X/Open Preliminary Specification

Systems Management: Event Management Service

Draft V0.2 (12/22/95)

X/Open Company, Ltd.

Table of Contents

Preface

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information systems suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

....

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focused on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

....

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

....

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

....

This Document

This document defines requirements for a programming interface that receives notifications in the form of events and, reliably, transports the events to applications.....

The structure of the document is as follows:

Chapter 1, Introduction - Defines the objectives of this specification and defines terminology used throughout the document.

Chapter 2, Overview of the Event Management Service - Begins with a description o f the event management service general model. This is followed with an architectural view of XEMS. This is followed by a description of the role of XEMS in the systems management domain. The chapter continues with a survey of usage scenarios and deployment strategies. The introduction of the X/Open Event Management Service (XEMS) data formats follows the survey. The chapter concludes with an overview of the consumer, producer, and administration interfaces.

Chapter 3, Data Formats - Describes the data formats used by XEMS.

Chapter 4, Administration Interface - Describes the application programming interface (API) and the administration commands used to configure and control an XEMS implementation.

Chapter 5, Consumer Interface - Describes the API and commands available to register with and receive events from an XEMS.

Chapter 6, Producer Interface - Describes the API and commands available to register with and send events to an XEMS.

Appendix A, Reference Implementations - Is divided into several parts, one for each reference implementation. Each part describes a specification implementation of the XEMS. The XEMS interfaces are source compatible across all implementations. The reference implementations are CORBA, DCE, and UMA.

Appendix B, Event Objects - Describes a minimum set of (standard) event objects. These mappings are required for all conforming implementations.

Appendix C, xems.h - Is the header for XEMS.

Trademarks

Referenced Documents

X/Open CAE Specification: The Common Object Request Broker: Architecture and Specification. X/Open Document Number: C432. August 1994. ISBN: 1-85912-044-X. This document is equivalent to OMG Document Number 93-12-43.

X/Open Preliminary Specification: Common Object Services Specification, Volume 1. X/Open Document Number: P432. June 1994. ISBN: 1-85912-0248-2. This document is equivalent to OMG Document Number 94-1-1.

X/Open CAE Specification: X/Open DCE: Directory Services. X/Open Document Number: C312. December 1994. ISBN: 1-85912-078-4.

IBM: Open Software Foundation: Request for Comments, DCE Event Management Service. Draft 3. OSF RFC: 67.0. March 1995.

X/Open CAE Specification: X/Open DCE: Remote Procedure Call. X/Open Document Number: C309. August 1994. ISBN: 1-85912-041-5.

X/Open Distributed Security Framework. August 1994.

X/Open Preliminary Specification: Systems Management: GDMO to XOM Translation Algorithm. X/Open Document Number: P319. March 1994. ISBN: 1-85912-023-7.

X/Open Preliminary Specification: Generic Security Service API (GSS-API) Base. X/Open Document Number: P308. January 1994. ISBN: 1-85912-025-3.

X/Open Snapshot: Generic Security Service API (GSS-API) Security Attribute and Delegation Extensions. X/Open Document Number: S307. January 1994. ISBN: 1-85912-026-1.

X/Open Guide: Internationalisation Guide, Version 2. X/Open Document Number: G304. July 1993. ISBN: 1-85912-002-4.

ISO/IEC 10165-2: 1992(E) Information technology — Open Systems Interconnnect — Structure of management information: Definition of management information.

ISO/IEC 10165-4: 1992(E) Information technology — Open Systems Interconnnect — Structure of management information — Part 4: Guidelines for the definition of managed objects.

ISO/IEC 10165-5: 1994(E) Information technology — Open Systems Interconnnect — Structure of management information: Generic management information.

ISO/IEC 10165-6: 1994(E) Information technology — Open Systems Interconnnect — Structure of management information: Requirements and guidelines for implementation conformance statement proformas associated with OSI management.

ISO/IEC 10165-7: 1992(E) Information technology — Open Systems Interconnnect — Structure of management information — Part 7: General Relationship Model.

ISO/IEC 10646-1: 1993(E) Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

OMG White Paper on Security. OMG Security Working Group. Issue: 1.0 April 1994.

X/Open CAE Specification: API to Directory Services (XDS), Issue 2. X/Open Document Number: C317. March 1994. ISBN: 1-85912-007-5.

X/Open CAE Specification: Systems Management: Management Protocols API (XMP). X/Open Document Number: C306. March 1994. ISBN: 1-85912-027-X.

X/Open CAE Specification: OSI-Abstract-Data Manipulation API (XOM), Issue 2. X/Open Document Number: C315. February 1994. ISBN: 1-85912-008-3.

CHAPTER 1

Introduction

1.1 Purpose

Ever increasing critical and complex systems will only be cost controllable if many of today's systems management and administration activities can be automated. This is true for the user's service provision organizations, system vendors and ISVs.

A well designed Event Management Service (EMS) will be of increasing importance to organizations as they become increasingly dependent on IT services; it is a fundamental component needed to maintain service availability by:

- giving timely warning of impending problems (for example, file capacity thresholds)
- notifying system administrators of failing processes and system components
- speedily identifying root causes of problems in ever more complex systems
- automatically fixing problems before service levels are degraded
- integrating application-specific events mechanisms so cross-application correlation can be done at a higher level (for example, network outages are the root cause of many application errors, and the administrator needs his attention drawn to the root cause)

- providing a specification which facilitates interoperability of multi-vendor event management systems.

Organizations are (increasingly) prepared to pay vast sums to duplicate system components (processors, disks, network connections) to maintain systems reliability. As system complexity increases, a largely automated EMS is a necessity for maintaining service availability at a reasonable cost, given systems created by integrating diverse components from an increasing number of suppliers (systems vendors, ISVs, in-house IT developers).

A set of event standards with which these components can inter-communicate events of interest is a fundamental need (that is, inter-connection is not enough, inter-comprehension is required).

1.2 Background

Automated detection and response to events is required in order to effectively manage and monitor distributed systems. Today's trouble ticketing systems are not sufficient, primarily because they are reactive systems, where a proactive system is required.

Examples of events include such things as program termination, node available, node down, administrator-defined traps, and so on. Provided independent vendors use the same EMS infrastructure, those same vendors can send and correlate events among each other's system management products.

The value of an EMS can only partially be measured by technical merit, and that value is leveraged many times over by the number of independent vendors that can and will utilize the same standard mechanism; therefore the value of this type of standard increases exponentially with the number of vendors in compliance with the standard.

1.3 Scope

This specification addresses event management services for systems administration purposes. However, like SNMP, this technology may well be applicable outside its original charter (for example, management of customer applications).

It is not the function of the EMS to generate events. The underlying managed objects (or some proxy) must raise events as appropriate. The EMS, however, must process all these events in a real-time fashion, administer their definition, enabling, and so on.

Likewise, it is not the function of EMS to provide high-resolution interval data. For example, EMS is not designed to be the feed for a performance meter for CPU utilization on 5 second intervals. The design space for a performance meter would preclude the use of a persistent cache (required for EMS reliability), e.g., one would not expect a performance meter to provide real-time and historical data.

1.3.1 Requirements

Event Notification API

An API is needed to specify how events are delivered to applications. This API must include the attributes described in Event Construction below.

Event Subscription API

An event subscription API is needed for applications to tell the system which events are of interest, which should be forwarded, and where. For example, a database expert might subscribe to all database-specific events, as well as some network-specific events.

When subscribing to events it must be possible to designate all instances of an event (for example, all table drop actions), all events of a sub-category (for example, all DDL actions) or all events of a category (for example, all user actions). It must be possible to qualify the events of interest (at any level) by Boolean expression of attribute (for example, events in a particular management domain and time stamps between 9am and 5pm, or all events in a geographic region of a certain priority).

Event Construction

There are a set of attributes that are common to all events that must always be populated (default attributes). These include:

- event identification, category and subcategory
- date/time/timezone of origin
- originating process (physical ID)
- component, subcomponent, module, subroutine, source code line, and so on, identifiers
- priority and severity code
- text message
- end-user device identifier.

In addition, the above classifications must be extensible to support specific applications and logical extensions of this technology without breaking compatibility in the way the event service and applications handle the default attributes.

Global Name Service Requirement

It is expected that EMS will be most useful in a large networked environment where multiple-sourced applications may subscribe to events from each other in order to coordinate their activity. The event service must participate in a global naming standard such that separately developed applications do not have name conflicts when these applications meet in the marketplace at the user's desktop.

The demand here is for a standard naming or numbering system which can:

- cope with the categories we know about today
- be expansive enough to last for a very considerable time to cope with new technologies (that is, regularly updated)
- allow users to define their own events without fear of creating chaos
- be capable of being both process and human interpretable
- be well structured and sustainable.

The name service selection should be done in concert with the networking group to minimize the need for name service gateways. The RTG requires that only one name service be adopted for event services, rather than have event services able to participate in multiple name services directly; this approach should eliminate need for event service naming gateways, enabling event coordination among separately-developed ISV and customer software.

Centralized Event Management

Key to successful event management is the ability to control and view event services from a single centralized point. The concept of a centralized event management service is purely logical. It may make sense from the performance point of view to implement overlapping distributed event handling processes that minimize network traffic and maximize availability. However, to the user, the EMS must be able to look like a centralized service.

At the same time, there is a need for an event routing system that allows particular categories/classes of event to be ultimately routed to a particular (configurable) process, user or desk. For example, database events to database administrators, network events to a network specialist, process failures to watchdog task, and so on.

It will probably be appropriate for there to be multiple routes to access event management information from a UI perspective; the issue of centralization is that once you are in position to access some event management information, you will actually be able to access all such information (to which you have authorized access).

Defining and Designating Events

There is a need to allow for arbitrary user-defined events; the event services mechanism must be user-extensible.

Some events will be pre-defined in the management applications, and some events are arbitrary in nature, and defined according to specific needs of a customer. An example of a pre-defined event is notification of a server shutdown or a communications failure. Management applications will have default behaviors for responding to pre-defined events. Statistics-based events (as described in Managed Server Performance Events below) are examples of customer-defined events.

Categories of Events

This sub-section describes the types of events that the EMS must handle. All these events should be handled symmetrically (that is, services available to handle any particular type of event should be equally applicable to all types of event).

Events can be grouped into categories and sub-categories. For example, one category of event may be User Action Events. Within that there may be a sub-category, Data Definition Language (DDL) Action Events (user changes a schema object).

One particular DDL Action Event might ``Column Added to Table''. All such instances of this event have exactly the same attributes: server, database, table name, column name, and so on.

The category hierarchy represents a taxonomy without rationale. Another notion would be to permit the categories to exist across management spaces, e.g., CMIP, RDBMS, Oracle, et al. At least one could treat categories and subcategories as object names, with a flat identity space, within a user-definable hierarchy. This mimics the way products, e.g., Microsoft Office for Windows/95, are installed. To wit, the user provides the root of the hierarchy and the install process carries out the rest. There is no confusion because of this, since the registry uses a unique identifier for the product, i.e., a flat identity space.

The following is an example of the category hierarchy that the event services must be able to handle:

1. Managed Server Status Events

   Notification of change in status of managed database server (for example, starts, becomes suspect, dies normally, dies abnormally).

2. Managed Server Performance Events

   - Notification of threshold breach by point statistic (for example, number of locks taken exceeds threshold, cache hit ratio falls below threshold, age of oldest lock wait exceeds threshold).
   - Notification of threshold breach by rolling statistic (for example, number of users exceeds certain threshold for particular proportion of recent prescribed time period, logical to physical read ratio below particular threshold for recent prescribed time period, rolling average breaches threshold).

   Note that these statistics may be at any level of aggregation (for example, by server, by object, by user, by group of servers).

3. User Action Events

   This refers to user actions that take place on managed servers.

   - Notification of connection/disconnection of user.
   - Notification of DDL execution.
   - Notification of change of user password.
   - Start-up or shut-down of a server.
   - Creation of a new database device.

- Notification of elapsed time or resource usage threshold breach for execution of user action.

An attribute of all these events should indicate success or failure of the operation. For long running actions (or perhaps all actions) it should be possible to raise events both at commencement and completion of the action.

4. Management Action Events
   - Notification of action started by a user in a management application.
   - Notification of completion of action taken in a management application.
   - Notification of elapsed time or resource usage threshold breach for execution of management action.

   An attribute of all these events should indicate success or failure of the operation. For long running actions (or perhaps all actions) it should be possible to raise events both at commencement and completion of the action.
5. External Signals
   - An E-mail handler should accept formatted text that can raise a user-defined event (see below).
   - UNIX signal(2) received.
6. Self-management Events
   - Management process failure (that is, management software process dies, event raised by these processes watching each other).
   - Subscription action failure due (see Subscription Service below).
   - Event notification messaging threshold breach.
   - Service failure (for example, log consolidation failure due to insufficient disk storage).
7. User-defined Events

   It should be possible to raise user-defined events that can have arbitrary data structures associated with them, so that a customer can use the event mechanism for asynchronous notifications of any sort. This would be particularly useful, for example, for standardized handling of error conditions discovered in the middle of command scripts without having to code the same error handling into multiple scripts.
8. Composite Events

   A mechanism is required for creating a composite event that is raised when a series of other events (be they simple or composite) occur within a given period of time. This is described more fully below.

A single event notification is often only one small part of a larger picture. Only when certain events occur in relative proximity can some sense be made of a situation and some appropriate response be made. Therefore, composite events are key to proper event management. Without a composite event mechanism, systems administrators would end up writing complex scripts to manage event combinations that mirror real situations. For example, a series of performance threshold breaches may indicate a serious problem where each individual breach is merely an interesting event to be watched for future use.

Additional background to the requirements is given below:

- *Architectural Niceties*

  Implementation of requirements in this section must not cause delay in providing the basic event management service; it is more important to get the basic EMS APIs (for subscription/notification) resolved so ISV and customer software can be rewritten to those APIs as soon as possible.

  However, these features are required for the widespread deployment and use of EMS, so the general requirement for this section is that these features must be able to be layered on top of the basic EMS.

- Binding Events to Actions

  An event subscription invokes some action sequence when an event notification is received. For example, when a managed server shuts down unexpectedly you may want an on-screen notification and a beeper to be called. In this case, one would subscribe to the event by associating the action sequence to post the notification and call the beeper with the occurrence of this event.

  It must be possible to associate any action to an event; indeed a generic execute script option is theoretically sufficient to meet all needs. However, there are certain actions for which our applications can provide more friendly support, and these include arbitrary combinations of:

  - email message notification
  - beeper called
  - log file entry posted
  - row inserted into some table in some database
  - managed server stored-procedure execution
  - SNMP alert raised
  - asynchronous desktop visual alert (for example, GUI pop-up)

- visual cue in iconic representations (for example, color change)
- arbitrary program execution
- invoke management services at API level (avoids heavyweight process spawn).

In some cases, subscriptions involve actions on objects that are not active (for example, insert a row in a table where server is down, run a shell script on a node that is not on the network). A mechanism for storing such actions and executing them when possible is required. An event subscription failure event is needed so that such delays can be noticed.

- Navigating the EMS SuperHighway

  There needs to be two basic ways of navigating through the UI to define events. One is starting from a general event management selection that drills down to the particular event you wish to manage, and the second is from the dialog set for managing a particular object by allowing you to select a ``manage events" option.

- Convenience Features/Toggling Event Subscriptions

  As well as creating a subscription, you may want to deactivate it temporarily, and then reactivate it at a later time without having to recall all the details of what action sequence was associated with what designated event(s).

  By placing subscriptions in collections one could activate and deactivate sets of subscriptions together. For example, if one had the need to watch a particular group of resources for threshold events from time to time (say, during heavy business cycles), one could activate the subscriptions for the period of interest and deactivate them for the rest of the time.

- Programmable Event Filters

  The event service should allow for the provision of programmable event filters at suitable nodes in a network to minimize the degree of ``event storms", and to ensure adequate (end-user) service levels. That is, the programmable event filters will identify the most important of a number of events arising at that point in a given (configurable) timezone and will identify the relationship and forward only the single consolidated event.

  In this way ``policies" can be established for both event filters and for action systems built on top of an event system.

  It is understood that programmable filters satisfy a higher level need which may best be layered above the event service; so the requirement here is to ensure programmable event filters can be layered on top of the event service.

- Event Definition Language

A common event definition language (EDL) should be specified. This language would allow the specification of all events which an application or managed object generates. This would allow ISVs to ship a list of events that their product could cause, resulting in minimum user effort in integration of new applicants into the EMS.

1.3.2 Performance

In pursuit of efficiency, events must only be posted when active subscriptions are associated with the event. Whenever an event is activated or deactivated, the EMS must ensure that the posting process (probably a managed server or its proxy) is informed whether or not to send the particular event notification.

A key requirement is that the event services have good performance characteristics, and that they do not degrade performance of other processes on the network. This general events mechanism must be efficient enough for real-time performance monitoring of operating system-level activity.

The event service should provide minimal event ``reduction” in cases where:

- Many duplicate events received in a short timeframe could be reported as a single event with a count.
- Events which represent rates of change can be computed by the event service prior to forwarding over the network.

The infrastructure should provide the ability to monitor event notification traffic, and to create performance threshold events on such data. Unfortunately, it will be relatively easy to configure the event services to create event cascades and event storms. This would be through specifying an action to occur when an event is raised, that itself will cause other event notifications which cause more actions with more events, and so on.

It may, therefore, be appropriate to create an event-action history which attaches to event notifications and to action executions that can be examined for cascade situations. It may be enough merely to warn of this possibility.

In general, it is required that there be minimal propagation of events, to avoid performance degradation. If a store and forward mechanism for event notifications is used, then event aging must be monitored as a key performance indicator.

Minimal performance degradation could be achieved through replicating subscription action scripts to the event handling agents near the corresponding managed nodes being watched. (Such replication also enhances fault tolerance.) This could be enhanced by self-load-balancing where event handling agents start-up and die according to need on arbitrary managed nodes.

It is important, for performance reasons, that the only events posted for distribution are those events with active subscriptions (that is, someone is interested in listening to them); otherwise event notifications may well clog the network.

From a performance standpoint, the transport protocol used by the event notification messages may be important. If the event notifications are not sent on a connectionless protocol (most likely to be performant) then appropriate performance characteristics must be validated.

1.3.3 Reliability

Since event management applications are responsible for sending notification of any system and network problems to responsible operators, the management application itself as well as the underlying EMS must be absolutely reliable.

1.3.4 Standardization and Portability

The EMS provides a generic (i.e., implementation independent) API. This API permits source code compatibility across implementations. The API provides functions for consumers, producers, and administrators.

