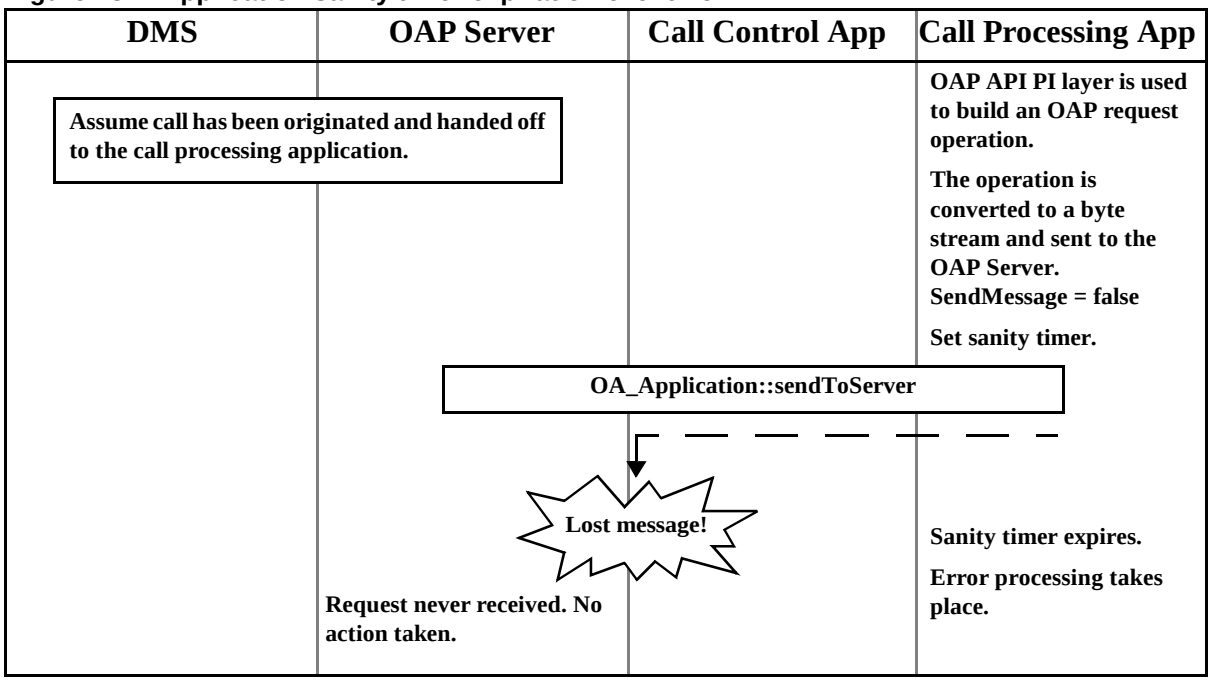
Timeout processing

It is possible that the request to send an OAP message can be lost, either before the OAP Server receives the request or when sent from the OAP Server to the DMS. The response form the DMS can also be lost. Again, this can occur before the OAP Server receives the response from the DMS or while the OAP server is attempting to send the response to the application. In order to detect and recover from lost messages, two levels of message timing are performed. Application sanity timing and OAP Server request operation timing.

The application should keep a sanity timer to time for a response to a request to the OAP Server to send an OAP message (sendToServer() method call). A response can come in the form of a call to the sendToClient(), returnResultNotify(), returnErrorNotify(), returnRejectNotify(), timeout(), or processError() methods on the application's call back interface. The application provides the OAP Server with an application transaction identifier when it makes a request to send an OAP message. The transaction identifier is returned to the application when any of the above methods are invoked. The application uses the transaction identifier to match a response to the original request. If a response is not received before the application's sanity timer expires, the application should consider the outstanding request to have failed. This scenario is illustrated in Figure 181 " Application sanity timer expiration event flow'.

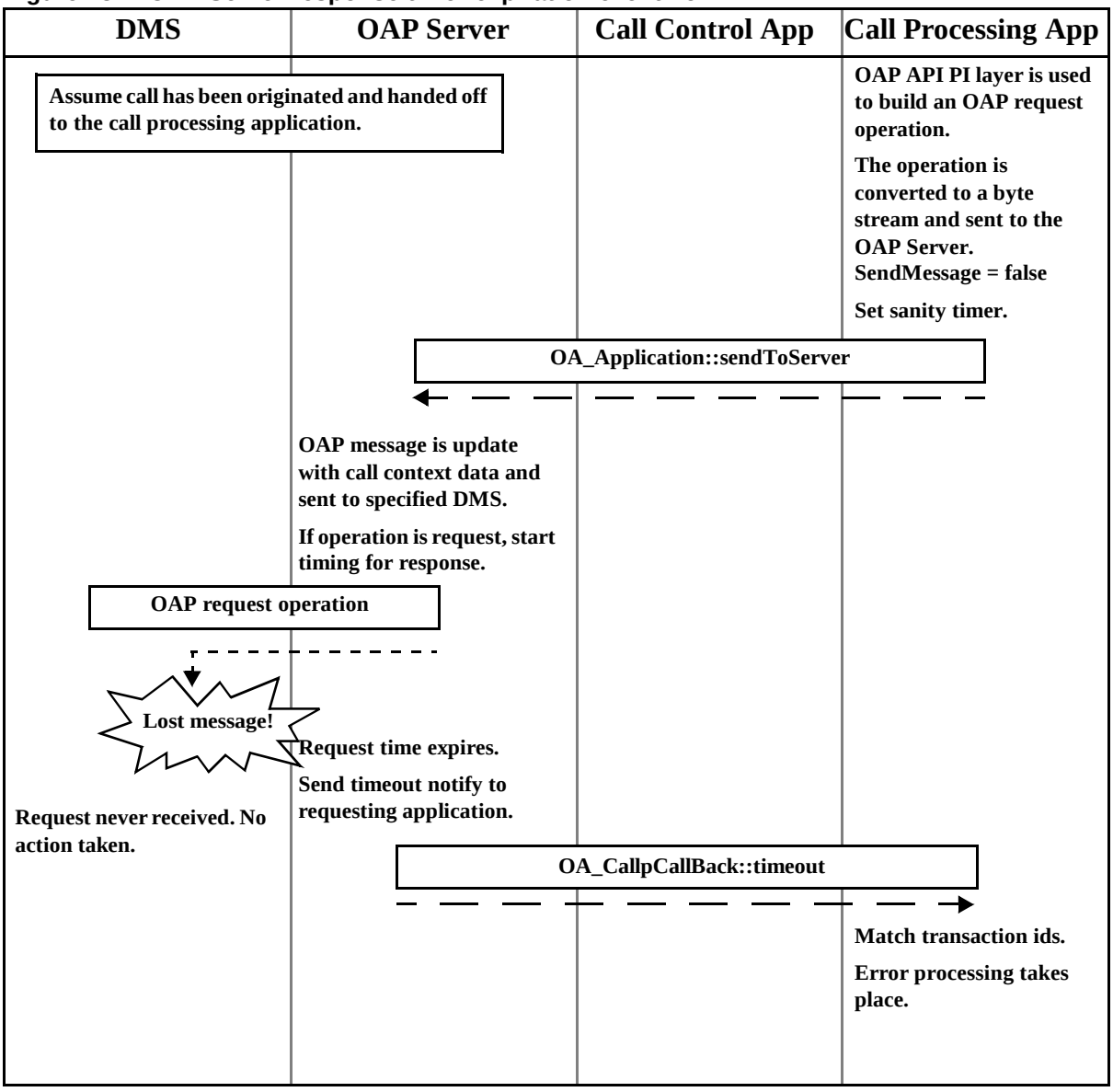
Note: Requests to send an OAP inform operation should not be timed for since they are inform messages and no response is expected.