1.3.5 Extensible

As the technology moves, new managed objects and associated events will be required. In addition, customers and vendors may supply events for their applications. All of this must occur in a seamless manner.

1.3.6 Security

Since the event subscription service API is the window for applications to see generic system-wide activity, applications must be prevented from unauthorized snooping of system behavior at this access point.

Access to event subscription and composite event construction must be secured by the access permissions of the managed objects.

Resolution of this requirement should be coordinated with the X/Open Security TWG and, because EMS will be most useful when applied in an enterprise configuration, coordination with the Networking TWG is also advisable.

1.3.7 Internationalization

The EMS must be I18N compliant.

1.3.8 Interoperability

The event management services from different vendors must inter-operate.

1.4 Terminology

This section establishes definitions for terms used in this specification.

consumers

Consumers process event data [COSSES-32]. In DCE, a server application which registers for, receives, and processes event data.

event

An individual data entity corresponding to some information that needs to be communicated from the managed environment to the management applications is known as an "event".

event channel

An event channel is a service that decouples the communications between suppliers and consumers. An event channel is both a consumer and a supplier of events. In CORBA, event channels are standard objects and communication with an event channel is accomplished using standard CORBA requests.

In CORBA, an event channel can provide asynchronous communication of event data between suppliers and consumers. Although consumers and suppliers communicate with the event channel using standard CORBA requests, the event channel does not need to supply the event data to its consumer at the same time it consumes the data from its supplier [COSSES-41].

event communication

Event communication may be generic or typed. In the generic case all communication is by means of generic push or pull operations that take a single parameter that packages all the event data. In the typed case, communication is via operations defined in IDL. Event data is passed by means of the parameters, which can be defined in any manner desired [COSSES-32].

event data

Event data are the objects communicated between suppliers and consumers [COSSES-32].

event service

An event service decouples the communication between objects. It defines two roles for objects: the supplier role and the consumer role [COSSES-32]. An event service is a system service, which supports the generation, registration, filtering, and forwarding of events to management applications and other management objects.

filter

An Event filter is a mechanism to select specific types of events, e.g., selection by time or type.

manager

A manager is the initiator of an event management interaction.

notification

Event notification is an asynchronous mechanism through which an event is received by a management application, e.g., a CORBA consumer.

proxy consumer

A proxy consumer is similar to a normal consumer (in CORBA, it inherits the interface of a consumer), but includes an additional method for connecting a supplier to the proxy consumer [COSSES-44].

proxy supplier

A proxy supplier is similar to a normal supplier (in CORBA, it inherits the interface of a supplier), but includes an additional method for connecting a consumer to the proxy supplier [COSSES-44].

pull model

An approach to initiating event communication, allowing a consumer of events to request the event data from a supplier. In CORBA, consumers request event data by invoking pull operations on the PullSupplier interface [COSSES-32].

push model

An approach to initiating event communication, allowing a supplier to initiate the transfer of the event data to consumers. In CORBA, suppliers communicate event data by invoking push operations on the PushConsumer interface [COSSES-32].

registration

Registration is a mechanism by which a management application can indicate an interest in receiving notification on specific events. In CORBA, this would be done by a consumer to create a proxy consumer.

suppliers

Suppliers produce event data [COSSES-32]. In DCE, an application which emits event data.

CHAPTER 2

Overview of the Event Management Service

2.1 General Model

Among the common goals of most systems management applications are to ensure the availability and reliability of the managed computing environment, while imposing a minimal amount of overhead onto the environment. To do this effectively, systems management applications have a requirement to be able to produce and consume large volumes of data that corresponds to important, time critical information about the managed environment.

An individual data entity corresponding to some information communicated from the managed environment to the management applications is known as an "event". The EMS described in this document defines the mechanisms necessary to generate and process events, thus enabling systems management applications to respond appropriately to changes in the availability and/or reliability of computing resources in the managed environment.

Today's commercial computing environments are characterized by both their distributed and heterogeneous natures. Typical environments are comprised of networks of a wide variety computer hardware and software produced by many different vendors, cooperating in some fashion to perform mission critical functions on behalf of end-users. As a result, systems management applications are required to manage a wide range of computing resources, and thus need to consume events produced by applications developed by several different vendors.

In order to enable the development of management applications that communicate and process the event information within a heterogeneous, distributed computing environment, the EMS API must be standardized. Developers of system resource monitoring services require a standard API for reporting the occurrence of events that must be processed by management applications, and management applications require a standard API for the consumption of events that may be generated by a wide variety of resource monitoring services.

This document describes both a general model for an EMS architecture, and a set of APIs tailored to mapping the general model to an X/Open compliant environment. While it would be ideal to have a single implementation that is appropriate for all computing environments, in reality different computing environments employ vastly different distribution mechanisms, and a single implementation is simply not feasible.

Appendix A of this specification describes the mapping of the EMS API to several implementations. This is meant to be a sampling of implementations. By agreeing upon the implementations for various spaces, e.g., CORBA event service, implementors of event services and gateways will be able to create interoperable products.

The EMS provides reliable, in sequence, asynchronous notification of events. It is implemented as an intermediary layer between management applications and managed objects.

An event is a partially opaque object emanating from a managed object. The information (object) is partially opaque in so far as there is a standard header and an encoded data stream. The standard header provides the minimal amount of information that an EMS must be capable of filtering for any management application. The standard header also provides the minimum amount of information that an EMS must be capable of filtering for a managed object.

The EMS specification does not prescribe the mechanism used to locate Event Service instances either from a management application or managed object perspective. These relate to the given environment, e.g., DCE.

The EMS specification does not address mechanisms for activating (or otherwise controlling) managed objects. These mechanisms may be incorporated into management applications.

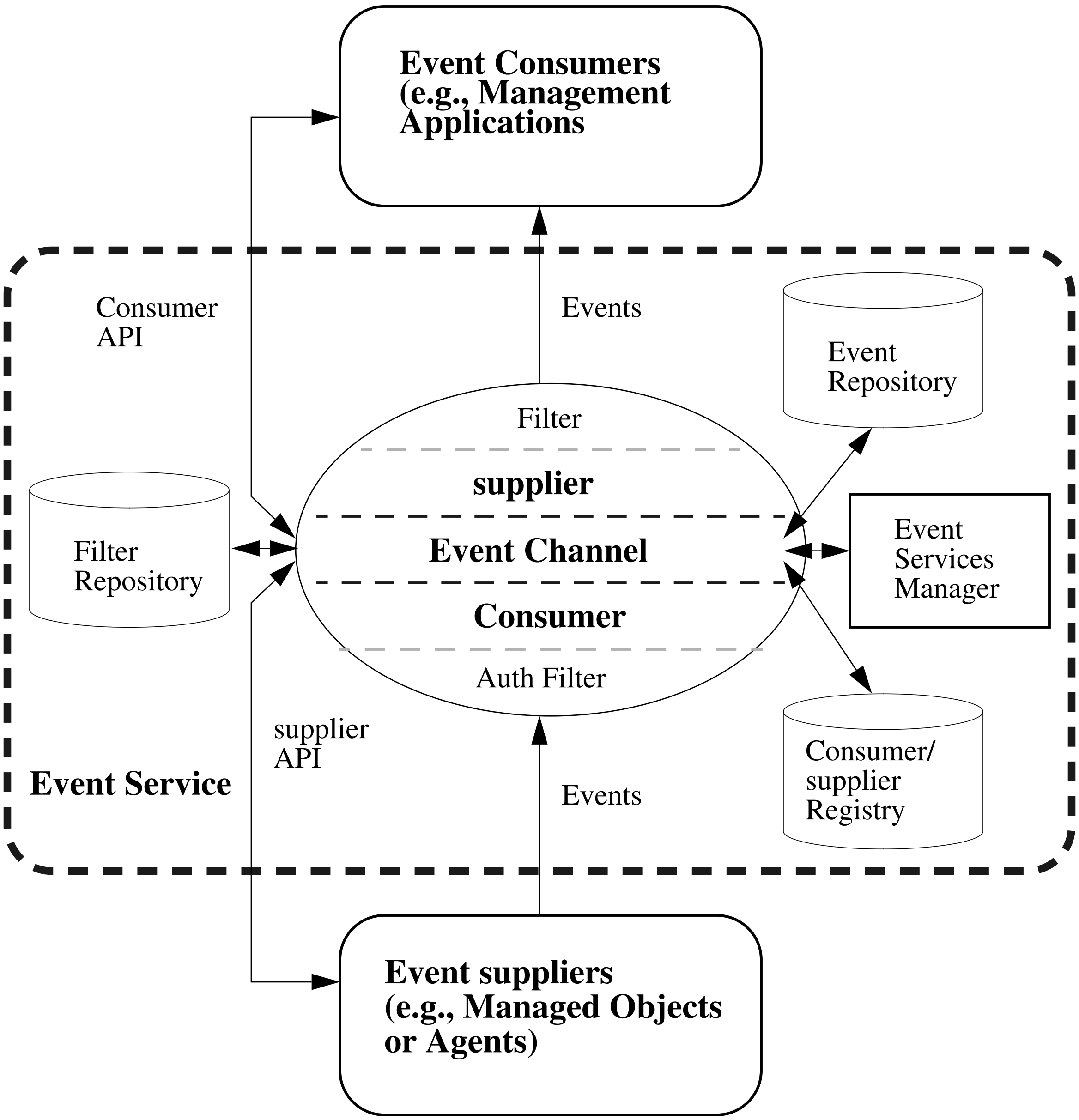
EMS implementations operate within the security policy of the given environment. They do not implement a distinct security framework.

2.1.1 Model

An X/Open EMS conforms to this model:

On the surface the core of the Event Service looks a lot like a COSSES Event Channel. There are, however, some notable differences.

- The EMS involves a pair of channels, centralizing the functionality associated with reliable delivery.
- The Event Channel includes filtering mechanisms. The interface to suppliers ensures that a supplier can only insert events (objects) for which they are authorized. The [consumer-side] filter performs two major functions. First, it delineates the objects in which the consumer is interested; both in terms of the object, *per se*, and in terms of the criteria receiving a given instance of an object. Second, it provides a security mechanism, where the consumer can only receive objects for they are authorized.
- The event repository is designed to hold events awaiting delivery to consumers. This is used in conjunction with the filter repository and the consumer/supplier repository. The event repository is not a permanent log. A consumer may act as a local logging facility; or, a consumer may act as a proxy for a centralized logging facility. In any case, utilizing a consumer for logging simplifies the implementation, configuration, and administration of the Event Service.
- The filter repository contains filters registered by consumers. These are analogous to the *where* clause in the *select* statement for SQL and ODMG queries. In subsequent bindings, consumers specify a set of filters to be used as criteria for event selection. The elements of the set are *anded*. Hence, an event will be sent to the consumer only when it passes all of the criteria in the set.
- The consumer/supplier registry contains state information about active consumers and suppliers. It also contains schema information for typed events.
- The event services manager is responsible for orchestrating the activities of the Event Service.



EMS Conceptual Model

The objectives for the remaining sections of this chapter are to demonstrate how the event services described in this specification would typically be configured, and how developers of software components that produce and/or consume events would integrate with this service.

The usage and deployment section illustrates aspects of the EMS through the deployment and use of an intelligent agent.

2.1.2 Conceptual Flow

This section describes the flow through the EMS. There are several timeframes: installation, authorization, registration, connection, delivery, and maintenance.

Installation of an EMS

The installation of EMS involves loading the package, allocating the repositories, and implementation setup. Subsequently, this EMS must be registered with other EMS instances. A security context must be created for this EMS. The implementation will provide the default set of event schemas. These are stored in the Consumer/supplier registry. The installation process will also instantiate the namespace (registry) for the basic event types.

Installation of a supplier

The installation of a supplier involves loading the package, adding the supplier to the Consumer/supplier registry, updating the namespace for the events provided by the supplier, and establishing the security context for the supplier.

Authorization of a Consumer

A consumer must be registered with the Consumer/supplier repository before the consumer may use the EMS. The registration defines the events that the consumer may receive. This provides the security context for the consumer. Registration may also involve the definition of space constraints for the user. The constraints may encompass the filter repository and the event repository. The consumer may be limited to a fixed number of filter groups or event delivery space.

Establishing Filters

The consumer registers filters with the EMS. A filter consists of a filter group. A filter group is a set of filter lists. A filter list is a set of filter expressions. Each filter expression represents a logical expression. An expression consists of the name of an event attribute, a logical operator, and a value. A set of filter expressions represents a logical anding of the members. So, the logical result of a filter list is true if all of the filter expressions are true. Filter lists are combined to form filter groups. The filter lists are ored within the filter group. Hence, the logical result of a filter group is true if any of the associated filter lists are true.

Interfaces are provided manipulating, querying, and administering filters. Consumers are not required to use filters. The effect of not using a filter group is the reception of all events consumer is authorized to receive.

supplier Connections

A supplier connects to the EMS at supplier start-up. The bind operation includes arguments for the reflecting hints to the supplier. To receive hints a supplier would typically operate as both a client and a server. The server side would respond to the hints, while the client operations are used to connect and disconnect from the EMS as well as to send events to the EMS. A supplier is not required to accept hints. Even if the supplier provides the server interface, it is not required to act on a hint.

The EMS applies an authorization filter to the supplier. A supplier may only insert event types for which it is registered. A supplier may be permitted to insert any event type registered in the EMS. This permits generic suppliers, e.g., gateways among EMS instances.

Consumer Connections

A consumer connects to an EMS to receive events. When the consumer connects, it indicates the recipient of the events. It also indicates the type and quality of service. The type of service is either the push or pull model. The quality of service controls the EMS delivery semantics. This is a hint to the EMS. It may not support quality of service levels. At a minimum, an EMS implementation must support reliable, in sequence delivery.

The normal sequence of operations by a consumer are:

- Use **ems_consumer_start** to establish the consumer session with the EMS. This will establish the principal to receive events, the mechanism for event delivery, and the quality of the delivery.
- Query the EMS to determine what event types are available to the consumer. Based upon the selections made from the query, form filter expressions. Each filter expression provides threshold criteria for an event. Combine the filter expressions into a list.
- Use **ems_filter_add** to add the list to the filter repository.

- Use **ems_add_filter_to_group** to create a consumer filter group. A consumer has a single active filter group at any moment. A filter group may be queried, manipulated, and deleted.
- Use **ems_consumer_register** to associate a filter group with an EMS instance, indicating that the latter is to send events meeting the filter criteria. This method is invoked once for every EMS instance and filter group of interest.
- Registration and filter group manipulation may be inter-twined to establish unique event filtering criteria at each EMS instance.
- An **ems_consumer_unregister** invocation should be made for each **ems_consumer_register** call. This quiesces event notification for a given EMS instance and filter group.
- Use **ems_consumer_stop** to destroy the consumer session with the EMS.

Administration

From time to administrators may want to audit the repositories and registries. EMS is designed to maintenance free, but occasional cleanup is inevitable. Administrative activities include:

- Updating the Consumer/supplier registry to add or remove consumers or suppliers.
- The Consumer/supplier registry may be updated to reflect new event types and schemas.
- The filter repository may become cluttered with forgotten filter lists. Or the filter lists for a consumer are to be deleted as a precursor to deleting the consumer from the registry.
- The event repository may have forgotten events. Where a consumer had registered for a set of events on behalf of a principal, but the events could not be delivered to the principal. This should be an anomaly. If events are requested, there is normally a more an casual interest in their reception.

2.2 Architecture

The EMS performs fan-in and fan-out operations. It receives events from suppliers. It supplies events to consumers. The EMS must perform these operations in an efficient, reliable and secure manner. The combination of basic operations and the constraints under which they must be performed lead to the general model given at the opening of this chapter.

A number of components may be gleaned from the general model. They are: the event persistent cache; the consumer/supplier registry; the filter repository; and the event channel.

2.2.1 Event Services API Overview

The EMS API can be divided into several interfaces grouped by function The different interfaces are: the registrations interface, the Event Type interface, the supplier Interface, the Filter Interface, the Consumer Interface, and the Management Interface.

The Registration interface allows registration with EMS for applications that are not suppliers or consumers. The Event Type interface provides support for manipulating the Event Type Database. Event filters can be created using the Event FIlter Interface. The Supplier interface allows suppliers to both register and send events to the EMS. The Consumer interface provides consumer setup routines and consumer registration routines. The EMS Management Interface allows management applications to administer the EMS databases, as well as the event service itself.

2.2.1.1 EMS Registration

The EMS registration routines allow clients, or users of the event service to both obtain a ‘handle’ to access the event service, as well as register with the event service how they intend to use it.

An event service handle is required to perform any operation with the event service. These operations can be event type operations, event filter operations or event service management operations. The following routines allow EMS clients to register with EMS:

ems_register	Register with the Event Service.
ems_unregister	Unregister with the Event Service.

Another type of registration requires consumers and suppliers to register with the event service to tell the event service that they are consumers or suppliers as well as what type of consumer or supplier they are. These routines are described in the supplier and consumer sections.

2.2.1.2 The EMS Event Type Interface

All events processed by EMS have an event type. Event types can be either generic, or defined by an event type schema.

EMS keeps a database of event types which consists of event type schemas. Routines are provided to manipulate the event types in the event type database.

The following routines summarize the EMS Event Type Interface:

ems_event_type_add	Add a new event type schema to the Event Type Database.
ems_event_type_delete	Delete an event type schema from the Event Type Database.
ems_event_type_get	Get an event type schema from the Event Type Database.
ems_event_type_get_list	Get a list of event type schemas from the Event Type Database
ems_event_type_free_list	Free the list of event type schemas.

2.2.2 Events

Conceptually, events consist of two objects. There is a base object (header) for all events. Optionally, there may be a derived object for the event.

2.2.2.1 Event Header

The event header contains the following information:

- The event identifier.
- The origin of the event.
- The severity of the event.
- The time when the event was generated.
- The time when the event was delivered.
- The priority of the event.

2.2.2.2 Event Data

The event data object contains the details of the event. This object is mapped by an Event Schema introduced in the next section. Event data can be self-describing; that is, it may consist of a sequence of *anys*. It may consist of a complex structure.

2.2.3 Schemas

Schemas are required for all events. The amount of information given in the schema may vary. Schemas are used to support filtering. Schemas are retained in a Schema Database. This database may not be required for all implementations. For example, it may be that a CORBA implementation uses the implementation repository for schema information.

2.2.4 Filters

Filters are used by suppliers and consumers to control which events get sent through the event channel. EMS supports the concept of two stage filtering. First stage filtering is applied to the supplier before receiving an event in the EMS. Second stage filtering is applied by EMS before forwarding events on to consumers.