Figure 181 Application sanity timer expiration event flow



When an application sends an OAP request operation, the OAP Server times for a response from the DMS. If the request timer expires before the OAP Server receives a response from the DMS, the OAP Server invokes the timeout method of the requesting application's call back interface. The transaction identifier of the original request is provided so that the application can match the timeout notification to the original request. This scenario is illustrated in Figure 182 "OAP Server response timer expiration event flow'.

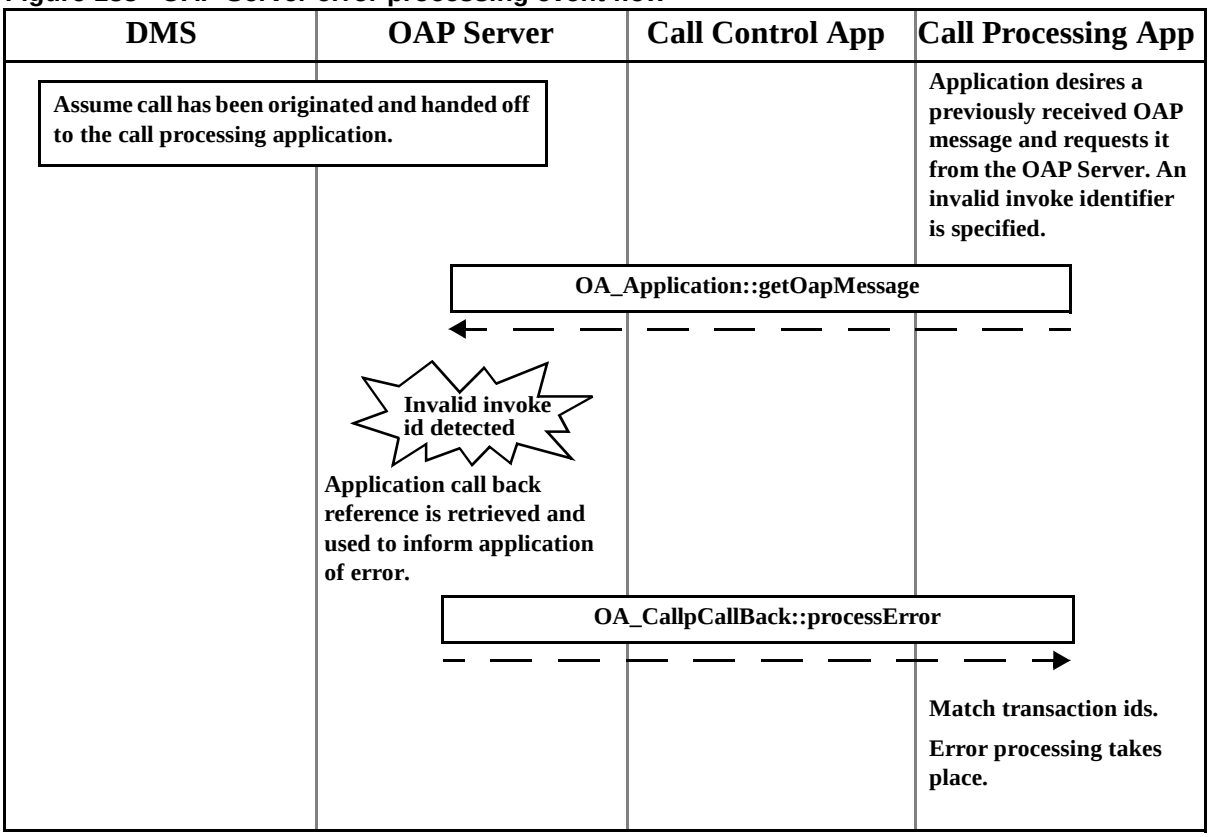
Figure 182 OAP Server response timer expiration event flow



Error processing

It is possible that the OAP Server is unable to successfully process an application's request. For example, an application should request to retrieve an OAP message from the server with an invalid invoke identifier. If such an error occurs, the OAP Server invokes the processError() method on the application's call back interface. An error code is provided indicating the cause of the error. The applications transaction identifier is also provided so that the application can match the error to the original request. Figure 183 "OAP Server error processing event flow' illustrates an error processing event flow.