The EMS API supports the second stage filtering, and provides routines to manipulate the EMS Event Filter database.

The following routines summarize the EMS Event Filter Interface:

ems_filter_add	Add a filter to the Event Filter Data base.

ems_filter_append	Append filter expressions to the Event Filter Database.
ems_filter_get	Get the contents of an event filter.
ems_filter_free	Free the storage used by an event filter.
ems_filter_delete	Delete a filter from the Event Filter Database.
ems_filter_get_namelist	Get a list of the names of all filters in the Event Filter Database.
ems_filter_free_namelist	Free a list of filter names.
ems_filter_get_list	Get a list of all the filters in the Event Filter Database.
ems_filter_free_list	Free the list of filters.

2.2.5 Event Channel

Conceptually, an EMS consists of a pair of event channels for supplier to consumer flow. One of the event channels connects the supplier with the EMS. The other event channel connects the consumer with the EMS. Logically, the EMS indicates to the supplier that the event has been received after it has been made persistent (assuming that there are consumers for the event or the EMS does not perform this optimization), insuring that the event is not lost during the transfer. Likewise, once the consumer indicates to the EMS that it has received the event, the EMS removes the event from the persistent store.

The channel pair provides a means of decoupling consumers and producers. In addition, it provides a mechanism for fan-in and fan-out of events.

2.2.6 Consumers

EMS consumers are both clients and servers. The EMS consumer interface provides support for the steps required to implement an event consumer. A routine is provided to perform consumer setup, **ems_consumer_start**, and consumer cleanup, **ems_consumer_stop**. These routines should be called when a consumer starts, and before consumer shutdown. Once a consumer has started, then an event handler routine must be registered to specify what consumer supplied routine should be called to process an event. This routine is **ems_consumer_handler_register**. After setup is complete, then a consumer must register with the EMS, and set up any filters that it wants to use to control which events get forwarded to this consumer.

The EMS maintains a consumer database to keep track of all registered consumers. Registering and unregistering with the EMS adds and deletes consumers to and from the database.

Two types of consumers, to correspond with the OMG Event Service Model, are supported, a push consumer and a pull consumer. A consumer would use the appropriate registration API to designate which type of consumer is desired

The following routines summarize the EMS Consumer Interface.

ems_consumer_start	Called to start an event consumer.
ems_consumer_stop	Called to stop an event consumer.
ems_consumer_handler_register	Register a consumer event handler.
ems_push_consumer_register	Register a push consumer with the Event Service.
ems_pull_consumer_register	Register a pull consumer with the Event Service.
ems_consumer_unregister	Unregister a consumer with EMS.
ems_add_filter_to_group	Add a filter name to a consumers event filter group.
ems_delete_filter_from_group	Delete a filter name from a consumers event filter group.
ems_get_filter_group	Get the list of filter names that comprise a consumers event filter group.

2.2.7 Suppliers

The EMS Supplier interface allows suppliers to send events to the Event Service. First stage filtering would be applied before making any EMS interface calls sending any events from the supplier to the event service. Note, the EMS will not accept event types from a supplier that the supplier has not registered.

The following routines allows suppliers to send events to the EMS:

ems_supplier_send	Send an event to EMS.
ems_push_supplier_register	Register a push supplier with the EMS.
ems_pull_supplier_register	Register a pull supplier with the EMS.
ems_supplier_unregister	Unregister a supplier with the EMS.

2.2.8 Management

The event service also provides a management API to allow administration of the event service as well. The management routines allow manipulation of event service attributes, and the consumer database. The EMS Filter Interface and Event Type Interface could also be used to administer the Event service, but those interfaces can also be used by suppliers and consumers.

The following routines summarize the EMS Management Interface:

ems_mgmt_list_ems	list all hosts running the event service.
ems_mgmt_free_ems	free the host list
ems_mgmt_list_attributes	lists attributes for a specific EMS.
ems_mgmt_free_attributes	free a list of ems attributes.
ems_mgmt_list_consumers	list consumers registered with EMS.
ems_mgmt_free_consumers	free a consumer list.
ems_mgmt_add_consumer	add a consumer to the Consumer Database.
ems_mgmt_add_consumer_of_event	permit the consumer to receive events of a specified type.

ems_mgmt_delete_consumer	delete a consumer from the Consumer Database.
ems_mgmt_delete_consumer_of_event	remove permission of the consumer to receive events of a specified type.
ems_mgmt_delete_filter_from_group	delete a filter name from a consumers filter group.
ems_mgmt_add_filter_to_group	add a filter name to a consumers filter group.
ems_mgmt_get_filter_group	get the list of names in a consumers filter group.
ems_mgmt_add_supplier	add a supplier to the Supplier Database.
ems_mgmt_add_supplier_of_event	permit the supplier to send events of the specified type.
ems_mgmt_list_suppliers	list suppliers registered with EMS.
ems_mgmt_free_suppliers	free a supplier list.
ems_mgmt_delete_supplier	delete a supplier from the Supplier Database.
ems_mgmt_delete_supplier_of_event	revoke permission for the supplier to send a type of event.

2.3 Performance

An event management system is a fundamental component. It is used in conjunction with management applications leveraging its services. Together these provide the basis for maintaining service availability by:

1. Giving timely warning of impending problems (e.g., file capacity thresholds).
2. Notifying system administrators of failing processes and system components.
3. Quickly identifying root causes to problems in increasingly complex systems.
4. Automatically fixing problems before service levels are degraded.

5. Integrating application-specific event mechanisms so system correlation can be done at a higher level (e.g., network outages are the root cause of many application errors; and the administrator needs his attention drawn to the root cause).

Key attributes of the EMS are:

- Lightweight — minimal network/system load

  A key requirement is that the event service have good performance characteristics, and that it not degrade performance of the network. This service must be efficient enough for real-time performance monitoring of system level activity.
- Extensible API

  The ability to define and extend events and event contents.
- Inter-operable across the network

  This attribute addresses the ability event management system products from different vendors to inter-operate without *a priori* knowledge of the specific vendor's offering.
- Robust

  Defined as the reliable delivery of events.

2.4 Reliability

Since event management applications are responsible to notify any system and network problems to responsible operators, the management application itself as well as the underlying event services must be absolutely reliable. The following issues outline the areas that must be addressed by an EMS implementation:

- no loss of events
- self-monitoring
- stable processes

2.4.1 No Loss of Events

EMS implementations are responsible for all events from the point in time the events are sent by a supplier until they are received by all consumers registered for notification of the respective event. Implementations must guarantee reliable delivery of all events from suppliers to consumers; in case the network connection to consumers is down, local buffering must be applied to enable later retransmission of events.

2.4.2 Self-Monitoring

Since an EMS is responsible for the delivery of problem notifications to a management application, it is essential that an appropriate mechanism is provided for the monitoring of EMS and the underlying components it depends on. In case of any failure, consumers must be notified of the failure in order to enable them to react appropriately.

2.4.3 Stable Processes

The stability of processes is an issue that applies to all pieces. However, due to its importance to applications that depend on event services, an implementation must be extremely robust even in moments of high network or CPU load on the local machine. Since event services is required in such exceptional states, this must be reflected in the robustness of its implementation.

2.5 Security

A multi-level access security model is required for the EMS. There are two categories of access for the EMS. One category deals with a consumer's or supplier's ability to register (bind) with an EMS. The second category focuses on the permissions for specific objects (i.e., event types). For example, a supplier may only be permitted to insert a specific event type. Or, a consumer may only be permitted to view certain event types. If multi-level access is not available, then an implementation may be configured such that all users of the specific EMS have the same permission sets for all objects. Expanding upon this, a set of EMS instances may be configured for a given node, where each instance supports a specific set of users.

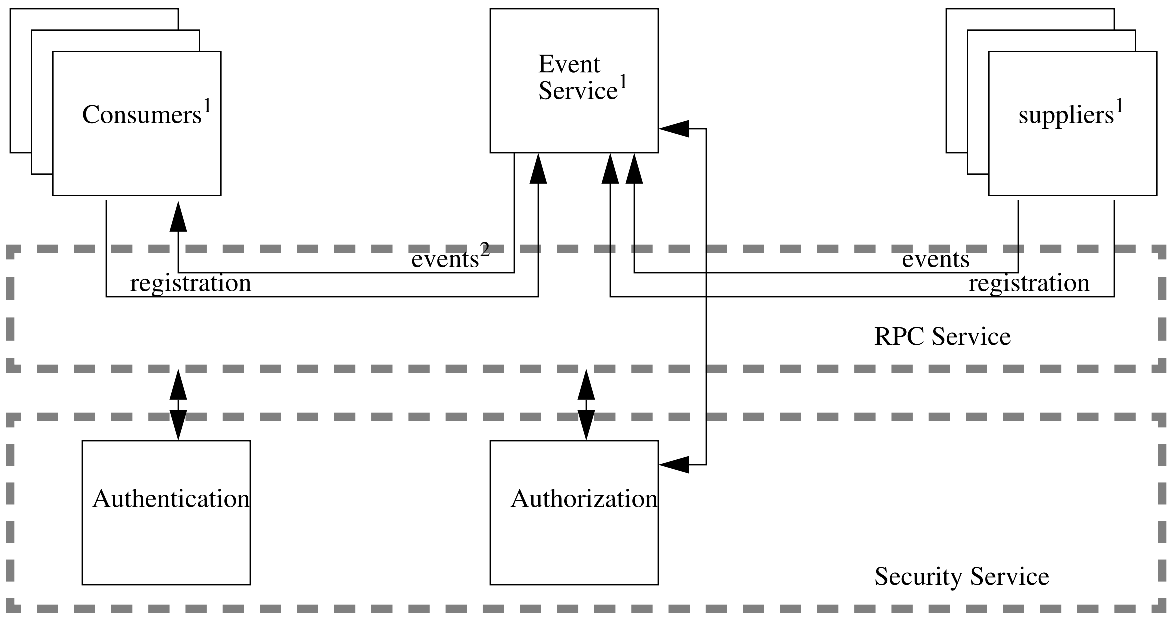
The EMS will operate within the security framework of the host environment. It is paramount that an implementation work within the host security mechanism. Security policies can be quite complex. The use role-based security schemes compounds the situation. For EMS implementations to have wide acceptance, they cannot insist upon using private security mechanisms.

The manifestation of a security policy is transparent to consumers and suppliers. The use of the security contexts of the consumers and suppliers, assists interoperability and usability. The sole adapter to be used for an EMS may be the authorization implementation for the vendor's abstract authorization class.

The use and understanding of a security architecture does not depend on a full-understanding of the underlying object implementations and interactions of the Event Service, consumers, and suppliers.

The conceptual model for the use of security with the EMS is depicted in the following figure. The keys to the security model are:

1. Neither consumers nor suppliers are aware of, i.e., need code to support, a security mechanism. They simply inherent the security context of their logon group.
2. The EMS requires object level security granularity, i.e., multi-level security. As a result, it must perform authorization checks for consumers and suppliers.
3. The administration functions are not depicted in this view. They are deemed to be outside the scope of the EMS specification.
4. The EMS specification is written in accordance with the proposal put forth in GSSAPIEXT to the extent that the latter provides specifications for multi-level access. For example, from the perspective of a DCE-based EMS, the calls to the ACL manager to determine whether a consumer (or supplier) has permission for a given operation on a specific event type (i.e., a specific object) are:
   - *rpc_binding_inq_auth_client()* to obtain the authentication and authorization information from the binding handle for an authenticated client. In particular, to obtain the client's privilege attribute certificate (PAC) which is used in the test call.
   - *sec_acl_bind()* to obtain the handle of the object in question.



1 The security context is derived from the normal means, e.g., set during logon or altered via system calls (setuid, setgid, etc.).

2 Consumers, like the EMS, are both clients and servers. As a result authentication and authorization are performed for both the registration and the event flow. The difference between the Consumer and the EMS is that the Consumer does not perform the multi-level check on the contents of the event container.

Conceptual Role of Security for the DCE-based EMS

- *sec_acl_test_access_on_behalf()* to test access to an object on behalf of another process, mapping the type of access implied in the original RPC call to a permission mask (e.g., read, write, insert, delete, etc.).

It is not clear whether GSSAPIEXT provides this support or has left it as an implementation specific aspect of security.

Changes in a principal's security profile may affect their ability to participate as a consumer (or supplier) either during or across conversations.

2.5.1 Global Namespace

The global namespace from a security perspective is concerned with the transfer and/or conversion of a principal's security context within/across security domains.

The namespace affects consumers and the EMS.

The namespace is transparent to suppliers and the EMS.

Security context transfers must be capable of mapping the principal and the principal's attributes, e.g., PACs for DCE-based security.

The mapping transformations must be two-way, since the information flow is both from the consumer to the EMS and *vice versa*.

The event flow (from the EMS to a consumer) does not require multi-level security at the consumer side.

2.5.2 Security Objects

There are two types of security objects in EMS. Both types are controlled by permission sets. The first type of security object concerns manipulation of EMS attributes, and EMS Databases. Here is a list of the EMS security objects of this type.

ems-server	controls access to the event service.
event-types	controls access to the Event Type Database
filters	controls access to the Filter Database
consumers	controls access to the Consumer Database
suppliers	controls access to the suppliers Database

There second type of security object concerns manipulation and access to objects inside of the databases. These objects are classified by category.

event-type	controls access to a given event type in the Event Type Database
filter	controls access to a given filter in the Filter Database

The following sections will describe what operations can be controlled on each security object.

Event Service Attributes Security Object

The **ems-server** security object controls access to the event service and its attributes. The two permissions on this security object are:

control modify this security object.

read read or get the attributes for an Event Service.

write write or modify the attributes for an Event Service.

Event Type Database Security Object

The **event-types** security object controls access to the Event Type Database. The permissions associated with this security object are:

control modify this security object.

delete delete an event type from the Event Type Database.

insert insert or add an event type to the Event Type Database.

read read or get an event type schema from the Event Type Database.

Filter Database Security Object

The **filters** security object controls access to the Filter Database. The permissions associated with this security object are:

control modify this security object.

delete delete a filter from the Filter Database.

insert insert or add a filter to the Filter Database.

read read or get a filter or the list of filters from the Filter Database.

Consumer Database Security Object

The **consumers** security object controls access to the Consumer Database. The permissions associated with this security object are:

control modify this security object.

delete delete a consumer from the Consumer Database.

insert insert (by registering) a consumer in the Consumer Database.

read read or get a list of consumers, or a specified consumer's filter group from the Consumer database.

write modify a consumers filter group.

Supplier Database Security Object

The **suppliers** security object controls access to the supplier Database. The permissions associated with this security object are:

control modify this security object.

delete delete a supplier from the Consumer Database.

insert insert (by registering) a supplier in the supplier Database.

read read or get a list of suppliers from the supplier database.

Event Type Security Objects

The event type object security object controls access to a given event type in the Event Type Database. An security object of this type is created for every Event type in the Event Type Database. The permissions associated with this security object are:

control modify this security object.

delete delete this event type from the Event Type Database.

read read or get the event type schema for this event type from the Event Type Database.

Filter Security Objects

The filter object security object controls access to a given filter in the Filter Database. A security object of this type is created for every filtering the Filter Database. The permissions associated with this security object are:

control modify this security object.

delete delete this filter from the Filter Database.

read read or get the filter expressions for this filter from the Filter Database.

write write or append a filter expression to this filter in the Filter Database.

2.5.3 Network Communications

Authentication is provided by the environment.

Security context support is provided by the environment.

Data encryption support is provided by the library. In a OMG CORBA environment, for example, this would be provided by the object system.

Encryption is at the conversation level and not at the object level. This eases the administrative burden in the sense that encryption semantics can be defined at the communications layer. The EMS need not be aware of the encryption mechanism.

2.6 Internationalization

Many of the initial suppliers for the EMS are likely to be legacy applications with EMS wrappers. These suppliers are expected to emit text based data. The exchange of textual data as the content of events introduces a number of problems.

- The language of the supplier may not be that of the consumer.
- The character set of the supplier may not be available to the consumer.
- Cultural data, e.g., time and date format, may be embedded within the text-based events.

Solutions to these problems (from the perspective of wrapped legacy systems) are limited. Internationalization of textual data usually leads to rewriting sections of the original application. Here is a partial list of changes:

- Many of the language issues may be solved through the use of message catalogs. Unfortunately, the catalogs must be accessible to the consumers. For message based solutions, the message number for the message skeleton is provided as a parameter. The number of insertions is another parameter. Each of the inserts, converted to an internal form (e.g., a timestamp may be shipped as a *time_t*), is provided as a typed (*any*) parameter. A complimentary adapter is required at the consumer that recognizes the event type and provides the message presentation services.
- There is little that can be done for missing character sets even with the use of a message catalog. Failures may occur when the insert is a character string, e.g., a file name, and the codeset for the language is not available at the consumer.
- Most cultural problems are skirted by transmitting the internal form of the data and not the presentation form. This may be used in conjunction with a message catalog or in isolation.

For suppliers built for the EMS, there are data types for all transmitted information, including constructed types for multi-byte and UCS character sets. While this does not eliminate the problems associated with unsupported character sets, it does provide reasonable semantics for failures due to this condition. Hence, sending textual data (e.g., message inserts for file names) in its specific character set mapping may permit the consumer to see a message in part, even though specific inserts might not be viewable.

2.7 Interoperability

Any two implementations of the EMS API described in this specification will be inter-operable. Consumers and suppliers would be capable of connecting to and operating with any conforming EMS. For implementations utilizing the same transport, e.g., CORBA V2, the implementations are directly inter-operable. For implementations utilizing different transports, they inter-operate through a bi-directional gateway.

An Event Service maintains one or more repositories. The composition of these repositories is beyond the scope of this specification. The interfaces to these repositories is in the administrative domain.

2.7.1 Different Event Management Applications

Since event management applications are consumers receiving events via the registration APIs, standardizing these APIs, as well as the common event format, address this particular requirement.

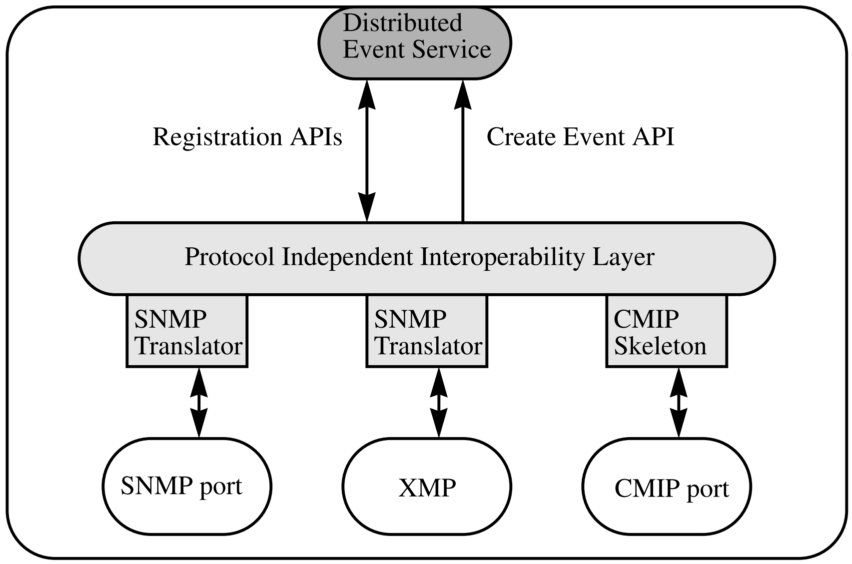
In particular this means, that regardless of the implementation of the EMS, the event management application of choice can be used for further processing and presentation.

2.7.2 Different Event Protocols

This is probably the most important interoperability aspect. Basically it addresses the need to integrate:

- Event suppliers based on different protocols (e.g. SNMP traps).
- Event consumers based on different protocols (e.g. OMG based event service implementation)

As stated at the beginning of this section, gateways are used to provide interoperability across event service implementations. The EMS APIs are compatible at the source level. Suppliers and consumers built for different EMS implementations may be connected to a given EMS via adapters. The idea is to implement a protocol independent interoperability layer connecting the different worlds.



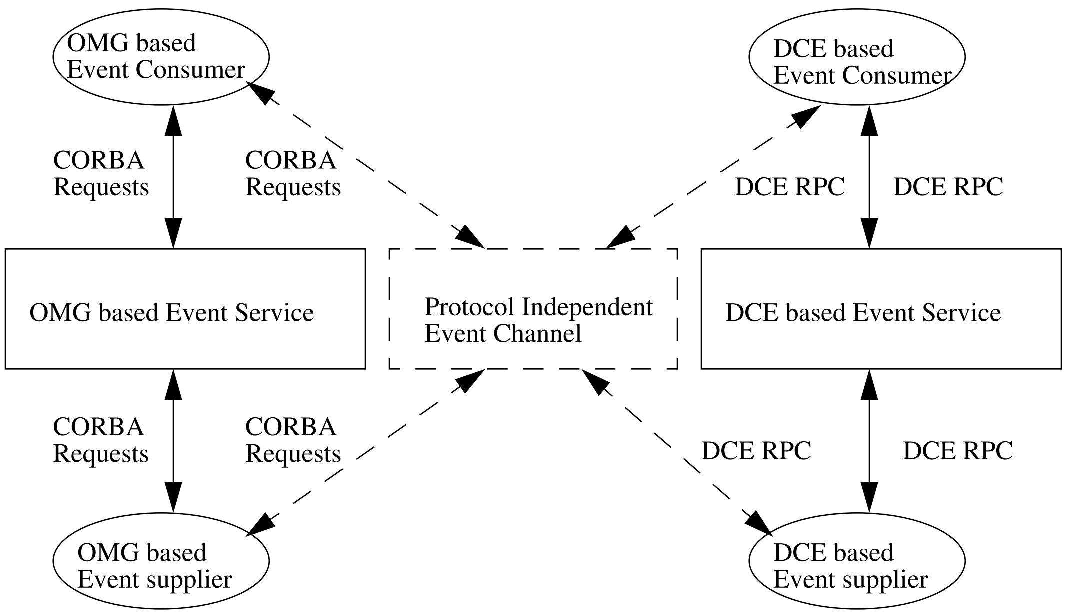
Managed Node

The translator modules establish access to the different event worlds by translating the particular event format (e.g. SNMP trap) into the common standardized format, and vice versa.

The protocol independent layer establishes the access to the event service, utilizing the standardized event creation and registration APIs. With that layer it's possible to centrally manage events from different protocols, and to communicate between different event service implementations (DCE vs. OMG).

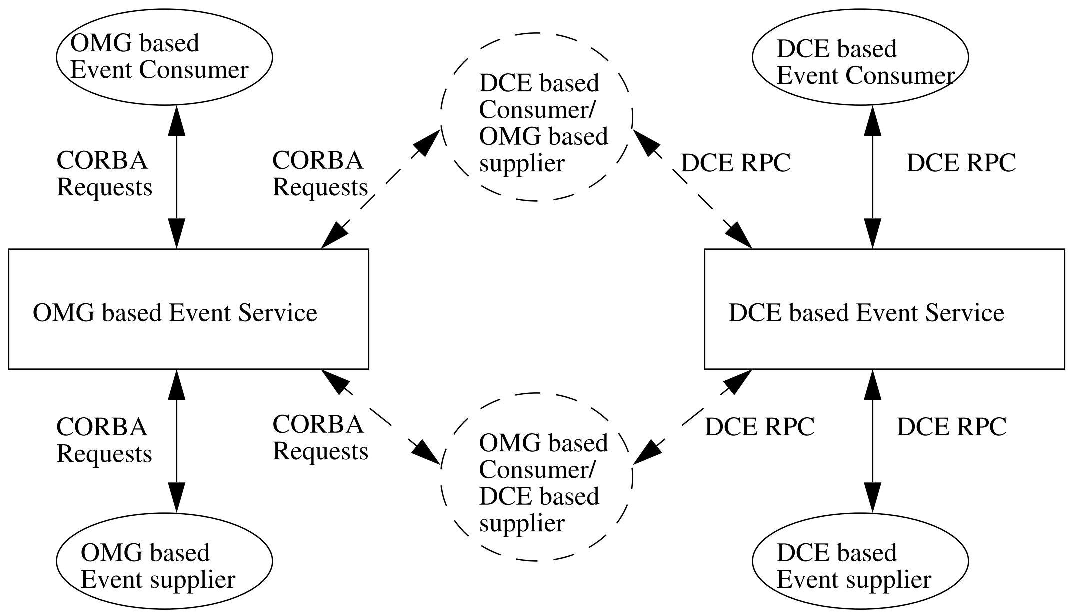
2.7.3 Interoperability of EMS Implementations

The previous section alluded to a general model for interoperability EMS implementations. Specifically, there is a general requirement for bi-directional interoperability between implementations. The figure below depicts the desired situation with respect to interoperability, using a CORBA-based and a DCE-based implementation as a prototypical example. The solid lines depict components that are easily conceivable using existing technologies. The dashed lines depict the components that must be established using a solution to the interoperability problem described in this section.



The figure above is, undoubtedly, a simplification of a complex problem that plagues the industry. But it provides a logical view of what is needed to solve interoperability. In general, some component is needed to translate events produced by a supplier using one transport to events that can be consumed by another transport event consumer, and vice versa. Such a component is purely logical, and may in fact be implemented using one to several processes.

As mentioned in the previous paragraph, the protocol independent event channel depicted in the figure is purely logical, and several vastly different implementations are imaginable. Probably the simplest, although not necessarily the most efficient, implementation would be to start with an environment that supports both implementations. Then, a component could be implemented that is both a consumer and a supplier. This scenario is depicted in the following figure.



2.8 Examples

2.8.1 Intelligent Agent

The intelligent agent example serves to demostrate use of many of the EMS methods. The idea is that a client (a consumer) sends an agent to a node to manage a resource. The client must be authorized to perform this function and an agent manager must *a priori* on the target node. Overview:

1. Connect with the agent factory.

2. Setup the transfer of initialization information.
3. Transfer the agent and applettes.
4. Setup the supplier with the EMS.
5. Setup the consumer with the EMS.
6. Start the supplier.
7. Send completion status to the consumer with a reference to the EMS conversation.

2.8.1.1 Initial Connection Flows

1. Connect to the agent manager factory on the destination node. The factory needs to know the event recipient, quality of service, and the agent. The factory returns an object reference for the agent.
2. The agent may consist of several applettes. The agent and each applette is shipped to the factory. The factory sets up the agent environment, insuring that it can be restarted in the event of a failure.

2.8.1.2 Setup The Supplier With The EMS

This work is performed by the factory.

1. Use *ems_register()* to bind to the EMS.
2. Use *ems_event_type_add()* to add the event type, the identity of the agent in this case.
3. Use *ems_mgmt_add_supplier()* to add the agent.
4. Use *ems_mgmt_add_supplier_of_event()* to connect the agent to the event type as a supplier.
5. Use *ems_unregister()* to disconnect from the EMS.

2.8.1.3 Setup The Consumer With The EMS

This work is performed by the factory.

1. Use *ems_register()* to bind to the EMS.
2. Use *ems_mgmt_add_consumer_of_event()* to permit the consumer to receive events of the given type. This method should not fail, because the factory used the EMS to ensure that the consumer utilize this facility.
3. Use *ems_filter_add()* to indicate that the consumer is to receive all events from the agent.

4. Use *ems_unregister()* to disconnect from the EMS.

2.8.1.4 Start The Supplier

1. The factory for the intelligent agent has placed the agent and the applettes in a directory. In addition to the executables, the factory has placed state and context information in the directory.
2. The factory spawns the agent in the context of the directory it previously built. The agent is spawned with the security credentials and permissions of the consumer.

2.8.1.5 The Supplier

1. The agent executes with the directory created by the factory as its current working directory.
2. The agent does not know the difference between the initial invocation and an invocation after a failure, e.g., power outage.
3. Use *ems_push_supplier_register()* to bind to the EMS.
4. For each event to be sent, use *ems_supplier_send()*.
5. Before terminating the agent, use *ems_supplier_unregister()* to remove the conversation with the EMS.
6. Set the conversation state to completed. The conversation manager will remove the files and directory information.
7. Conversation manager remove supplier.

2.8.1.6 Conversation Manager Remove Supplier

The conversation manager cleans up after the agent, removing it from the EMS.

1. Use *ems_register()* to connect to the EMS.
2. Use *ems_mgmt_delete_supplier_of_event()* to disassociate the agent from the event type.
3. Use *ems_mgmt_delete_supplier()* to remove the agent from the EMS.
4. Use *ems_unregister()* to disconnect from the EMS.

2.8.1.7 Conversation Manager Remove Conversation

The consumer normally asks the conversation manager to clean up the agent, et al, when it no longer wishes the agent to exist at the given node. The steps it follows are:

1. Locate the directory for the agent.
2. Conversation manager remove supplier.
3. Conversation manager remove consumer.
4. Conversation manager remove event type.
5. Remove the agent directory and its contents.

2.8.1.8 Conversation Manager Remove Consumer

Here, the conversation manager is removing the association between the consumer and the event type.

1. Use *ems_mgmt_delete_consumer_of_event()* to remove the association between the consumer and the event type. This has the side-affect of removing all undelivered events of this type for this consumer.

2.8.1.9 Conversation Manager Remove Event Type

Here, the conversation manageer is attempting to remove the event type. This method will fail when: (1) there are other consumers the undelivered events of this type; or, (2) the event type does not exist.

1. Use *ems_event_type_delete()* to remove the event type.

2.8.1.10 The Consumer

This section chronicles the flow of the consumer, when the event recipient is the consumer.

1. Initial connection setup.
2. Use *ems_consumer_start()* to connect or reconnect to the EMS. The library determines whether it is an initial connection or a reconnect based on the input argument. The library returns the object reference representing the conversation.
3. Use *ems_consumer_handler_register()* to present a method to receive event notifications to the EMS library.
4. Use *ems_push_consumer_register()* to set the EMS library notification mode.

5. Use *ems_consumer_stop()* to disconnect the conversation. The conversation is to terminated; rather, the delivery of events to the consumer is interrupted.

CHAPTER 3 Data Formats

The EMS Data structure section is divided into several sub-sections according to the usage of the data structures. Following is a brief description of each of the data structure sub-sections.

The data structure section first defines some generic types that will be used throughout the definition of EMS. Some of these types define standard C data types and some machine dependant types so that implementations of EMS can define them to be machine dependent. There are also some other non-scalar types that are use to standardize use of items such as timestamp, and character strings.

Event attributes are used in several other data structures in EMS, such as the event data structure as well as the event type schema data structure, and the attribute list data structure. Event attributes contain an attribute name as well as a self defining value which has a format and a value.

The EMS Event structure contains a fixed header, and a variable size array of event attributes. The header contains fields that identify the event and its type, the event origin, what severity the event is, as well as a place for the Event Service to put the received time as well as the time the event was delivered to an interested consumer. An additional field has been added to mark the priority of an event.

The event type data structures allow the construction of event type schemas.

Event filters are constitute in EMS using the event filter data structures. These structures allow building filters by first constructing event expressions that contain attribute names, operators, and attribute values. Expressions are then collected into expression lists which then become event filters. There is also a structure for event filter name lists which are used to define consumer filter groups and to return lists of filters from the event filter database.

The consumer and supplier data structures are used to return information about consumers and suppliers from EMS.

There is also an attribute list data structure which allows getting and setting attributes in EMS. The actual values of the attributes are not defined in the specification.

The event list data structure allows management of undelivered events. A list of undelivered events can be returned in an event list.

The ems_handle, is an opaque data type which is used in calls to EMS routines. An ems_handle represents a connection to an Event Service on a particular host. When calling EMS routines, one of the EMS registration routines is called to unitize the handle, and it is used in all subsequent calls to EMS routines that want to affect the Event Service on that host.

3.1 EMS Generic Data Types

The data structure section first defines some generic types that will be used throughout the definition of EMS.

Scalar types

EMS has defined generic C data types which can be defined per implementation these types correspond to standard data types.

```
typedef unsigned char      ems_boolean;
#define ems_false          false
#define ems_true           true
typedef unsigned char      ems_byte;
typedef unsigned char      ems_char;
```

```
typedef signed char            ems_small_int;
typedef unsigned char          ems_usmall_int;
typedef short int              ems_short_int;
typedef unsigned short int     ems_ushort_int;
typedef long int               ems_long_int;
typedef unsigned long int      ems_ulong_int;
struct ems_hyper_int_rep_s_t {
        ems_long_int           high;
        ems_ulong_int          low;
} ems_hyper_int;
struct ems_uhyper_int_rep_s_t {
        ems_ulong_int          high;
        ems_ulong_int          low;
} ems_uhyper_int;
typedef float                  ems_short_float;
typedef double                 ems_long_float;
```

Strings

Strings are used throughout the EMS data structures, and as parameters in the API.

```
typedef char *                 ems_string_t;
```

Unique Identifier

ems_uuid_t is a data structure which contains a unique identifier which is used to uniquely identify different objects in EMS.

```
typedef struct uuid_t {
       ems_ulong_int       time_low;
       ems_ushort_int      time_mid;
       ems_ushort_int      time_hi_and_version;
       ems_usmall_int      clock_seq_hi_and_reserved;
       ems_usmall_int      clock_seq_low;
       ems_byte            node[6];
```

```
} ems_uuid_t;
```

Time Stamp

The EMS time structure contains a timestamp represented in Coordinated Universal Time (UTC).

```
typedef struct utc {
    ems_byte        char_array[16];
} ems_utc_t;
```

Error Status

The EMS error status is used to return status to callers of EMS routines to indicate whether the call succeeded or not. The meanings of the error status values can be found in a later section (See Section 3.10, “XEMS Status Codes”).

```
typedef ems_ulong_int        ems_error_t;
```

Event Type

EMS Event types are used to classify events.

```
typedef ems_uuid_t        ems_event_type_t;
```

3.2 EMS Event Attributes

Event Attribute Types

The event attribute type is used to specify the data type of an event attribute. The attribute type specifies what format the data is in the event attribute value union (*ems_attr_value_t*).

All event attribute types are defined as

```
typedef ems_ushort_int    ems_attr_type_t;
```

An event attribute type can be one of the following

TABLE 3-1. Event Attribute Type Specifiers

Attribute Type	Data Type
ems_c_attr_small_int	ems_small_int
ems_c_attr_short_int	ems_short_int

TABLE 3-1. Event Attribute Type Specifiers

Attribute Type	Data Type
ems_c_attr_long_int	ems_long_int
ems_c_attr_hyper_int	ems_hyper_int
ems_c_attr_usmall_int	ems_usmall_int
ems_c_attr_ushort_int	ems_ushort_int
ems_c_attr_ulong_int	ems_ulong_int
ems_c_attr_uhyper_int	ems_uhyper_int
ems_c_attr_short_float	ems_short_float
ems_c_attr_long_float	ems_long_float
ems_c_attr_boolean	ems_boolean
ems_c_attr_uuid	ems_uuid_t
ems_c_attr_utc	ems_utc_t
ems_c_attr_severity	ems_severity_t
ems_c_attr_byte_string	ems_byte *
ems_c_attr_char_string	ems_char *
ems_c_attr_bytes	see structure

Byte strings and character strings are terminated with a 0 (zero) byte.