Figure 183 OAP Server error processing event flow



Release call processing

A call can be released from the service node by the DMS, by the call processing application, or by the call control application. This section describes the different release call processing scenarios.

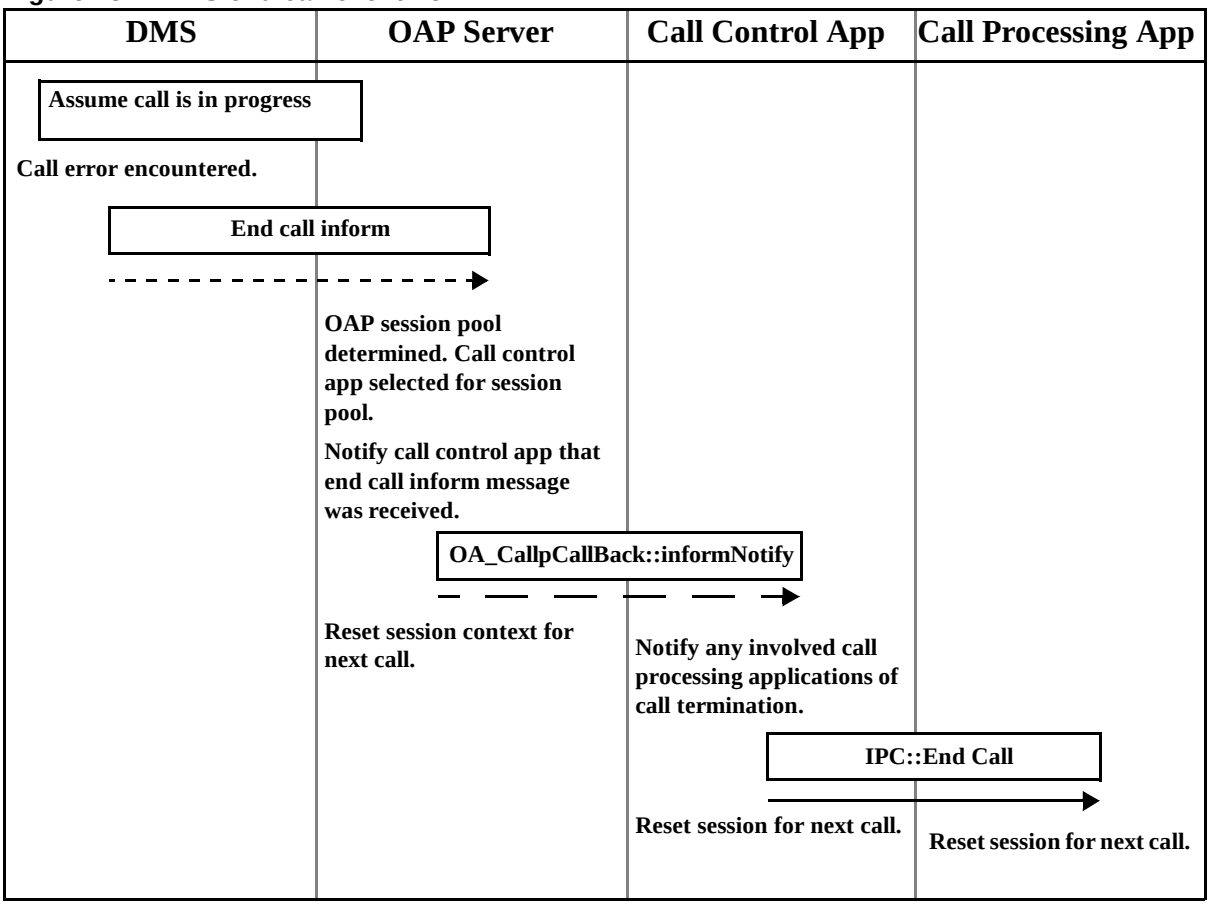
Release by DMS

The only time the DMS causes a call to be released from a service node is when a software error occurs. If the DMS encounters a software error, the call is terminated and an end call inform message is sent to the OAP Server. The OAP Server resets its internal call context for the affected session and notifies the call control application of the call termination.

The call control application is responsible for cleaning up resources to prepare for a new call and notifying any call processing applications that may currently be involved with the call.

Figure 184 “DMS end call event flow' illustrates the DMS end call event flow.

Figure 184 DMS end call event flow



Release by call processing application

The most usual case for releasing a call is triggered by the call processing application itself. Several OAP request operations result in the releasing of a call from the service node. They are:

- Call float request
- Call merge request
- Call resume request
- End call request
- Release node request
- Route to treatment request
- Transfer to carrier request

- transfer to control list request

The sendToServer() method allows the call processing application to specify a return result disposition when sending an OAP message to the OAP Server. Two dispositions are supported.

- The application receives all responses to the request. This disposition is usually used when call control and call processing functions are handled by a single application.
- The call control application receives successful responses to the request. The call processing application receives unsuccessful responses (reject, timeout, etc.) to the request. This disposition is usually used if call control and call processing functions are split among multiple applications.

Figure 185 “Call released by call processing application' illustrates the event flow where the call is released by the request of the call processing application but a call control application handles the successful response.

Release by call control application

Certain scenarios may make it necessary for the call control application to end the call. For example, if a resource used by the call such as a voice link becomes unavailable the call control application may choose to transfer the call to a live agent for assistance. The call control application can use the OAP Server application interface to send a request to end or transfer the call. The call control application must also inform the call processing application that the current call has ended.

Since the call control application may not know the current state of the call at the call processing application, it may be necessary to end the call using an alternative scheme. The call control application can notify the call processing application that the call must end due to a resource problem in the call control application. The call processing application can then send the request to the OAP Server to end the call and indicated that the response is to be returned to the call control application.

Figure 186 “Call released by call control application' illustrates the event flow where the call control application triggers the call processing application to transfer the current call.

Figure 185 Call released by call processing application

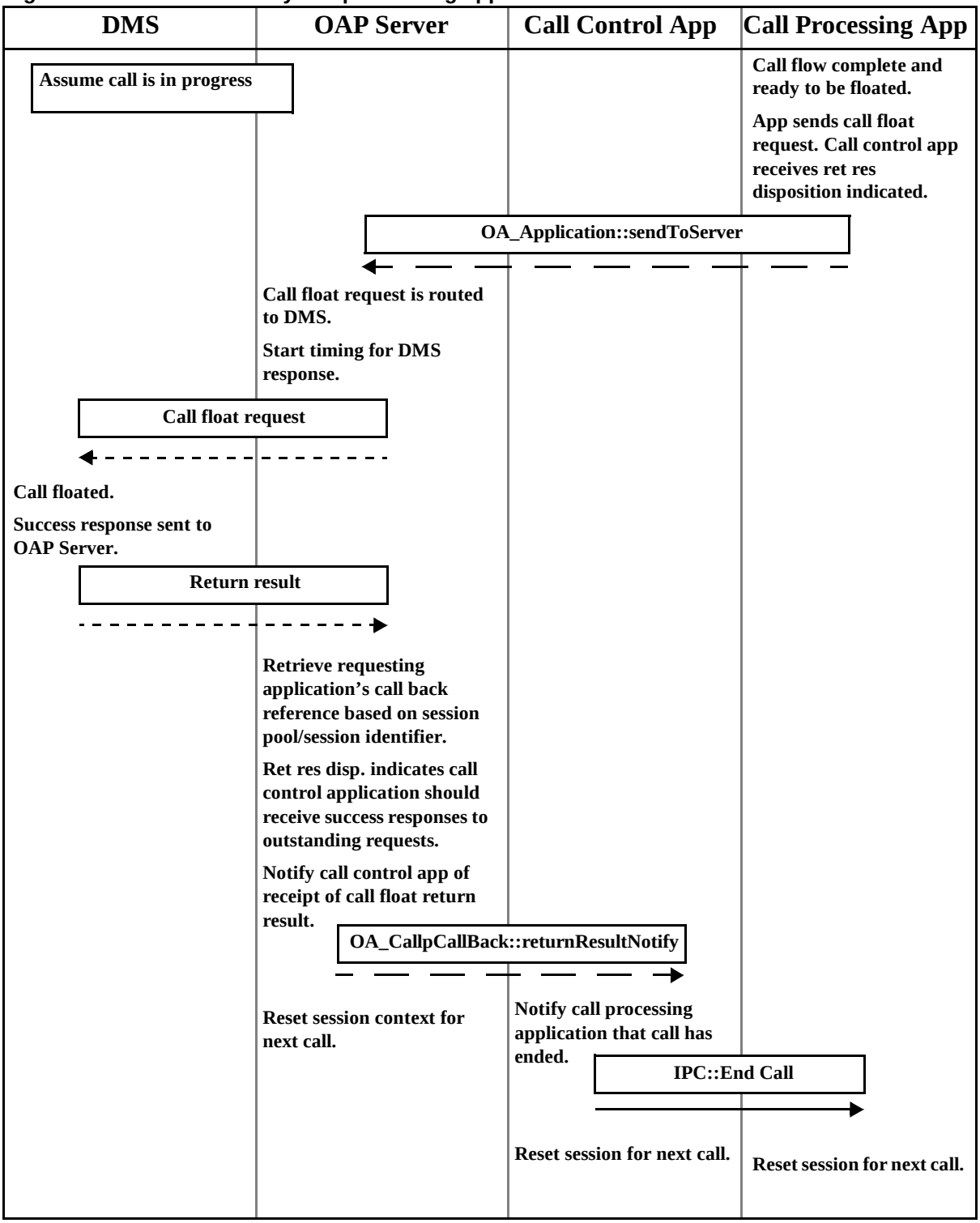
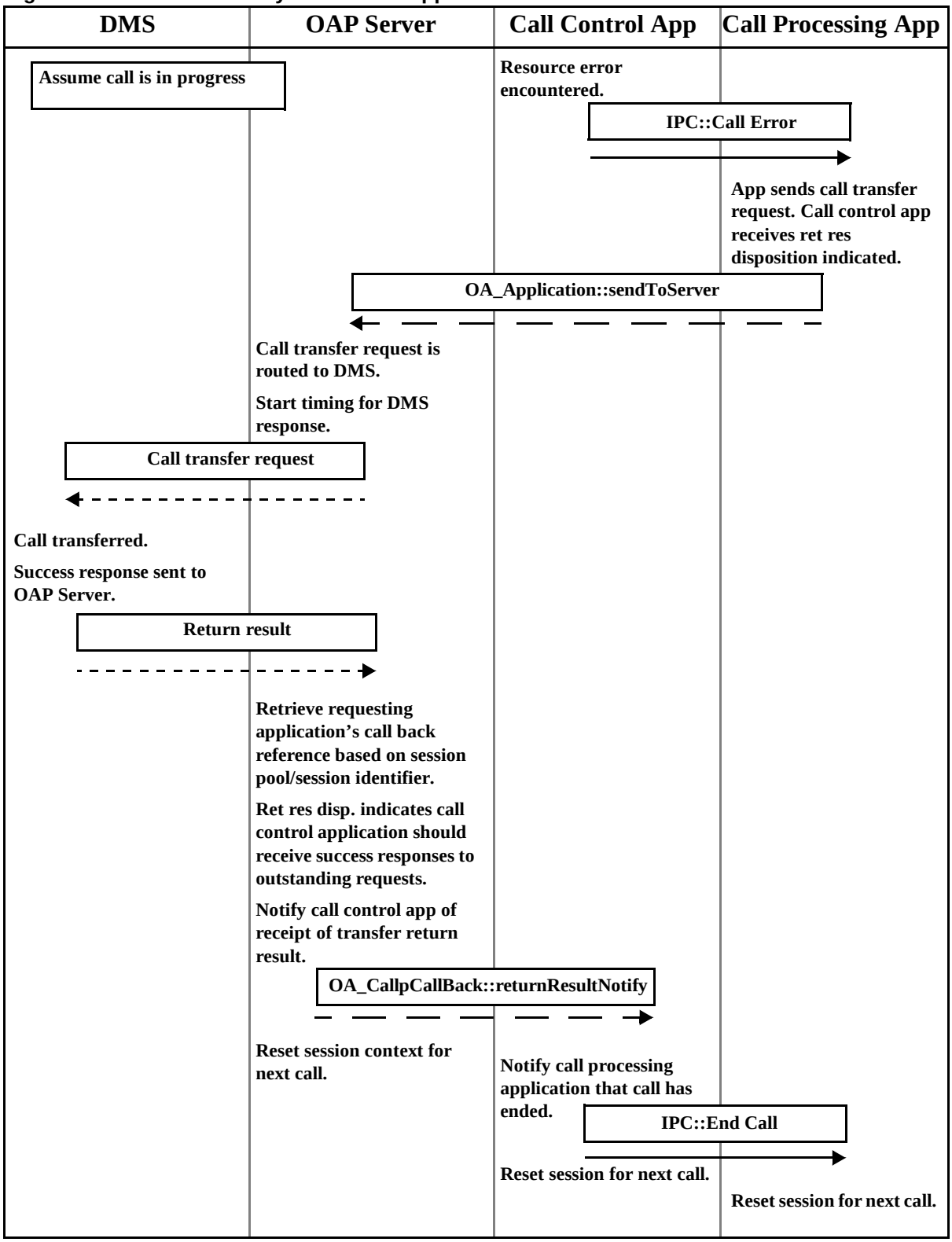