Here are the actual attribute format type values:

```
#define ems_c_attr_small_int     (0)
#define ems_c_attr_short_int     (1)
#define ems_c_attr_long_int      (2)
#define ems_c_attr_hyper_int     (3)
#define ems_c_attr_usmall_int    (4)
#define ems_c_attr_ushort_int    (5)
#define ems_c_attr_ulong_int     (6)
#define ems_c_attr_uhyper_int    (7)
```

```
#define ems_c_attr_short_float     (8)
#define ems_c_attr_long_float      (9)
#define ems_c_attr_boolean         (10)
#define ems_c_attr_uuid            (11)
#define ems_c_attr_utc             (12)
#define ems_c_attr_severity        (13)
#define ems_c_attr_byte_string     (15)
#define ems_c_attr_char_string     (16)
#define ems_c_attr_bytes           (17)
```

Event Attribute Values

The event attribute value union is a self defining data structure which has an attribute type specifier (*format*) which tells what type of data is in the union, and then appropriate union members (*tagged_union.<format_specific_field_name>*) to hold the value of the data specified.

```
typedef struct ems_bytes_s_t {
    ems_ulong_int size;
    ems_byte      *data;
} ems_bytes_t;
typedef struct {
    ems_attr_type_t format;
    union {
      /* case(s): ems_c_attr_small_int */
      ems_small_int small_int;
      /* case(s): ems_c_attr_short_int */
      ems_short_int short_int;
      /* case(s): ems_c_attr_long_int */
      ems_long_int long_int;
      /* case(s): ems_c_attr_hyper_int */
      ems_hyper_int hyper_int;
      /* case(s): ems_c_attr_usmall_int */
      ems_usmall_int usmall_int;
      /* case(s): ems_c_attr_ushort_int */
      ems_ushort_int ushort_int;
      /* case(s): ems_c_attr_ulong_int */
```

```
        ems_ulong_int ulong_int;
        /* case(s): ems_c_attr_uhyper_int */
        ems_uhyper_int uhyper_int;
        /* case(s): ems_c_attr_short_float */
        ems_short_float short_float;
        /* case(s): ems_c_attr_long_float */
        ems_long_float long_float;
        /* case(s): ems_c_attr_boolean */
        ems_boolean bool;
        /* case(s): ems_c_attr_uuid */
        ems_uuid_t uuid;
        /* case(s): ems_c_attr_utc */
        ems_utc_t *utc;
        /* case(s): ems_c_attr_severity */
        ems_severity_t severity;
        /* case(s): ems_c_attr_byte_string */
        ems_byte *byte_string;
        /* case(s): ems_c_attr_char_string */
        char *char_string;
        /* case(s): ems_c_attr_bytes */
        ems_bytes_t bytes;
    } tagged_union;
} ems_attr_value_t;
```

Event Attribute

Event attributes contain an event attribute name/value pair which define an event attribute. Event attributes are used in events to provide self defining data as part of an event. Event attributes are also used in event type schema's to define the contents of an event of specific event type. The *name* field specifies the attribute name, and the *value* field contains the value and format of an event attribute.

```
typedef struct ems_attribute_s_t {
    ems_string_t        name;
    ems_attr_value_t    value;
} ems_attribute_t;
```

3.3 EMS Event Structure

Event Identifier

An event identifier uniquely identifies a given event. Each event has both an event *type* which is unique to all events of this type, and an event *id* which is unique to a specific event.

```
typedef struct ems_eventid_s_t {
    ems_event_type_t     type;
    ems_uuid_t           id;
} ems_eventid_t;
```

Event Type

An event type specifies the unique id for a given event type.

TABLE 3-1. Default Event Types

Event Type	Event Type Name
ems_c_generic_type	Generic

Events of type *Generic*, do not have event type schema's associated with them, and can only be filtered by expressions with header attributes in them (See Table 3-2: Event Header Attributes).

Network Name

A network name identifies the network name of a given host machine.

The name service specifies which name service recognizes the given network name.

```
typedef enum {
    ems_ns_other,
    ems_ns_dns,
    ems_ns_dce,
    ems_ns_x500,
    ems_ns_nis,
    ems_ns_sna
} ems_nameservice_t;
```

The *ems_netaddr_t* structure specifies the actual network name. It can be interpreted according to the name service specified. *ems_octet_t* defines an 8-bit field. *len* specifies how many 8-bit quantities there are in *name*.

```
typedef char ems_octet_t;
typedef struct ems_netaddr_s_t {
    ems_ulong_int      len;
    ems_octet_t        name[1];
} ems_netaddr_t;
```

The *ems_netname_t* consists of *service* which specifies which name service recognizes the name specified by *netaddr.*

```
typedef struct ems_netname_s_t {
    ems_nameservice_t    service;
    ems_netaddr_t        *netaddr;
} ems_netname_t;
```

For a DCE hostname, the following example will set the *ems_netname_t* structure;

```
static char * dce_hostname = "/.:/hosts/eagle.austin.ibm.com";
ems_netname_t      netname;
netname.service = ems_ns_dce;
netname.netaddr->len = strlen( dce_hostname )+1;
netname.netaddr->name = (char *)malloc( netname.netaddr->len );
strcpy( netname.netaddr->name, dce_hostname );
```

Event Origin

The event origin specifies where the event originated (i.e. the supplier). The origin specifies the *netname* of the host where the supplier is running, the name of the supplier, *descname*, and supplier process identification (*pid*, *uid*, *gid*). These values may not be meaningful for all hosts.

```
typedef struct ems_origin_s_t {
    ems_netname_t      netname;
    ems_string_t       descname;
    ems_ulong_int      pid;
```

```
        ems_ulong_int        uid;
        ems_ulong_int        gid;
} ems_origin_t;
```

Event Severity

The event severity specifies the severity of the event.

```
typedef enum {
        ems_sev_info,
        ems_sev_fatal,
        ems_sev_error,
        ems_sev_warning,
        ems_sev_notice,
        ems_sev_notice_verbose,
        ems_sev_debug
} ems_severity_t;
```

Event Priority

The event priority specifies the priority of the event.

```
typedef ems_ulong_int        ems_priority_t;
```

Event Header

The event header describes the fixed part of the event data structure. The header contains the *eventid*, the *origin* of the event, the *severity* along with the time the event was both *received* at EMS, and *delivered* to the consumer.

```
typedef struct ems_hdr_s_t {
        ems_eventid_t        eventid;
        ems_origin_t         origin;
        ems_severity_t       severity;
        ems_utc_t            received;
        ems_utc_t            delivered;
        ems_priority_t       priority;
} ems_hdr_t;
```

A set of Filter attributes are provided for event header filtering. The following names can be used for the filter attribute in an event filter expressions.

TABLE 3-2. **Event Header Attributes**

Attribute Name	Attribute Type
eventid.id	ems_c_attr_uuid
eventid.type	ems_c_attr_uuid
origin.netname.service	ems_c_attr_ulong
origin.netname.netaddr	ems_c_attr_bytes
origin.descname	ems_c_attr_char_string
origin.pid	ems_c_attr_ulong
origin.uid	ems_c_attr_ulong
origin.gid	ems_c_attr_ulong
severity	ems_c_attr_severity
received	ems_c_attr_utc
received.tod	ems_c_attr_char_string
received.mday	ems_c_attr_ushort_int
received.year	ems_c_attr_ushort_int

TABLE 3-2. Event Header Attributes

Attribute Name	Attribute Type
received.wday	ems_c_attr_ushort_int
received.yday	ems_c_attr_ushort_int

Event

The *ems_event_t* structure contains a fixed part, the event *header*, and a variable part, the event data *item*s. Each data item is a self-defining value which contains an attribute type, and attribute data. *Count* specifies how many data items are in the event.

```
typedef struct ems_event_s_t {
    ems_hdr_t          header;
    ems_ulong_int      count;
    ems_attribute_t    item[1];
} ems_event_t;
```

3.4 EMS Event Types

The EMS Event Type structures are used to define the EMS Event types.

Event Type Schema

The event type schema is used to define an event type. It consists of an event type id, *type*, a *name* field which specifies the name of the event type, and a list of event type attributes describing the format of this event type. Size specifies the number of attributes in an event type. The event type schemas only specifies the fixed part of an event. An event can have as many unnamed attributes following the list of attributes specified here.

```
typedef struct ems_event_schema_s_t {
    ems_event_type_t    type;
    ems_string_t        name;
    ems_long_int        size;
    ems_attribute_t     attribute[1];
} ems_event_schema_t;
```

Event Type List

The event type list contains a list of *size* event type *schemas*.

```
typedef ems_event_schema_t     *ems_schema_ptr_t;
typedef struct ems_event_type_list_s_t {
        ems_long_int               size;
        ems_schema_ptr_t        schema[1];
} ems_event_type_list_t;
```

3.5 EMS Event Filters

The event filter data structures allow the definition of both event filters, and event filter lists.

Attribute Operators

Attribute operators define the boolean operation to perform on the attribute name, and the attribute value in the event filter expression. The attribute operator type is defined as:

```
typedef ems_ushort_int        ems_attr_op_t;
```

TABLE 3-1. Attribute Operators

Attribute Operator	Description of Attribute Operator
ems_c_attr_op_eq	TRUE if *attr_name* equal (==) to *attr_value*
ems_c_attr_op_gt	TRUE if *attr_name* greater than (>) *attr_value*
ems_c_attr_op_lt	TRUE if *attr_name* less than (<) *attr_value*
ems_c_attr_op_ge	TRUE if *attr_name* greater than or equal (>=) to *attr_value*
ems_c_attr_op_le	TRUE if *attr_name* greater than or equal (<=) to *attr_value*
ems_c_attr_op_ne	TRUE if *attr_name* not equal (<>) to *attr_value*
ems_c_attr_op_bitand	TRUE if *attr_name* bitwise anded with *attr_value* is greater than 0
ems_c_attr_op_substr	TRUE if *attr_name* contains the string value specified by *attr_value*

The actual values of the operators are:

```
#define ems_c_attr_op_eq (0)
#define ems_c_attr_op_gt (1)
#define ems_c_attr_op_lt (2)
#define ems_c_attr_op_ge (3)
#define ems_c_attr_op_le (4)
#define ems_c_attr_op_ne (5)
#define ems_c_attr_op_bitand (6)
#define ems_c_attr_op_substr (7)
```

Event Filter Expression

The event filter expression structure contains the elements of an event filter expression that is used to build an event filter. Event filter expressions contain an attribute name, operator, value triplet (*attr_name*, *attr_operator*, *attr_value*) which defines a boolean filter expression.

```
typedef struct ems_filter_exp_s_t {
    ems_string_t        attr_name;
    ems_attr_op_t       attr_operator;
    ems_attr_value_t    attr_value;
} ems_filter_exp_t;
```

Event Filter Expression List

An event filter expression list groups a list of filter expressions together in a list to form an **ande**d filter expression used in defining an event filter.

```
typedef struct ems_filter_exp_list_s_t {
    ems_long_int        size;
    ems_filter_exp_t    filter_exps[1];
} ems_filter_exp_list_t;
```

Event Filter

An event filter specifies a series of event filter expressions that will be **and**ed together to perform a filter operation. The event filter contains a name (*filter_name*) and a list of filter expressions (*event_exp_list*).

Filters with event type of generic, can only have filter expressions with header attribute names in them (See Table 3-2: Event Header Attributes).

```
typedef struct ems_filter_s_t {
    ems_string_t          filter_name;
```

```
        ems_event_type_t        type;
        ems_filter_exp_list_t   filter_exp_list;
} ems_filter_t;
```

Event Filter Name List

An event filter list contains a list of *size event_fitler_names*;

```
typedef struct ems_filtername_list_s_t {
        ems_long_int            size;
        ems_string_t            filter_names[1];
} ems_filtername_list_t;
```

Event Filter List

The event filter list structure contains a list of *size filters*.

```
typedef ems_filter_t *ems_filter_ptr_t;
typedef struct ems_filter_list_s_t {
        ems_long_int        size;
        ems_filter_ptr_t    filter[1];
} ems_filter_list_t;
```

3.6 EMS Consumer Data Structures

Consumer

The consumer data structure defines an ems consumer. Each consumer has a *name*, a *hostname* where the consumer is running, and a *uuid* unique to that consumer.

```
typedef struct ems_consumer_s_t {
        ems_string_t        name;
        ems_netname_t       *hostname;
        ems_uuid_t          uuid;
} ems_consumer_t;
```

Consumer List

The consumer list structure contains a list of *size consumer* entries.

```
typedef struct ems_consumer_list_s_t {
        ems_long_int        size;
```

```
    ems_consumer_t      consumer[1];
} ems_consumer_list_t;
```

3.7 EMS Supplier Data Structures

Supplier

The supplier data structure defines an ems supplier. Each supplier has a *name*, a *hostname* where the supplier is running, and a *uuid* unique to that supplier.

```
typedef struct ems_supplier_s_t {
    ems_string_t        name;
    ems_netname_t       *hostname;
    ems_uuid_t          uuid;
} ems_supplier_t;
```

Supplier List

The supplier list structure contains a list of *size supplier* entries.

```
typedef struct ems_supplier_list_s_t {
    ems_long_int        size;
    ems_supplier_t      supplier[1];
} ems_supplier_list_t;
```

3.8 EMS Attribute and Event list

Attribute List

The attribute list data structure defines a list of attributes associated with an event service,. An attribute list consists of a *size attr* entries that each represent an event service attribute. Event attributes are implementation dependent.

```
typedef struct ems_attrlist_s_t {
    ems_long_int        size;
    ems_attribute_t     attr[1];
} ems_attrlist_t;
```

Event List

The event list data structure contains a list of events. It is used to return the list of undelivered events. An event list consists of *size event* entries where each event entry is a pointer to an event.

```
typedef ems_event_t          *ems_event_ptr_t;
typedef struct ems_event_list_s_t {
        ems_long_int         size;
        ems_event_ptr_t      event[1];
} ems_event_list_t;
```

3.9 EMS Event Service Handle

An ems_handle represents a connection to an Event Service on a particular host. When calling EMS routines, one of the EMS registration routines is called to initialize the handle, and it is used in all subsequent calls to a routines that want to affect the Event Service on that host.

Event Service Handle

ems_handle_t is a pointer to an opaque data structure which contains information used to allow users of EMS to connect to the Event Service. The actual contents of the data structure are implementation dependant.

```
typedef void *ems_handle_t;
```

3.10 EMS Status Codes

All EMS routines return status codes which contain values that indicate whether the call to that routine was successful or not. Rather than list the specific status codes for each routine, here is a list of status codes and their meanings.

ems_s_status_ok	Success
ems_s_event_type_not_found	The specified event type was not found.

ems_s_invalid_handle	The handle parameter is not valid.
ems_s_invalid_name	The name parameter contains invalid characters.
ems_s_insufficient_permission	Caller does not have sufficient permission to perform operation.
ems_s_no_memory	An EMS handle cannot be allocated.
ems_s_unsupported_nameservice	unsupported nameservice on host name.
ems_s_filter_exits	The given filter name already exists.
ems_s_invalid_filter	The input parameters specifies an invalid filter.
ems_s_unknown_supplier	Tried to unregister a supplier that was not registered.
ems_s_consumer_already_started	Consumer already started.
ems_s_consumer_not_started	Consumer not started.

CHAPTER 4

Registration Interface

The Registration interface allows registration with EMS for applications that are not suppliers or consumers.

4.1 Registration Interface

Register with EMS

```
#include <xems.h>
void ems_register(
        ems_netname_t *      hostname,
        ems_handle_t *       handle,
        ems_error_t *        status);
```

DESCRIPTION:

This routine registers with EMS by obtaining an EMS handle. The EMS handle is then used to on future calls to the Event Service.

PARAMETERS:

Input

hostname	is the name of the host machine where an Event Service is running. If the hostname is NULL, then the local host is assumed

Output

handle	returns an EMS handle to use for future calls to EMS routines.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** **ems_s_unsupported_nameservice**

Unregister with EMS

```
#include <xems.h>
void ems_unregister(
        ems_handle_t *      handle,
        ems_error_t *       status);
```

DESCRIPTION:

This routine unregisters an EMS handle with EMS. The resources held by the ems handle are freed, and *handle* is assigned NULL. The handle can be one obtained by *ems_register*, or *ems_consumer_register*.

PARAMETERS:

Input

handle	the EMS handle to unregister.

Output

handle	assigns NULL to handle.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_handle

CHAPTER 5 Event Type Interface

The Event Type interface provides support for manipulating the Event Type Database.

5.1 Event Type Interface

Add an Event Type

```
#include <xems.h>
void ems_event_type_add(
        ems_handle_t             handle,
        ems_event_schema_t *     schema,
        ems_error_t *            status);
```

DESCRIPTION:

This routine is used by an event supplier to add new event types to the EMS event type Database. A supplier can add a new event type, then start producing that event type by transmitting events to EMS.

PARAMETERS:

Input

handle	a handle returned from a call to any *ems_register* call.
schema	is an EMS event type schema which describes the format of an event type.

Output

status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_handle

ems_s_event_type_exits

ems_s_insufficient_permission

ems_s_invalid_event_type

Delete an Event Type

```
#include <xems.h>
void ems_event_type_delete(
        ems_handle_t          handle,
        ems_string_t          type_name,
        ems_event_type_t *    type,
        ems_error_t *         status);
```

DESCRIPTION:

This routine is used by an event supplier to delete an event types in the EMS event type Database.

PARAMETERS:

Input

handle	a handle returned from a call to *ems_register*.
type_name	is the name of an EMS event type.
type	event type id of the EMS event type to delete.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_event_type_not_found

ems_s_invalid_handle

ems_s_invalid_name

ems_s_insufficient_permission

Get an Event Type

```
#include <xems.h>
void ems_event_type_get(
        ems_handle_t              handle,
        char *                    type_name,
        ems_event_type_t *        type,
        ems_event_schema_t **     schema,
        ems_error_t *             status);
```

DESCRIPTION:

This routine is used by an event supplier to get an event types from the EMS event type Database.

PARAMETERS:

Input

handle	a handle returned from a call to *ems_register.*
type_name	is the name of an EMS event type to get.
type	event type id of the EMS event type to get.

Output

schema	event type id of the EMS event type to get.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_event_type_not_found

ems_s_invalid_handle

ems_s_invalid_name

ems_s_insufficient_permission

Get Event Types List

```
#include <xems.h>
void ems_event_type_get_list(
        ems_handle_t                handle,
        ems_event_type_list_t **    type_list,
        ems_error_t *               status);
```

DESCRIPTION:

This routine is used by EMS event consumers to find out what event types are available to register for. The consumer can then set up filters for attributes in one of the available event types.

PARAMETERS:

Input

handle — should be the handle returned from a call to *ems_consumer_register* call.

Output

type_list — returns the list of available event types.

status — returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_handle .

ems_s_no_type_list

ems_s_invalid_name

ems_s_insufficient_permission

Free Event Types List

```
#include <xems.h>
void ems_event_type_free_list(
        ems_event_type_list_t **   type_list,
        ems_error_t *              status);
```

DESCRIPTION:

This routine is used by callers of *ems_get_event_types* to free the storage used by an event type list.

PARAMETERS:

Input

type_list	an event type list as returned by *ems_event_type_get_list*. *Type_list* will be set to NULL by this routine.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

CHAPTER 6

Event Filter Interface

Event filters can be created using the Event FIlter Interface.

6.1 Event Filter Construction Interface

EMS provides several routines to construct event filters. These are routines to add, delete and update an event filter.

Add an Event Filter

```
#include <xems.h>
void ems_filter_add(
        ems_handle_t              handle,
        ems_string_t              filter_name,
        ems_event_type_t          type,
        ems_filter_exp_list_t *   exp_list,
        ems_error_t *             status);
```

DESCRIPTION:

This routine is used to add a new event filter to the EMS Event Filter Database.

There is currently no mechanism for indicating all events.