Figure 186 Call released by call control application



OAP Protocol Versioning

The OAP protocol supports protocol versioning to facilitate software upgrades of OSSAIN switches and service nodes. Protocol version negotiation allows two systems (DMS and SN) to communicate with each other using an optimal version of the OAP protocol. Each OSSAIN DMS supports up to four versions of the OAP protocol (current to current - 3). This allows service nodes at the same OAP version level or nodes with up to three earlier versions of the OAP protocol to inter-work with the DMS.

Subscriber originated calls

In an OSSAIN centralized network (OSAC), different DMS switches may be running at many different versions of the OAP protocol. The implication of this to service nodes that are accessed by multiple switches is that subscriber originated calls can be at different protocol versions with each new call.

The OAP Server is responsible for negotiating the optimal OAP protocol version with the OSSAIN host switch. When subscriber originated calls are routed to the OAP server, the OAP server verifies that the protocol version being used for the call is the optimal protocol for the DMS/node interaction. The protocol version is stored in the OAP Server call context for later use.

When the OAP Server notifies the call control application of a call arrival, the version of the OAP protocol being used for the call is provided to the application. This is the version of the OAP protocol that should be used for all OAP messages processed during the call and should be shared with any call processing application involved in the call.

When a call processing application sends an OAP message to the OAP Server to be sent to the DMS, the version of the OAP message is checked against the protocol version stored by the OAP Server in the server's call context. If the versions do not match, the OAP server invokes the processError() method of the requesting application's call back interface to inform the application of the protocol version error.

Service node originated calls

As stated above, DMS switches in an OSAC network may be operating at different OAP protocol versions. Applications originating calls to remote OSAC switches should do so at the optimum OAP protocol version. This can be determined by registering with the OAP Server to monitor session pool resources. By doing so, the application is notified of and session pool state inform messages received by the OAP Server.

When the OAP Server receives a session pool state inform message, it notifies all registered applications using the osacPoolStateInform() method. This method provides the optimal OAP protocol to be used for the OSAC switch sending the session pool state inform message. The application should store the optimal OAP protocol version for each OSAC remote during execution of the osacPoolStateInform() method. In this way, the application has ready access to the optimal OAP protocol version for each switch in the OSAC network.

Corba processing

The OAP Server uses Iona Orbix 2.3c as its source for Corba software. Some Orbix specific implementations, e.g. use of opaque class, were utilized in order to improve real time performance of OAP Server/client application interactions. This section describes how client applications are expected to utilize the Orbix software to interact with the OAP Server.

Starting the OAP Server

To run the OAP Server, you must

- Run the Orbix daemon process, orbixd, at the server host.
- Register the OAP Server in the Orbix Implementation Repository.
- Execute the OAP Server.

Running the Orbix daemon process

Run the Orbix daemon on the server host as follows:

```
% orbixd &
```

Registering the OAP Server

The Implementation Repository is the component of Orbix that maintains information about servers available in the system. Before running the OAP Server, it must first be registered with the Implementation Repository. This is done by running the putit command as follows:

```
% putit OAPServer -persistent
```

In this form, the putit command registers the OAPServer as a persistent server. The putit command only needs to be executed once when running the OAP Server in the persistent mode.

Execute the OAP Server

The OAP Server can be started from a NT command line.

```
% oapserver <config.dat>
```

If a configuration file name is not specified, the default file name oapserver.dat will be used.

Binding to the OAP Server

Client applications must bind to the OAP Server interfaces before they can be utilized. This is done be declaring an Orbix smart pointer for each of the desired interfaces and then binding to the OAP Server for the interface. Figure 187 “Binding to OAP Server interfaces' shows an example of how to bind to the OAP Server Corba interfaces.

Figure 187 Binding to OAP Server interfaces

```
#include “corba/oapserver.hh”

OA_CallControl_var oapServerCC;
OA_Application_var oapServerAPP;
OA_Maintenance_var oapServerMTC;

oapServerCC = OA_CallControl::_bind(“:OAPServer”, host);
oapServerAPP = OA_Application::_bind(“OAPServer”, host);
oapServerMTC = OA_Maintenance::_bind(“:OAPServer”, host);
```

Application call back objects

Many of the OAP Server interface methods require that the application provide a reference to a call back object. Call back objects are used by the OAP Server to send OAP messages and maintenance events to applications.

Call back class definitions

There are two types of call back objects utilized by the OAP Server, call processing and maintenance. Applications that utilize the OAP Server Call Control and Application interfaces must implement a call processing call back interface. Figure 188 “Callp call back implementation class header' show an example the class header for a minimal implementation of the call processing call back interface.

Applications that utilize the OAP Server maintenance interface must provide a maintenance call back interface. An example of a minimal maintenance call back interface implementation is shown in Figure 189 “Maintenance call back implementation class header'.

Figure 188 Callp call back implementation class header

```
#include "corba/oapclient.hh"
#include "corba/oapserver.hh"

class My_CallpCallBackIF  : public virtual OA_CallpCallBackBOAImpl
{
public:

  My_CallpCallBackIF();
  ~My_CallpCallBackIF();

  void sendToClient (CORBA::UShort nodeId, CORBA::UShort poolId,
                CORBA::UShort sessionId, CORBA::UShort oapVersion,
                CORBA::UShort operationId, CORBA::ULong transactionId,
                const OA_OapBuffer* buffer, CORBA::UShort bufferSize,
                CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv());

  void informNotify (CORBA::UShort nodeId,CORBA::UShort poolId,
                CORBA::UShort sessionId, CORBA::UShort oapVersion,
                CORBA::ULong invokeId, CORBA::UShort operationId,
                CORBA::UShort functionId, CORBA::ULong callId,
                CORBA::UShort channelId, CORBA::UShort trunkGroupId,
                CORBA::UShort trunkMemberId,
                CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void returnResultNotify (CORBA::UShort nodeId, CORBA::UShort poolId,
                     CORBA::UShort sessionId, CORBA::UShort oapVersion,
                     CORBA::ULong transactionId, CORBA::UShort operationId,
                     CORBA::ULong callId, CORBA::UShort channelId,
                     CORBA::UShort trunkGroupId, CORBA::UShort trunkMemberId,
                      CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void returnErrorNotify (CORBA::UShort nodeId, CORBA::UShort poolId,
                    CORBA::UShort sessionId, CORBA::UShort oapVersion,
                    CORBA::ULong transactionId, CORBA::UShort operationId,
                    CORBA::ULong callId, CORBA::UShort errorId,
                    CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void returnRejectNotify (CORBA::UShort nodeId, CORBA::UShort poolId,
                     CORBA::UShort sessionId, CORBA::UShort oapVersion,
                     CORBA::ULong transactionId, CORBA::UShort operationId,
                     CORBA::ULong callId, CORBA::UShort problemType,
                     CORBA::ULong rejectReason,
                      CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void timeOut (CORBA::UShort nodeId, CORBA::UShort poolId,
             CORBA::UShort sessionId, CORBA::ULong transactionId,
             CORBA::UShort operationId, CORBA::ULong callId,
             CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void processError (CORBA::UShort nodeId,
                CORBA::UShort poolId, CORBA::UShort sessionId,
                CORBA::ULong transactionId, CORBA::UShort operationId,
                CORBA::ULong callId, CallpErrorCodes errorCode,
                CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

  void restart (CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());
  void shutdown (CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

};
```

Figure 189 Maintenance call back implementation class header

```
#include "corba/oapmaint.hh"
#include "corba/oapserver.hh"

class My_MaintCallBackIF : public virtual OA_MaintCallBackBOAImpl
{

public:
{
   My_MaintCallBackIF();
   ~My_MaintCallBackIF();

   CORBA::Boolean nodeStatusNotify (NodeState oldState, NodeState newState,
                           CORBA::Environment &IT_env=
                           CORBA::IT_chooseDefaultEnv ());

   CORBA::Boolean poolStatusNotify (CORBA::UShort poolId, PoolState oldState,
                           PoolState newState, CORBA::UShort oapVersion,
                           CORBA::Environment &IT_env=
                           CORBA::IT_chooseDefaultEnv ());

   CORBA::Boolean osacPoolStateInform (CORBA::UShort switchId, CORBA::UShort poolId,
                              PoolState state, CORBA::UShort oapVersion,
                              CORBA::Environment &IT_env=
                              CORBA::IT_chooseDefaultEnv ());

   CORBA::Boolean poolTestRequest (CORBA::UShort poolId, char *& mtcText,
                           CORBA::Environment &IT_env=
                           CORBA::IT_chooseDefaultEnv ());

   CORBA::Boolean poolAuditRequest (CORBA::UShort poolId, char *& mtcText,
                           CORBA::Environment &IT_env=
                           CORBA::IT_chooseDefaultEnv ());

   void restart (CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());

   void shutdown (CORBA::Environment &IT_env=CORBA::IT_chooseDefaultEnv ());
};
```

Processing Corba events

Client applications must have a Corba event loop to process method calls on their call back interface. The Orbix method processNextEvent() is invoked by the client application to handle a single incoming Corba event. If an incoming event is available for processing, the Corba software invokes the appropriate application method to handle the event. If no event is available, the processNextEvent() method will timeout and return.

In a multi-threaded application, a dedicated thread can be started to handle the Corba events for the application. The processNextEvent() method can be continually called from the dedicated thread handling Corba events as they occur. Refer to Figure 190 “Dedicated corba thread example'. In this example, the call to processNextEvent() will block for up to 100 ms waiting for a Corba event. The processNectEvent() method is being invoked from within a while loop that never ends. The idea being that the thread in which this method runs is terminated by a main thread when the application is terminated.