PARAMETERS:

Input

handle	a handle returned from a call to *ems_consumer_register* call.
filter_name	specifies the event filter name for this event filter. This name can be used to add the event filter to a consumers event filter group.
type	specifies the event type that this filter will be applies against.
exp_list	a list of filter expressions which are part of the event filter *filter_name*.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** **ems_s_invalid_handle** **ems_s_insufficient_permission** **ems_s_filter_exits** **ems_s_invalid_filter** **ems_s_invalid_name**

Append to an Event Filter

```
#include <xems.h>
void ems_filter_append(
        ems_handle_t              handle,
        ems_string_t              filter_name,
        ems_filter_exp_list_t *   exp_list,
        ems_error_t *             status);
```

DESCRIPTION:

This routine is used to add filter expressions to an event filter. The filter expressions are added to the end of the current list of filter expressions in the event filter.

PARAMETERS:

Input

handle	should be the handle returned from a call to *ems_consumer_register* call.
filter_name	specifies the name of the event filter to add the filter expressions to.
exp_list	a list of filter expressions which will be added to the end of event filter *filter_name*.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_name

ems_s_invalid_handle

ems_s_invalid_name

ems_s_filter_not_found

ems_s_insufficient_permission

Get an Event Filter

```
#include <xems.h>
void ems_filter_get(
      ems_handle_t               handle,
      ems_string_t               filter_name,
      ems_event_type_t *         type,
      ems_filter_exp_list_t **   filter_exprs,
      ems_error_t *              status);
```

DESCRIPTION:

This routine is used to get the filter expressions in an event filter.

PARAMETERS:

Input

handle	should be the handle returned from a call to *ems_consumer_register* call.
filter_name	specifies the name of the event filter to get.

Output

type	the event type of the filter.
exp_list	the list of filter expressions which are part of event filter *filter_name*.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_name

ems_s_invalid_handle

ems_s_filter_not_found

ems_s_insufficient_permission

Delete an Event Filter

```
#include <xems.h>
void ems_filter_delete(
        ems_handle_t        handle,
        ems_string_t        filter_name,
        ems_error_t *       status);
```

DESCRIPTION:

This routine is used to delete an event filter from the Event Filter Database. The name *filter_name* cannot appear in any consumers event filter group when this routine is called.

PARAMETERS:

Input

handle	should be the handle returned from a call to *ems_consumer_register* call.
filter_name	specifies the name of the event filter to delete.

Output

status — Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_filter_not_found

ems_s_filter_in_use

ems_s_insufficient_permission

ems_s_invalid_name

ems_s_invalid_handle

List Event Filter Names

```
#include <xems.h>
void ems_filter_get_namelist(
    ems_handle_t                handle,
    ems_filtername_list_t **    name_list,
    ems_error_t *               status);
```

DESCRIPTION:

This routine is used to get a list of the names of the event filters in the Event Filter Database.

PARAMETERS:

Input

handle	should be the handle returned from a call to *ems_consumer_register* call.

Output

name_list	will contain a list of all the event filter names in the Event Filter Database. The routine *ems_event_filter_get* can be used to find out the contents of each event filter.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_insufficient_permission

ems_s_invalid_handle

ems_s_empty_filter_db

Free Event Filter Names

```
#include <xems.h>
void ems_filter_free_namelist(
      ems_filtername_list_t **      name_list,
      ems_error_t *                 status);
```

DESCRIPTION:

This routine is used to free a list of the names of returned by the *ems_filter_get_namelist* routine

PARAMETERS:

Input

name_list	List of filter names to free.

Output

name_list	Sets to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok**

Get Event Filter List

```
#include <xems.h>
void ems_filter_get_list(
    ems_handle_t           handle,
    ems_filter_list_t **   filter_list,
    ems_error_t *          status);
```

DESCRIPTION:

This routine is used to get a list of the event filters in the Event Filter Database.

PARAMETERS:

Input

handle	a handle returned from a call to *ems_consumer_register* call.

Output

filter_list	will contain a list of all the event filters in the Event Filter Database. This list should be freed using *ems_filter_free_list*.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_insufficient_permission

ems_s_invalid_handle

ems_s_empty_filter_db

Free Event Filter List

```
#include <xems.h>
void ems_filter_free_list(
      ems_filter_list_t **      filter_list,
      ems_error_t *             status);
```

DESCRIPTION:

This routine is used by callers of *ems_get_event_filter_database* to free the storage used by an Event Filter Database (*ems_filter_list_t*) structure.

PARAMETERS:

Input

filter_list	a list of event filters that make up the Event Filter Database as returned by the routine *ems_filter_get_list*.

Output

filter_list	will be set to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok**

CHAPTER 7

Consumer Interface

The EMS event consumer interface consists of two parts. One part is used by the consumer to set itself up as a consumer, and the other is used to register with EMS.

All event consumers have to make calls to the EMS event consumer setup routines before receiving EMS events. These routines perform required setup, and are EMS implementation dependent.

The event consumer interface allows event consumers to register and unregister with EMS. Once registered, consumers can add and delete event filters define what events they are interested in. When EMS receives events from event suppliers, the event will be filtered using the event filter, and only the matching events will be forwarded on to the interested consumers.

Consumer Start

```
#include <xems.h>
void ems_consumer_start(
        ems_string_t        consumer,
        ems_ulong_int       flags,
        ems_string_t        reference,
        ems_error_t *       status);
```

DESCRIPTION:

This routine should be called at the beginning of each event consumer before making any register calls. It will create an *ems_uuid_t* to uniquely identify this event consumer and perform any local consumer initialization required.

PARAMETERS:

Input

consumer	specifies the consumer name.
flags	reserved for future use.
reference	reference for this conversation.

Output

reference	reference for this conversation.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible ems status codes are:

ems_s_status_ok

ems_s_no_memory

ems_s_no_memory

ems_s_consumer_already_started

Consumer Stop

```
#include <xems.h>
void ems_consumer_stop(
        ems_error_t *        status);
```

DESCRIPTION:

This routine should be called at the end of each event consumer. It will perform any consumer cleanup required.

PARAMETERS:

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.
	The possible EMS status codes are:
	ems_s_status_ok
	ems_s_consumer_not_started

Consumer Event Handler Register

```
#include <xems.h>
void ems_consumer_handler_register(
        ems_handler_t       hfunc,
        ems_error_t *       status);
```

DESCRIPTION:

This routine declares the event consumer's event handler. The event consumer developer is responsible for providing the handler to process events.

PARAMETERS:

Input

hfunc specifies the name of the event handler function. The handler's signature should be:

```
typedef void   (*ems_handler_t) ( ems_event_t  *event,
                                  ems_error_t *status);
```

Output

status Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

ems_s_no_memory Error allocating memory

Push Consumer Register

```
#include <xems.h>
void ems_push_consumer_register(
        ems_netname_t *             hostname,
        ems_filtername_list_t *     filter_group,
        ems_handle_t *              handle,
        ems_error_t *               status);
```

DESCRIPTION:

This routine is used by EMS event consumers to register as a push consumer with EMS. This routine should be called once for each host that this consumer wants to receive events from.

PARAMETERS:

Input

hostname	is the name of the host machine where the Event Service is running. If the hostname is NULL, then the local host is assumed.
filter_group	is a list of event filter names which will define this consumers initial event filter group. If *filter_group* is empty, no filter group is specified, and EMS will not forward any events to this consumers until the consumer makes a call to *ems_add_event_to_group*.

Output

handle	returns an EMS handle which can be used on subsequent calls to EMS routines.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory

ems_s_already_registered Consumer with this name already registered.

Pull Consumer Register

```
#include <xems.h>
void ems_pull_consumer_register(
      ems_netname_t *          hostname,
      ems_filtername_list_t *  filter_group,
      ems_handle_t *           handle,
      ems_error_t *            status);
```

DESCRIPTION:

This routine is used by EMS event consumers to register as a pull consumer with EMS. This routine should be called once for each host that this consumer wants to receive events from.

PARAMETERS:

Input

hostname	is the name of the host machine where the Event Service is running. If the hostname is NULL, then the local host is assumed.
filter_group	is a list of event filter names which will define this consumers initial event filter group. If *filter_group* is empty, no filter group is specified, and EMS will not forward any events to this consumers until the consumer makes a call to *ems_add_event_to_group*.

Output

handle	returns an EMS handle which can be used on subsequent calls to EMS routines.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory

ems_s_already_registered Consumer with this name already registered.

Consumer Unregister

```
#include <xems.h>
void ems_consumer_unregister(
    ems_handle_t *        handle,
    ems_error_t *         status);
```

DESCRIPTION:

This routine is used by EMS event consumers to unregister with EMS. This routine should be called once for each call to *ems_push_consumer_registe or ems_pull_consumer_register*. The event consumer should call this routine before calling the *ems_consumer_stop* routine.

PARAMETERS:

Input

handle	a handle returned from a call to one of the consumer register routines.

Output

handle	This routine will free up memory used by handle, and set *handle* to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

ems_s_unknown_consumer Tried to unregister a consumer that was not registered.

Add Event Filter to Group

```
#include <xems.h>
void ems_add_filter_to_group(
      ems_handle_t               handle,
      ems_filtername_list_t *    event_filters,
      ems_error_t *              status);
```

DESCRIPTION:

This routine is used by EMS event consumers to add event filter names to a consumers event filter group. This routine can be called multiple times for each consumer.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_push_consumer_register* or *ems_pull_consumer_register.*
event_filters	contains a list of one or more event filter names to add to this consumers event filter group. consumers can use the names of new event filters after building them with the *ems_filter_add* routine, or existing filters which can be obtained by using the *ems_filter_get_namelist* routine.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** Success

Delete Event Filter From Group

```
#include <xems.h>
void ems_delete_filter_from_group(
    ems_handle_t               handle,
    ems_filtername_list_t *    filter_name,
    ems_error_t *              status);
```

DESCRIPTION:

This routine is used by EMS event consumers to delete event filter names from consumer event filter groups.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_push_consumer_register* or *ems_pull_consumer_register*.
filter_name	specifies the event filter name(s) to delete from the consumers event filter group.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

Get Filter Group

```
#include <xems.h>
void ems_get_filter_group(
        ems_handle_t                handle,
        ems_filtername_list_t **    filter_group,
        ems_error_t *               status);
```

DESCRIPTION:

This routine returns a list of event filter names that comprise the consumers event filter group.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_push_consumer_register* or *ems_pull_consumer_register*.

Output

filter_group	will contain the list of event filter names which are in the consumers event filter group. It is up to the requesting consumer to free the storage allocated for *filter_group*.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** Success

The EMS event consumer interface consists of two parts. One part is used by the consumer to set itself up as a consumer, and the other is used to register with EMS.

All event consumers have to make calls to the EMS event consumer setup routines before receiving EMS events. These routines perform required setup, and are EMS implementation dependent.

The event consumer interface allows event consumers to register and unregister with EMS. Once registered, consumers can add and delete event filters define what events they are interested in.   When EMS receives events from event suppliers, the event will be filtered using the event filter, and only the matching events will be forwarded on to the interested consumers.

CHAPTER 8 Supplier Interface

The "pull-supplier" interface has not been entered. There is no way to supply hints, e.g., polling interval, number of consumers for a given event type, etc., to a supplier.

Push Supplier Register

```
#include <xems.h>
 void ems_push_supplier_register(
      ems_netname_t *        hostname,
      ems_handle_t *         handle,
      ems_error_t *          status);
```

DESCRIPTION:

This routine is used by EMS event suppliers to register with EMS. This routine should be called once for each host that this supplier wants to push events to.

PARAMETERS:

Input

hostname is the name of the host machine where the Event Service is running. If the hostname is NULL, then the local host is assumed.

Output

handle	returns an EMS handle which can be used on subsequent calls to EMS routines.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_no_memory

ems_s_already_registered

ems_s_insufficient_permission

ems_s_unsupported_nameservice

Pull Supplier Register

```
#include <xems.h>
void ems_pull_supplier_register(
      ems_netname_t *        hostname,
      ems_handle_t *          handle,
      ems_error_t *           status);
```

DESCRIPTION:

This routine is used by EMS event suppliers to register with EMS. This routine should be called once for each host that this supplier wants to be a pull supplier for.

PARAMETERS:

Input

hostname	is the name of the host machine where the Event Service is running. If the hostname is NULL, then the local host is assumed.

Output

handle	returns an EMS handle which can be used on subsequent calls to EMS routines.
status	returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_no_memory

ems_s_already_registered

ems_s_unsupported_nameservice

Supplier Unregister

```
#include <xems.h>
void ems_supplier_unregister(
        ems_handle_t *        handle,
        ems_error_t *         status);
```

DESCRIPTION:

This routine is used by EMS event suppliers to unregister with EMS. This routine should be called once for each call to *ems_push_supplier_register* or *ems_pull_supplier_register*.

PARAMETERS:

Input

handle	a valid supplier handle returned from a call to a supplier register routine. This routine will free up memory used by handle, and set *handle* to NULL.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** **ems_s_unknown_supplier** **ems_s_invalid_handle**

Supplier Send

```
#include <xems.h>
void ems_supplier_send(
        ems_handle_t        handle,
        ems_event_t *       event,
        ems_error_t *       status);
```

DESCRIPTION:

This routine is called by event suppliers to send events to the Event Service.

PARAMETERS:

Input

handle	should be the handle returned from a call to the *ems_register* call.
event	contains the actual event data. For the content of the event messages, see the "Data Structures" section.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok

ems_s_invalid_handle

ems_s_no_memory

ems_s_insufficient_permission

CHAPTER 9 Administration Interface

The EMS Management interface provides a means to manage various aspects of the EMS. Using this interface applications can manage event servers, event consumers, event filters, and undelivered events in the EMS event log.

Additional management interfaces have been added, but are not yet reflected in this chapter.

List Event Service Hosts

```
#include <xems.h>
void ems_mgmt_list_ems(
        ems_string_t **          host_list,
        ems_error_t *            status);
```

DESCRIPTION:

List hosts running the Event Service. These hosts can be used with calls to the *ems_register*, *ems_push_supplier_register*, *ems_pull_supplier_register*, *ems_push_consumer_register* and *ems_pull_consumer_register* routines.

PARAMETERS:

Output

host_list	contains the list of hosts running the Event Service. Use *ems_mgmt_free_ems* to free this list.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not. The possible status codes are: **ems_s_status_ok** Success

Free Event Service Host List

```
#include <xems.h>
void ems_mgmt_free_ems(
        ems_string_t **        host_list,
        ems_error_t *          status);
```

DESCRIPTION:

Free *host_list* structure obtained from a call to *ems_mgmt_list_ems*.

PARAMETERS:

Input

host_list	list of hosts obtained from *ems_mgmt_list_ems* to free.

Output

host_list	set to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

List Event Service Attributes

```
#include <xems.h>
void ems_mgmt_list_attributes(
      ems_handle_t          h,
      ems_attrlist_t **     list,
      ems_error_t *         status);
```

DESCRIPTION:

List Event Service attributes. These attributes are implementation defined

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_register* routine.

Output

list	contains the list of Event Service attributes
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Free Event Service Attributes

```
#include <xems.h>
void ems_mgmt_free_attributes(
        ems_attrlist_t **        list,
        ems_error_t *            status);
```

DESCRIPTION:

Set an Event Service attribute. These attributes are implementation defined.

PARAMETERS:

Input

list	contains the list of attributes to free.

Output

list	set to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

Management List Consumers

```
#include <xems.h>
void ems_mgmt_list_consumers(
        ems_handle_t            handle,
        ems_consumer_list_t **  list,
        ems_error_t *           status);
```

DESCRIPTION:

List consumers registered with EMS.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_register* routine.

Output

list	contains the list of consumers.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory
ems_s_no_consumers	No consumers registered

Management Free Consumers List

```
#include <xems.h>
void ems_mgmt_free_consumers(
        ems_consumer_list_t **     list,
        ems_error_t *              status);
```

DESCRIPTION:

Free the storage used by an *ems_consumer_list_t* structure obtained by a call to *ems_mgmt_list_consumers*.

PARAMETERS:

Input

list consumer list to free.

Output

list set to NULL.

status Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

Management Delete Supplier

```
#include <xems.h>
void ems_mgmt_delete_consumer(
        ems_handle_t        handle,
        ems_string_t        consumer,
        ems_uuid_t *        uuid,
        ems_error_t *       status);
```

DESCRIPTION:

Deletes a consumer from the EMS consumer database. After this call, the specified consumer will not receive any events unless it reregisters with the Event Service.

PARAMETERS:

Input

handle — must contain a valid consumer handle obtained from *ems_register* routine.

consumer — specifies the consumer name to clear. This name is the name returned in the *ems_consumer_list_t* data structure after calling *ems_mgmt_list_consumers* or the name used on the *ems_consumer_start* routine.

uuid — specifies the consumer uuid which uniquely identifies the consumer to clear. If this parameter is NULL, then only one consumer can exist with the name *consumer.*

Output

status — Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Management Delete Event Filter From Group

```
#include <xems.h>
void ems_mgmt_delete_filter_from_group(
        ems_handle_t                 handle,
        ems_string_t                 consumer,
        ems_uuid_t *                 uuid,
        ems_filtername_list_t *      filter_name,
        ems_error_t *                status);
```

DESCRIPTION:

This routine deletes a specified event filter name(s) from a consumers event filter group.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_register* routine.
consumer	specifies the consumer whose event filter group is getting updated.
uuid	specifies the consumer uuid which uniquely identifies the consumer to clear. If NULL is specified, then only one consumer can exist with the name *consumer*.
filter_name	name(s) of the filters to delete from the consumer's filter group.

Output

status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

Management Add Event Filter to Group

```
#include <xems.h>
void ems_mgmt_add_filter_to_group(
        ems_handle_t                handle,
        ems_string_t                consumer,
        ems_uuid_t *                uuid,
        ems_filtername_list_t *     filter_name,
        ems_error_t *               status);
```

DESCRIPTION:

This routine adds event filter names to a consumers event filter group.

PARAMETERS:

Input

handle	must contain a valid handle obtained from *ems_register* routine.
consumer	specifies the consumer whose event filter group is getting updated.
uuid	specifies the consumer uuid which uniquely identifies the consumer to clear. If NULL is specified, then only one consumer can exist with the name *consumer*.
filter_name	specifies the list of event filter names to add.

Output

status Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok Success

Management Get Filter Group

```
#include <xems.h>
void ems_mgmt_get_filter_group(
```

```
ems_handle_t              handle,
ems_string_t              consumer,
ems_uuid_t *              uuid,
ems_filtername_list_t **  filter_group,
ems_error_t *             status);
```

DESCRIPTION:

This routine returns a list of event filter names in a consumers event filter group.

PARAMETERS:

Input

handle	must contain a valid consumer handle obtained from *ems_register* routine.
consumer	specifies which consumers event filter group to return. The consumer name is the name given to the *ems_start_consumer* routine, or the name returned in the *ems_consumer_list_t* data structure from the routine *ems_mgmt_list_consumers*.
uuid	specifies the consumer uuid which uniquely identifies the consumer to clear. If this parameter is NULL, then only one consumer can exist with the name *consumer.*

Output

filter_group	contains the list of event filter names in the specified consumers event filter group.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Management List Suppliers

```
#include <xems.h>
void ems_mgmt_list_suppliers(
        ems_handle_t            handle,
        ems_supplier_list_t **  list,
        ems_error_t *           status);
```

DESCRIPTION:

List suppliers registered with EMS.

PARAMETERS:

Input

handle	must contain a valid handle obtained from *ems_register* routine.

Output

list	contains the list of suppliers.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory
ems_s_no_suppliers	No suppliers registered

Management Free Suppliers List

```
#include <xems.h>
void ems_mgmt_free_suppliers(
        ems_supplier_list_t **    list,
        ems_error_t *             status);
```

DESCRIPTION:

Free the storage used by an *ems_supplier_list_t* structure obtained by a call to *ems_mgmt_list_suppliers*.

PARAMETERS:

Input

list	supplier list to free.

Output

list	set to NULL.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Management Delete Supplier

```
#include <xems.h>
void ems_mgmt_delete_supplier(
        ems_handle_t        handle,
        ems_string_t        supplier,
        ems_uuid_t *        uuid,
        ems_error_t *       status);
```

DESCRIPTION:

Clear all information stored in EMS about the specified supplier.

PARAMETERS:

Input

handle	must contain a valid supplier handle obtained from *ems_register* routine.
supplier	specifies the supplier name to clear. This name is the name returned in the *ems_supplier_list_t* data structure after calling *ems_mgmt_list_suppliers*.
uuid	specifies the supplier uuid which uniquely identifies the supplier to clear. If this parameter is NULL, then only one supplier can exist with the name *supplier*.

Output

status

Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Management Get Undelivered Events

```
#include <xems.h>
void ems_mgmt_get_undelivered_events(
            ems_handle_t            handle,
            ems_event_type_t        type,
            ems_event_list_t **     list,
            ems_error_t *           status);
```

DESCRIPTION:

Get a list of events that have not been delivered to interested consumers.

PARAMETERS:

Input

handle	must contain a valid handle obtained from *ems_register* routine.
type	the event type to control the number of returned events.

Output

list	contains the list of undelivered events.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory
ems_s_no_events	No undelivered events.

Management Free Undelivered Events

```
#include <xems.h>
void ems_mgmt_free_undelivered_events(
        ems_event_list_t **    event,      /* undelivered events    */
        ems_error_t *          status);    /* event get status      */
```

DESCRIPTION:

Get a list of events that have not been delivered to interested consumers.

PARAMETERS:

Input

list	contains the list of undelivered events to free.

Output

list	set to free.
status	Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success

Management Delete Undelivered Event

```
#include <xems.h>
void ems_mgmt_delete_undelivered_event(
        ems_handle_t        handle,
        ems_eventid_t *     event_id,
        ems_error_t *       status);
```

DESCRIPTION:

Delete an undelivered event from the EMS Event log.

PARAMETERS:

Input

handle	must contain a valid handle obtained from *ems_register* routine.
event_id	the event id of the event to delete.

Output

status

Returns the status code from this routine which indicates whether the routine completed successfully or, if not, why not.

The possible status codes are:

ems_s_status_ok	Success
ems_s_no_memory	Error allocating memory
ems_s_no_events	No undelivered events.

CHAPTER 10

Command Line Interface

Besides the set of APIs, the Event Service provides a command line interface to assist wrapping legacy applications and shell scripts as event suppliers. The importance of the CLI will certainly decrease with the increasing acceptance of the ES and the usage of its APIs by integrating applications. Due to the nature of UNIX commands, the performance of CLI calls will suffer compared to API calls.

The supplier command line interface allows immediate integration of event generating applications into the Event Service.

The Event Service Management Command Line interface provides a command line interface that can be used by a system administrator to manage EMS.

10.1 Event Supplier Interface

The Event Supplier Interface provides commands to send (or push) an event to the Event Channel.

Supplier Send

ems_supplier_send <type> [origin_netname=<service:addr> [origin_desc=<descname>]] [pid=<pid>] [uid=<uid>] [gid=<gid>] [severity=<sev>] [<attr_name>=<attr_value> [...]]

Description:

This command is called by event suppliers to create and send an event to the Event Service. An event of the specified type is created and the attributes are set according to the name=value pairs. Unspecified attributes are set to default values.

Parameters:

type	specifies the name of the event type in the Event Type Schema database.
origin_netname	specifies the originating node in the format <service>:<addr>, where service is one of[1] - other - dns - dce - x500 - nis - sna Default value for this parameter is the local node.[2]
origin_desc	a string provided as a description of the originator; defaults to an empty string.
pid	the process ID or the originating application; defaults to -1.
uid, gid	user and group ID of the originating application; defaults to the uid and gid of the calling process.

1. the keywords that are allowed for *service* match the values of XES_NameService
2. NOTE: It needs to be clarified whether both service and address can always be determined on a node.

severity severity of the event; one of[1]
- info
- emergency
- alert
- critical
- warning
- notice
- debug

attr_name specifies the name of the attribute that is set

attr_value value that is assigned to the attribute *attr_name*; the value is converted to the respective data type that is defined for *attr_name* in the Event Type Schema of event type *type*.

10.2 Event Service Management Command Line Interface

The Event Service Management command line interface allows for management of all EMS Objects. The interface has been defined as a set of objects, with operations on those objects. The four objects defined are:

- ems
- emsconsumer
- emsfilter
- emsevent-type
- emslog

The management objects and command line syntax are defined below:

1. the keywords that are allowed for *severity* match the values of XES_Severity

Event Service Object

ems catalog

ems show [-host hostname]

ems modify [-host hostname] [-add attr_name:attr_value -change attr_name:attr_value |
-delete attr_name]

Description:

This command is used to manage hosts running the EMS Event Service. The ems object represents an Event Service on a host.

Operations:

catalog returns all hosts in the current domain providing Event Services (e.g. all DCE hosts running the event service daemon)

show returns all the Event Service attributes for the Event Service on the given host.

modify allow for the modification of the attributes of an Event Service on a given host. Attributes can be added, deleted, or their values can be changed.

Parameters:

-host hostname specifies the name of the host running the Event service. If this parameter is not present, then the local host is assumed.

-add attr_name:attr_value add the specified attribute to the Event Service attribute list. *attr_name* specifies the name of the attribute that is set. *attr_value*specifies the value that is assigned to the attribute *attr_name*;

-change attr_name:attr_value change the specified attribute in the Event Service attribute list. *attr_name* specifies the name of the attribute to change. *attr_value* specifies the new value that is assigned to the attribute *attr_name*;

-delete attr_name delete the specified attribute in the Event Service attribute list. *attr_name* specifies the name of the attribute to delete.

Consumer Object

emsconsumer catalog [-host hostname]

emsconsumer show <consumer_name> [-uuid consumer_uuid][-host hostname]

emsconsumer delete <consumer_name> [-uuid consumer_uuid][-host hostname]

emsconsumer modify <consumer_name> [-uuid consumer_uuid][-host hostname]
-add filter_name | -delete filter_name

Description:

This command is used to manage registered consumers in the EMS Consumer Database. The emsconsumer object represents an EMS consumer.

Operations:

catalog	returns a list registered consumers in the Consumer Database.
show	shows the filter group for consumer *consumer_name*.
delete	delete consumer *consumer_name* from the Consumer Database.
modify	modify consumer *consumer_name*'s filter group.

Parameters:

-host hostname	specifies the name of the host running the Event Service. If this parameter is not present, then the local host is assumed.
consumer_name	specifies the name of the consumer to perform the operation on.
-uuid consumer_uuid	specifies the consumers unique identifier to uniquely identify *consumer_name*.
-add filter_name	add *filter_name* to the specified consumers filter group.
--delete filter_name	delete *filter_name* from *consumer_name*'s filter group.

Filter Object

emsfilter catalog [-host hostname]

emsfilter show <filter_name> [-host hostname]

emsfilter delete <filter_name>[-host hostname]

emsfilter modify <filter_name> [-host hostname] [-append filter_exp]

Description:

This command is used to manage filters in the EMS Filter Database. The emsfilter object represents an EMS filter.

Operations:

catalog	returns a list of filters in the Filter Database.
show	shows the contents of the event filter *filter_name*.
delete	delete the event filter *filter_name* from the Event Filter Database.
modify	modify the event filter *filter_name*.

Parameters:

-host hostname	specifies the name of the host running the Event Service. If this parameter is not present, then the local host is assumed.
filter_name	specifies the name of the filter to perform the operation on.
-append filter_exp	append *filter_exp* to *filter_name*'s filter expression list. This parameter is required for the **modify** operation.

Event Type Object

emsevent-type catalog [-host hostname]

emsevent-type show <event_type_name> [-uuid event_type_uuid][-host hostname]

emsevent-type add <event_type_name>[-uuid event_type_uuid] [-host hostname]
[[-attr attr_name:attr_fmt]..].

emsevent-type delete <event_type_name>[-uuid event_type_uuid][-host hostname]

Description:

This command is used to manage event types in the EMS Event Type Database. The **emsevent-type** object represents an EMS event type.

Operations:

catalog returns a list of event types in the Event Type Database.

show shows the contents of the event type *event_type_name*.

add add the event type *event_type_name* to the Event Type Database.

delete delete the event type *event_type_name* from the Event Type Database.

Parameters:

event_type_name specifies the name of the event type to perform the operation on.

-uuid event_type_uuid specifies the event type that uniquely identifies *event_type_name*. This parameter required for the **add** operation.

-host hostname specifies the name of the host running the Event Service. If this parameter is not present, then the local host is assumed.

-attr attr_name:attr_fmt specifies the an attribute in the event type schema. *attr_name* specifies the attribute name, and *attr_fmt* specifies the attribute format name as specified in the Data Structures section. Attributes defined in schema correspond one for one with the

attributes specified by the *-attr* parameters, in the order specified on this command. At least one -attr parameter is required for the **add** operation.

Event Log Object

emslog catalog [-host hostname]

emslog delete <event_uuid> [-host hostname]

Description:

This command is used to manage undelivered events in the EMS Event log. The emslog object represents an EMS event log.

Operations:

catalog	returns a list of undelivered events in the EMS Event log.
delete	delete the event specified by *event_uuid* from the Event log.

Parameters:

-host hostname	specifies the name of the host running the Event Service. If this parameter is not present, then the local host is assumed.
event_uuid	specifies the unique identifier for the undelivered event to delete.

Appendices

Appendix A

Reference Implementations

This appendix contains reference implementations of the EMS. Three implementations are addressed. This is not all of the possible implementations. It is simply meant to provide a grounding for implementors of the EMS. Each implementation supports the generic EMS data structures, APIs, and command level interfaces at the source level. Each may have unique libraries, macros, etc., permitting them to provide transport and environment specific operability. The reference implementations are for CORBA, DCE, and UMA.

Appendix A.1

CORBA Implementation

This appendix describes the OMG IDL interfaces mapping of the EMS generic APIs which were described earlier. The information provided here describes additional features and/or facilities provided by a CORBA implementation to operate with CORBA consumers and producers.

The following subsection lists the descriptions of the interfaces and their associated operations, followed by an IDL definition of the proposed interfaces.

A.1.1 Interface Descriptions

A.1.1.1 The FilterRepository Interface

Each notification manager will have one filter repository associated with it. As described in the general model, the filter repository contains registered filters. Consumers can specify a list of the names of any registered filters when registering with the notification manager. These filters are used by the notification manager to determine which events should be forwarded to which consumers.

The FilterRepository interface defines the methods that support the management of filters by a notification manager. The specific operations it supports are detailed below.

filter_add

The *filter_add* operation adds a new filter to the filter repository. The first parameter gives a name which is used to subsequently identify the filter being added among other filters in the repository. This name must be unique among all filters existing in the repository. The second parameter specifies the type of event that the filter is associated with. The content of this parameter can either be the unique identifier of an event type currently known to the target notification manager, or the string "generic", meaning the filter being added can only be applied to an event header. The third parameter contains the list of filter expressions which make up the new filter.

Each filter expression identifies an attribute, an operator, and a value. When a filter is applied to an event to determine whether or not the event should be forwarded to a particular consumer, each filter expression is applied to the event. When applying a filter expression to an event, the event is examined to see if it supports an attribute of the same name as the attribute name in the filter expression. If the event supports such an attribute, the event attribute's value is compared against the value field of the filter expression, using the comparison operator specified in the filter expression. If the result of the comparison is TRUE, the filter expression logically evaluates to TRUE. If the event does not contain an attribute of the same name as the attribute field of the filter expression, or if it does contain such an attribute and the result of the specified comparison is FALSE, the filter expression evaluates to FALSE. An event is only forwarded to a consumer with which there is an associated filter in the consumer/producer registry whose filter expressions all evaluate to TRUE when applied to the event.

Syntax

```
void filter_add (in string filter_name,
                in event_type_t type,
                in filter_exp_list_t exp_list)
            raises (ExFilterExists,
                ExInvalidFilter,
                ExInvalidType);
```

Exceptions

If a filter by the same name as the one being added already exists in the filter repository, the ExFilterExists exception is raised. If one of the input filter expressions indicates an operation that cannot be applied to a value of the type given (for example, the specified operation indicates the substring operation and the value given is of type “long”), the ExInvalidFilter expression is raised, with the “attr_name” field of the exception being set to the name of the first invalid attribute encountered in the filter expression. If the event type specified by second parameter is not known to the target notification manager, the ExInvalidType exception is raised.

filter_append

The *filter_append* operation adds a list of filters to an existing filter expression in the filter repository. The first input parameter should specify the name of a filter that already exists within the filter repository. The second input parameter contains a list of filter expressions of the type described above for the *filter_add* operation. The filter repository is updated to append the list of supplied filter expressions to the end of the current list of filter expressions associated with the given filter.

Syntax

```
void filter_append (in string filter_name,
                    in filter_exp_list_t exp_list)
            raises (ExFilterNotFound,
                    ExInvalidFilter);
```

Exceptions

If no filter of the name supplied by the first input parameter is found to exist within the filter repository, the **ExFilterNotFound** exception is raised. If one of the input filter expressions indicates an operation that cannot be applied to a value of the type given (for example, the specified operation indicates the substring operation and the value given is of type “long”), the **ExInvalidFilter** expression is raised, with the “attr_name” field of the exception being set to the name of the first invalid attribute encountered in the filter expression.

filter_get

The *filter_get* operation returns the filter expressions associated with a given filter in the filter repository. The first parameter specifies the name of the filter whose filter expressions should be returned. The second parameter is an output parameter that, upon successful completion of the operation, identifies the type of event to which the filter applies. A value of “generic” returned in this parameter implies that the filter can only be applied to event header attributes. The third parameter is an output parameter that, upon successful completion of this operation, will contain the list of filter expressions associated with the given filter.

Syntax

```
void filter_get (in string filter_name,
                 out event_type_t type,
                 out filter_exp_list_t exp_list)
             raises (ExFilterNotFound);
```

Exceptions

If no filter of the name supplied by the first parameter is found to exist within the filter repository, the **ExFilterNotFound** exception is raised.

filter_delete

The *filter_delete* operation removes an existing filter from the filter repository. The parameter gives the name of the filter to be removed. This operation will only succeed if there is no consumer in the consumer/producer registry that is registered to use the target filter.

Syntax

```
void filter_delete (in string filter_name)
             raises (ExFilterNotFound,
                 ExFilterInUse);
```

Exceptions

If no filter of the name supplied by the input parameter is found to exist within the filter repository, the **ExFilterNotFound** exception is raised. If there exists a consumer in the consumer/producer registry that is registered to use the input filter, the **ExFilterInUse** exception is raised, with the consumer field of the exception structure set to the object reference of the first such consumer encountered in the consumer/producer registry.

filter_get_namelist

The *filter_get_namelist* operation returns a list of the names of all filters that exist within the filter repository.

Syntax

```
void filter_get_namelist(
                 out filtername_list_t name_list)
```

```
            raises (ExNoFilterList);
```

Exceptions

If no filters currently exist in the filter repository, the **ExNoFilterList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

filter_get_list

The *filter_get_list* operation returns a list of all filters that currently exist in the filter repository. Each element in the returned list contains the name of a filter, the type of event the filter is intended to be applied to, and the list of filter expressions associated with the filter.

Syntax

```
void filter_get_list (out filter_list_t filter_list)
            raises (ExNoFilterList);
```

Exceptions

If no filters currently exist in the filter repository, the **ExNoFilterList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

A.1.1.2 The CPRegistry Interface

Each notification manager will have one consumer/producer registry associated with it. As described in the general model, the consumer/producer registry contains information about all consumers and producers registered with a notification manager, the schema information that describes each type of event that is available for consumers to register for, and for each registered consumer, the list of event types it is registered to receive along with the filters that will be applied to the events of the indicated types to determine which should actually be forwarded to the consumer.

The CPRegistry interface defines the methods that support the management of the information contained in consumer/producer registry associated with a given notification manager. The specific operations it supports are detailed below.

event_type_add

The *event_type_add* operation adds a new type of event to the event type database maintained by the consumer/producer registry. Effectively, this makes the new event type known to the notification manager associated with the target consumer/producer registry.

The input parameter contains the schema that describes a new event type. This structure includes a string which should uniquely identify the event type among all other event types currently existing in the consumer/producer registry, a (not necessarily unique) human readable name for the event type being added, and a sequence containing the attributes supported by the new event type. Each attribute in this sequence consists of a name for the attribute, along with an any value that indicates the attribute's type. Note that each event type automatically supports the attributes defined for an event header, so the list of attributes supplied for the new type of event should be any additional attributes events of the new type will support in addition to the default header attributes.

Syntax

```
void event_type_add (in event_type_schema_t schema)
            raises (ExTypeAlreadyExists);
```

Exceptions

If there already exists an event type in the database maintained by the consumer/producer registry with a unique identifier field that is identical to the one supplied with the input type's schema, the **ExTypeAlreadyExists** exception is raised.

event_type_delete

The *event_type_delete* operation removes an event type schema from the consumer/producer registry. The input parameter should be the unique identifier for an event type that currently exists in the consumer/producer registry.