Figure 190 Dedicated corba thread example

```
void
My_App::processCorbaEvents()
{
  while ( 1 == 1 )
  {
    try {
      CORBA::Orbix.processNextEvent(100L);
    }
    catch ( CORBA::SystemException &sysEx ){
      cerr << "System exception: " << &sysEx << endl;
      throw;
    }
  }
}
```

List of terms

AABS

Automated Alternate Billing Service

AMA

automatic message accounting

API

application programmer's interface. *See* OSSAIN API.

Automated Alternate Billing Service (AABS)

A DMS TOPS feature that allows automated call completion of a calling card, collect, and third-number billed calls.

automatic message accounting (AMA)

An automatic recording system that documents all the necessary billing data of subscriber-dialed long distance calls.

base layer

The layer of the OSSAIN API that provides low-level utilities for communication and data configuration, and the building blocks used by the other layers.

branding

A feature that allows operating companies the option to connect customer-definable announcements to calls before placing them in a queue or connecting them to an available operator or automated operator system.

call code

A call type descriptor used in AMA recording. The call code defines the type of call or statistic being recorded.

call context block

A generic block of data that contains additional information about a call or the parties involved in a call. It is passed by an SN or operator terminal to the DMS switch.

call floated trigger processing

An OSSAIN capability that allows a call to recall back to an SN after the call has been floated. Floated calls can transfer to a function or a control list.

call type for queuing (CT4Q)

In TOPS and OSSAIN, a method of characterizing an incoming call based on certain criteria, so that the call can be assigned a queue to receive service.

centralized OSSAIN (OSAC)

A DMS architecture in which call processing control is distributed among more than one switch and several *centralized* SNs. OSAC allows the services offered by a centralized SN to be shared by multiple switches.

centralized service node (SN)

An SN in a centralized OSSAIN (OSAC) configuration. Services provided by a centralized SN can be shared by multiple switches. A centralized SN has the node type of either OSNM or OSN. *See also* OSN and OSNM.

CM

computing module

computing module (CM)

The processor and memory of the dual-plane combined core used by the DMS SuperNode. Each CM consists of a pair of CPUs with associated memory that operate in a synchronous matched mode on two separate planes. Only one plane is active; it maintains overall control of the system while the other plane is on standby.

control list

The mechanism used to interwork OSSAIN with existing TOPS functionality. The control list contains the name of a function, such as branding, that OSSAIN applies to the call.

conversation

A sequence of OAP messages between the SN and switch.

CT4Q

call type for queuing

DAS

directory assistance system

data block

A logical grouping of data fields in the OAP message format. Each protocol operation or response can have zero or more data blocks.

directory assistance system (DAS)

A system that provides directory assistance information and call intercept service.

directory number (DN)

The full complement of digits required to designate a subscriber's station within one numbering plan area (NPA)—usually a three-digit central office (CO) code followed by a four-digit station number.

direct transfer

In OSSAIN processing, a type of transition that allows a call to transfer directly from an SN or an operator to a function provider.

DN

directory number

EIU

Ethernet interface unit

error response

An OAP message that indicates the operation request was attempted, but failed.

Ethernet interface unit (EIU)

The unit that connects the DMS SuperNode to the local area network.

failed message threshold

The maximum number of failed messages originating from an SN that, once exceeded, determines the switch should take the node out of service.

Fiber Link Interface Shelf (FLIS)

The DMS SuperNode component that connects the computing module (CM) with the message switch (MS) using a DS512 fiber link. In the OSSAIN network, the Ethernet interface units (EIU) can be provisioned on a FLIS.

file transfer protocol (FTP)

A protocol used to transfer files, such as load files and patch files, across the Ethernet local area network (LAN) facility.

FLIS

Fiber Link Interface Shelf

FTP

file transfer protocol

function

A service or portion of a service that is provided by an SN, an operator, or an existing TOPS automated system. Examples of functions are branding and alternate billing.

function provider

In OSSAIN processing, an SN, an operator, or an existing TOPS automated system.

inform message

An OAP message that updates the message receiver of a call or SN maintenance characteristic. Inform messages also request an operation that does not require a response.

Internet addressing

Physical or subnet addressing used by the Internet Protocol (IP) in which each host is assigned a unique integer address, written in the form of decimal notation. The address is referred to as IP address.

Internet Protocol (IP)

A protocol used at the network layer in data communication across the Ethernet local area network (LAN).

invoke

A request or inform operation in the OAP.

IP

Internet Protocol

LAN

local area network

LIDB

line information database

line information database (LIDB)

A database used to query alternate billed intra-LATA calls. The LIDB relays to the DMS switch information regarding billing number verification for a given dialing number (for example, the collect bill-to-third calls that are always refused and the collect bill-to-third calls that are always accepted).

link interface unit (LIU)

A peripheral module (PM) that processes messages entering and leaving a link peripheral processor (LPP) through an individual signaling data link.

link peripheral processor (LPP)

The DMS SuperNode equipment frame for DMS STP that contains two types of peripheral modules (PM): a link interface module (LIM) and a link interface unit (LIU). In the OSSAIN network, the Ethernet interface units (EIU) can be provisioned on an LPP.

LIU

link interface unit

local area network (LAN)

A network that permits the interconnection and intercommunication of a group of computers. In the OSSAIN network, the DMS switch uses an Ethernet LAN to exchange call control and maintenance messages with service nodes (SN).

LPP

link peripheral processor

maintenance and administration position (MAP)

A group of components that provides a user interface between operating company personnel and the DMS-100 Family of switches. The interface consists of a video display unit and keyboard, a voice communications module, test facilities, and special furniture.

MAP

maintenance and administration position

MAU

media access unit

media access unit (MAU)

The local area network circuitry required by Ethernet interface units (EIU) to connect to the hubs in the LAN-bay.

NI

node infrastructure layer

node audit

A message sent by the switch to an SN when the switch has not received a message from the SN within a pre-defined period.

node infrastructure (NI) layer

The layer of the OSSAIN API that provides well-defined objects that correspond to the node, session pool, and session components in OSSAIN software. This layer also provides the structure and flow of control of API applications.

Northern Telecom publication (NTP)

A document that contains descriptive information about Northern Telecom (Nortel) hardware or software modules and performance-oriented practice for installing, testing, or maintaining the system. The document is often supplied as part of the standard documentation package provided to an operating company.

NTP

Northern Telecom publication

OAP

Open Automated Protocol

OM

operational measurements

Open Automated Protocol (OAP)

The protocol required to communicate data between a DMS switch and an external OSSAIN service node (SN).

Open Position Protocol (OPP)

The protocol required to communicate data between a DMS switch and an OPP-compatible terminal, such as the TOPS IWS.

operation

An OAP action request or inform message.

operational measurements (OM)

The hardware and software resource of the DMS-100 Family switches that control the collection and display of measurements taken on an operating system. The OM subsystem organizes the measurement data and manages its transfer to displays and records. The OM data is used for maintenance, traffic, accounting, and provisioning decisions.

operation class

In the OSSAIN Application Programmer's Interface (API), a class that consists of a class header, and operation header, and zero or more data blocks.

Operator Services Node (OSN)

A centralized SN datafilled at the OSAC remote switch. The OSN node is associated with the OSNM node at the OSAC host switch. The OSN node type is not used in a standalone OSSAIN network.

Operator Services Node Maintained (OSNM)

An SN that has a maintenance relationship with the DMS switch. The OSNM node type is used for SNs in the standalone OSSAIN network and for centralized SNs in the OSAC network.

Operator Services System Advanced Intelligent Network (OSSAIN)

A generic switch-to-service node (SN) interface that allows an SN to control switch functionality. There are two basic OSSAIN network configurations: standalone OSSAIN and centralized OSSAIN (OSAC).

OPP

Open Position Protocol

OSAC

centralized OSSAIN

OSN

OSSAIN service node

OSNM

Operator Services Node Maintained

OSSAIN

Operator Services System Advanced Intelligent Network

OSSAIN Application Programmer's Interface (API)

A set of C++ interfaces to objects that model components in the DMS switch domain. Developers can use the OSSAIN API to build an application to communicate with and process calls involving a DMS switch running OSSAIN software.

OSSAIN preprocessing

A limited session with an OSSAIN SN provided to a TOPS call prior to connecting to an operator or automated system.

pass-through message

In an OSSAIN simultaneous connection, pass-through messages are blocks of data sent in the OPP interface between the switch and an OPP-compatible position; or sent in the OAP interface between the switch and an SN.

PI

protocol interface layer

PIC

point in call

point in call (PIC)

A call state consisting of null, initial call setup, call control, and call floated.

positive assertion

A way that the DMS switch handles receiving a call on an active session (SN session or host-remote session). In positive assertion, the switch takes down the first call and begins a new call on the same session.

protocol interface (PI) layer

The layer of the OSSAIN API that provides the high-level interface to the OAP. The API uses the PI layer to encode, decode, and validate OAP messages.

reject message

An OAP message that indicates a protocol violation with a message or the message could not be parsed.

request message

An OAP message that requests a specific operation to be performed by the message receiver.

request timeout

A request that expired because its response did not arrive within the allowed time interval.

service node (SN)

An external node that interacts with the switch to provide OSSAIN services.

session

In OSSAIN processing, an agent that serves call queues (equivalent to an operator).

session identifier

A numeric representation of a session. A session identifier is used beginning with the initial message sent by the switch or SN and ending with the last message of the session.

session pool

Entire groups of sessions that service specific call queues. Session pools are associated with either an OSSAIN SN (SN session pools) or with an OSAC node (host-remote session pools).

signaling transfer point (STP)

A node in a common channel signaling 7(CCS7) network that routes messages between nodes. Signaling transfer points transfer messages between incoming and outgoing signaling links but, with the exception of network management information, do not originate or terminate messages.

simultaneous connection

Two function providers that are connected to a single call at the same time. OSSAIN allows an SN to connect to a call simultaneously with another SN or with an operator.

SN

service node

STP

signaling transfer point

state transition

A node change from one maintenance state to another; for example, from system busy to in service.

success response

An OAP message that indicates the requested operation was successfully performed by the receiver of the request.

tag field

A value in the API used to create a set of unique data block identifiers. The tag field suffix distinguishes related data blocks when they are allowed to appear multiple times in an OAP operation.

TDP

trigger detection point

TOPS

Traffic Operator Position System

TOPS IWS

Traffic Operator Position System Intelligent Workstation System

Traffic Operator Position System (TOPS)

A call processing system made up of a number of operator positions. Each operator position consists of a visual display unit, a controller, a keyboard, and a headset.

Traffic Operator Position System Intelligent Workstation System (TOPS IWS)

An integrated operator assistance, intercept, and DA position, which uses a personal computer with customized software, keyboard, and interface.

transition

An activity that bypasses control list processing to allow for greater control of the OSSAIN call by the SN.

trigger

An event that causes an OSSAIN call to be redirected to an SN or operator.

trigger detection point (TDP)

The point in call processing when the switch determines the action associated with a trigger. Two possible actions are associated with a TDP: continue with normal call processing (no trigger is hit) or bridge on an SN or operator (trigger is hit).

trigger profile

A mechanism that allows operating companies to control trigger processing on an individual call basis.

UDP

User Datagram Protocol

User Datagram Protocol (UDP)

A member of the Internet protocol suite of protocols, UDP is used at the transport layer for OSSAIN data messaging.

WAN

wide area network

wide area network (WAN)

A large-scale, high-speed communications network used primarily for interconnecting local area networks (LAN) located in different cities or nations.

DMS-100 Family

OSSAIN API

User's Guide

Publication number: Q260-1
Product release: Release 7.0
Document release: DRAFT 03.01
Date: April 2000
Printed in the United States of America