Syntax

```
void event_type_delete (in event_type_t name)
            raises (ExInvalidType);
```

Exceptions

If the input identifier for the event type to be removed does not match the unique identifier for an event type that currently exists in the consumer/producer registry, the **ExInvalidType** exception is raised.

event_type_get_list

The *event_type_get_list* operation returns the list of all event type schemas currently maintained within the consumer/producer registry. Each element within the returned list is a structure of the same type described above for the input parameter to *event_type_add*.

Syntax

```
void event_type_get_list (
                out event_type_list_t type_list)
            raises (ExNoTypeList);
```

Exceptions

If no event type schema currently exist in the consumer/producer registry, the **ExNoTypeList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

push_consumer_register

The *push_consumer_register* operation registers a new push consumer with the notification manager associated with the target consumer/producer registry. The first input parameter specifies the object reference of an object which supports the **CosEventComm::PushConsume**r interface. As described in COSS V1, an object supporting this interface will support a *push* operation which is invoked by an event supplier in order to send the consumer an event. The second input argument specifies a list of the names of filters that describe which events the new consumer is interested in receiving. Each filter name in the list should correspond to a filter that exists in the filter repository associated with the same notification manager that the target consumer/producer registry is associated with. The information about the new consumer and the names of filters that should be applied to events to determine which the new consumer is interested in receiving is maintained in the consumer/producer registry database.

Note that from the client's point of view, this operation is invoked on the CPRegistry interface supported by a particular notification manager. Effectively, this operation forms a connection between the target notification manager and a consumer. Recall that internally, each notification manager manages one or more event channels of the type described in COSS V1. It is anticipated that during the course of performing this operation the notification manager will invoke the appropriate operations to create the push consumer relationship between the appropriate event channel and the input consumer. Any event received by the notification manager for which one of the filters in the consumer's filter group evaluates to TRUE will be forwarded by the event channel to the consumer by invoking the *push* operation supported by the consumer's **CosEventComm::PushConsumer** interface. The precise relationship between notification managers and event channels, and how connections are established between event channels, event suppliers, and event consumers is left as an implementation detail since these aspects will be largely dependent on the specific implementation of the underlying event channels.

Syntax

```
void push_consumer_register (
                in CosEventComm::PushConsumer consumer,
                in filtername_list_t filter_group)
            raises (ExAlreadyRegistered,
                ExFilterNotFound,
                ExNotAuthorized);
```

Exceptions

If the input consumer is already registered with the target notification manager, the **ExAlreadyRegistered** exception is raised. If one of the event filters named in the list contained by the second input parameter does not exist in the filter repository associated with the target notification manager, the **ExFilterNotFound** exception is raised, with the "filter_name" field of the exception structure set to the name of the first such filter name encountered in the input list. If the consumer being registered is not authorized to register with the target notification manager, the **ExNotAuthorized** exception is raised with the "type" field set to the empty string. If the consumer being registered is authorized to register with the target notification manager, but not authorized to receive events of one or more types indicated by the filters being registered for, the **ExNotAuthorized** exception is raised with the "type" field set to the unique identifier of the first such event type encountered.

pull_consumer_register

The *pull_consumer_register* operation registers a new pull consumer with the notification manager associated with the target consumer/producer registry. The first input parameter specifies the object reference of an object which supports the **CosEventComm::PullConsumer** interface. As described in COSS V1, an object supporting this interface will form a connection with the an object supporting the CosEventComm::PullSupplier interface. Such an object supports *pull* and *try_pull* operations that enable the consumer to receive events available from the supplier, which is likely to be an event channel.

The second input argument specifies a list of the names of filters that describe which events the new consumer is interested in receiving. Each filter name in the list should correspond to a filter that exists in the filter repository associated with the same notification manager that the target consumer/producer registry is associated with. The information about the new consumer and the names of filters that should be applied to events to determine which the new consumer is interested in receiving is maintained in the consumer/producer registry database.

Note that from the client's point of view, this operation is invoked on the CPRegistry interface supported by a particular notification manager. Effectively, this operation forms a connection between the target notification manager and a consumer. Recall that internally, each notification manager manages one or more event channels of the type described in COSS V1. It is anticipated that during the course of performing this operation the notification manager will invoke the appropriate operations to create the pull consumer relationship between the appropriate event channel and the input consumer. Any event received by the notification manager for which one of the filters in the consumer's filter group evaluates to TRUE will be made available for pulling by the consumer. This can be done by having the client invoke the *consumer_receive* operation supported by the notification manager. This operation is likely to be implemented in terms of the *pull* or *try_pull* operations supported by an event channel managed by the notification manager. The precise relationship between notification managers and event channels, and how connections are established between event channels, event suppliers, and event consumers is left as an implementation detail since these aspects will be largely dependent on the specific implementation of the underlying event channels.

Syntax

```
void pull_consumer_register (
                in CosEventComm::PullConsumer consumer,
                in filtername_list_t filter_group)
            raises (ExAlreadyRegistered,
                ExFilterNotFound,
                ExNotAuthorized);
```

Exceptions

If the input consumer is already registered with the target notification manager, the **ExAlreadyRegistered** exception is raised. If one of the event filters named in the list contained by the second input parameter does not exist in the filter repository associated with the target notification manager, the **ExFilterNotFound** exception is raised, with the “filter_name” field of the exception structure set to the name of the first such filter name encountered in the input list. If the consumer being registered is not authorized to register with the target notification manager, the **ExNotAuthorized** exception is raised with the “type” field set to the empty string. If the consumer being registered is authorized to register with the target notification manager, but not authorized to receive events of one or more types indicated by the filters being registered for, the **ExNotAuthorized** exception is raised with the “type” field set to the unique identifier of the first such event type encountered.

consumer_handler_register

The *consumer_handler_register* operation indicates a specific method that should be invoked on a given consumer when transmitting an event of a given type to the consumer. This operation effectively makes the typed event style communication described in COSS V1 available to clients of this service, in addition to providing convenient support for a specific event->action binding.

The first input parameter specifies the object reference of the CosTypedEventComm::TypedPushConsumer interface of an event consumer. The consumer supporting this interface should have previously registered with the target notification manager as a push consumer. The second input parameter identifies a type of event to which this registration applies, which should be a type that already exists in the consumer/producer registry database. The third input parameter is the fully scoped name of an operation supported by the consumer which should be invoked instead of *push* whenever the notification manager forwards events of the type specified by the second input parameter to the consumer specified by the first input parameter.

Note that the mechanism by which the notification manager determines whether or not to forward each event of the given type to the consumer is identical to that which would be used if the consumer were a regular push consumer, and not a typed push consumer. The only difference is that some operation other than *push* would be invoked to transmit the event to the consumer.

Syntax

```
void consumer_handler_register (
    in CosTypedEventComm::TypedPushConsumer consumer,
    in event_type_t event_type,
    in string fully_scoped_name )
            raises (ExNotRegistered,
                ExInvalidType,
```

```
                    ExNotAuthorized);
```

Exceptions

If the input consumer is not currently registered with the target notification manager, the **ExNotRegistered** exception is raised. If the type specified by the second input parameter does not indicate an already registered event type, the **ExInvalidType** exception is raised. If the input consumer is not authorized to receive events of the input type, the **ExNotAuthorized** exception is raised.

consumer_unregister

The *consumer_unregister* operation removes knowledge of the consumer from the consumer/producer registry associated with the target notification manager. The input parameter specifies the object reference of the consumer to be unregistered.

Syntax

```
void consumer_unregister (in EventConsumer consumer)
            raises (ExNotRegistered);
```

Exceptions

If the input object reference does not indicate a consumer that is already registered with the target notification manager, the **ExNotRegistered** exception is raised.

consumer_get_list

The *consumer_get_list* operation returns two lists containing all of the consumers currently registered with the target notification manager. The first output parameter contains the list of object references for all registered push consumers, while the second output parameter contains the list of object references for all registered pull consumers.

Syntax

```
void consumer_get_list (
                out consumer_list_t push_consumers,
                out consumer_list_t pull_consumers)
            raises (ExNoConsumerList);
```

Exceptions

If no consumers are currently registered with the target notification manager, the **ExNoConsumerList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

push_supplier_register

The *push_supplier_register* operation registers a new push supplier with the notification manager associated with the target consumer/producer registry. The input parameter specifies the object reference of an object which supports the **CosEventComm::PushSupplier** interface. As described in COSS V1, an object supporting this interface will form a connection with the an object supporting the **CosEventComm::PushConsumer** interface. Such an object, which is often an event channel, supports a *push* operation that enables the supplier to send events to it.

Note that from the client's point of view, this operation is invoked on the CPRegistry interface supported by a particular notification manager. Effectively, this operation forms a connection between the target notification manager and a supplier. Recall that internally, each notification manager manages one or more event channels of the type described in COSS V1. It is anticipated that during the course of performing this operation the notification manager will invoke the appropriate operations to create the push supplier relationship between the appropriate event channel and the input supplier. This will effectively enable the supplier to transmit events through an event channel managed by the notification manager. This is performed under the seams whenever the supplier invokes the *supplier_send* operation supported by the notification manager. The precise relationship between notification managers and event channels, and how connections are established between event channels, event suppliers, and event consumers is left as an implementation detail since these aspects will be largely dependent on the specific implementation of the underlying event channels.

Syntax

```
void push_supplier_register (
                in CosEventComm::PushSupplier supplier)
            raises (ExAlreadyRegistered,
                ExNotAuthorized);
```

Exceptions

If the input supplier is already registered with the target notification manager, the **ExAlreadyRegistered** exception is raised. If the input supplier is not authorized to register with the target notification manager, the **ExNotAuthorized** exception is raised with the “type” field set to the empty string.

pull_supplier_register

The *pull_supplier_register* operation registers a new pull supplier with the notification manager associated with the target consumer/producer registry. The input parameter specifies the object reference of an object which supports the **CosEventComm::PullSupplier** interface. As described in COSS V1, an object supporting this interface will support *pull* and *try_pull* operations which is invoked by an event consumer in order to transmit events to the consumer an event.

Precisely how a notification manager will subsequently no when to pull events from suppliers registered using this operation is a detail left up to the implementer of this specification. It is envisioned that not all implementations will support this feature.

Syntax

```
void pull_supplier_register (
                in CosEventComm::PullSupplier supplier)
            raises (ExAlreadyRegistered,
                ExNotAuthorized);
```

Exceptions

If the input supplier is already registered with the target notification manager, the **ExAlreadyRegistered** exception is raised. If the input supplier is not authorized to register with the target notification manager, the **ExNotAuthorized** exception is raised with the “type” field set to the empty string.

supplier_unregister

The *supplier_unregister* operation removes knowledge of the consumer from the consumer/producer registry associated with the target notification manager. The input parameter specifies the object reference of the supplier to be unregistered.

Syntax

```
void supplier_unregister (in EventSupplier supplier)
            raises (ExNotRegistered);
```

Exceptions

If the input object reference does not indicate a supplier that is already registered with the target notification manager, the **ExNotRegistered** exception is raised.

supplier_get_list

The *supplier_get_list* operation returns two lists containing all of the suppliers currently registered with the target notification manager. The first output parameter contains the list of object references for all registered push suppliers, while the second output parameter contains the list of object references for all registered pull suppliers.

Syntax

```
void supplier_get_list (
                out supplier_list_t push_suppliers,
                out consumer_list_t pull_suppliers)
            raises (ExNoSupplierList);
```

Exceptions

If no suppliers are currently registered with the target notification manager, the **ExNoSupplierList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

add_filter_to_group

The *add_filter_to_group* operation adds new event filters to a consumer's event filter group. A consumer's event filter group contains all filters that the consumer has registered for, and thus collectively describes the all criteria that determine which events will be forwarded to the consumer.

The first input parameter is the object reference of the consumer whose event filter group is being updated. This consumer should be currently registered in the consumer/producer registry maintained by the target notification manager. The second input parameter contains a sequence of the names of event filters that should be added to the consumer's event filter group. Each element in the sequence should be the name an event filter that currently exists in the filter repository maintained by the target notification manager.

Syntax

```
void add_filter_to_group (in EventConsumer consumer,
                 in filtername_list_t filters)
            raises (ExNotRegistered,
                ExFilterNotFound,
                ExNotAuthorized);
```

Exceptions

If the first input parameter is not the object reference of a currently registered consumer, the **ExNotRegistered** exception is raised. If any of the elements in the second parameter is not the name of a filter which currently exists in the filter repository, the **ExFilterNotFound** exception is raised with the “filter_name” field of the exception structure set to the first such name encountered in the sequence. If the input consumer is not authorized to receive events of one or more types indicated by the filters being registered for, the **ExNotAuthorized** exception is raised with the “type” field set to the unique identifier of the first such event type encountered.

delete_filter_from_group

The *delete_filter_from_group* operation removes a filter from the group of filters that a consumer is currently registered for. The first input parameter is the object reference of the consumer whose event filter group is being updated. This consumer should be currently registered in the consumer/producer registry maintained by the target notification manager. The second input parameter contains a sequence of the names of event filters that should be removed from the consumer’s event filter group. Each element in the sequence should be the name an event filter that currently exists in the filter repository maintained by the target notification manager, and that is currently one of the event filters currently in the input consumers event filter group.

Syntax

```
void delete_filter_from_group (
                in EventConsumer consumer,
                in filtername_list_t filters)
            raises (ExNotRegistered,
                ExFilterNotFound);
```

Exceptions

If the first input parameter is not the object reference of a currently registered consumer, the **ExNotRegistered** exception is raised. If any of the elements in the second parameter is not the name of a filter which currently exists in the filter repository or which is not currently in the consumer’s event filter group, the **ExFilterNotFound** exception is raised with the “filter_name” field of the exception structure set to the first such name encountered in the sequence.

get_filter_group

The *get_filter_group* operation returns the list of filters for which a consumer is currently registered. The first parameter is the object reference of an event consumer that should be currently registered with the target notification manager. Upon return from this operation, the second parameter will contain a list of the names of all event filters for which the consumer is currently registered.

Syntax

```
void get_filter_group (in EventConsumer consumer,
                out filtername_list_t filters )
            raises (ExNotRegistered,
                ExNoFilterList);
```

Exceptions

If the first parameter is not the object reference of a currently registered consumer, the **ExNotRegistered** exception is raised. If the consumer is currently not registered for any event filters, the **ExNoFilterList** exception is raised. The purpose of raising this exception in this case is to differentiate a true empty list result from a failed operation.

A.1.1.3 The EventIterator Interface

As larger and more complex systems are built, there quickly becomes the possibility that a large number of events could be stored in the event repository at any given point. The EventIterator interfaces allows a client to iterate through a list of events - a subset of the events that are stored in the event repository - using the operations described in this section. The result of using this interface is an increased scalablity, usability and performance

next_one

The *next_one* operation returns the next event as well as *true* - showing that there was an event available for the request. If there are no more events, *false* is returned.

Syntax

```
boolean next_one (out event_t event);
```

Exceptions

Standard CORBA exceptions.

next_n

The *next_n* operation returns *how_many* events (the number of requested events) from the queue. Should there be fewer than *how_many* events available, the maximum number of events available will be sent back. In this method, the interpretation is that at most, *how_many* events will be returned. Should there be no more events, *false* is returned.

Syntax

```
boolean next_n ( in unsigned long how_many,
                out event_list_t el);
```

Exceptions

Standard CORBA exceptions.

destroy

The *destroy* operation destroys the iterator.

Syntax

```
void destroy();
```

Exceptions

Standard CORBA exceptions.

A.1.1.4 The NotificationManager Interface

The NotificationManager interface defines the operations supported by notification manager objects. Notification managers maintain information about event types, event filters, event suppliers, and event consumers known to the system. Multiple notification manager objects distributed throughout the network will cooperate to provide a distributed event management system in a large distributed environment. Precisely how notification managers are distributed (e.g., one per host, one per application per host, etc.) and how they cooperate are left as implementation details.

Notification managers conceptually play the role of the an object than encapsulates the behavior of all components of the event service conceptual model except for event producers and consumers. In other words, notification managers encapsulate the behavior of the filter repository, consumer producer registry, and event repository. In addition the notification manager plays the role of the event services manager that has the responsibility of configuring and managing event channels in order that they may provide the appropriate communication between event producers and consumers. It is assumed that COSS event channels will be the mechanism that notification managers use internally to provide the communication channels, however no part of the interface proposed here specifically enforces this as a requirement. The goal in defining the interface of the notification manager was that the behavior could be implemented on top of any event service implementation that complies with COSS V1, while not making any assumptions about the implementation specific decisions made in the underlying implementation of the COSS V1 compliant event service.

The NotificationManager interface inherits from the FilterRepository and CPRegistry interfaces described in the previous two subsections. It also defines some additional methods that enable push style suppliers to send messages, and pull style consumers to receive messages. In addition, the NotificationManager interfaces supports an administrative operation that is capable of returning the list of all other notification managers with which a particular notification manager is currently cooperating to provide a distributed service, and operations that enable management of the event repository associated with each notification manager. These operations are described below.

supplier_send

The *supplier_send* operation transmits a message from a push style supplier to a notification manager, so that it can be forwarded to all clients registered with a filter that evaluates to TRUE when applied to the event. The supplier sending the message should have previously registered with the target notification manager as a push supplier. Prior to invoking this operation, the supplier should set the “supplier” field of the origin structure in the event header to the object reference it used when registering with the target notification manager as a push supplier.

Syntax

```
void supplier_send (in event_t event)
            raises (ExNotRegistered,
                ExNotAuthorized);
```

Exceptions

If the supplier invoking this operation is not currently registered as a push supplier with the target notification manager, the **ExNotRegistered** exception is raised. If the supplier sending the event is not authorized to send events of the type being generated, the **ExNotAuthorized** exception is raised.

consumer_receive

The *consumer_receive* operation enables a pull style consumer to receive an event from a notification manager with which it has registered as a pull consumer. Note that the parameter passed to this operation can hold an event, and is listed as both an input and an output parameter. The reason why it is used as an input parameter is that it is the consumer's responsibility to set the supplier field of the origin structure in the even header to its own object reference when making this request. This enables the notification manager to verify that the consumer is indeed registered, and to locate the events that have passed through the consumer's event filter group and that have not yet been delivered to the consumer. The notification manager will maintain any events in its event repository that have passed a registered pull consumer's event filter group and have not yet been requested by the consumer.

Upon return, the parameter will contain an event if one is available for receiving by this consumer from the notification manager. If no event is available, the supplier field of the origin structure of the header will be set to OBJECT_NIL by the notification manager.

Syntax

```
void consumer_receive (inout event_t event)
            raises (ExNotRegistered);
```

Exceptions

If the consumer invoking the request has not previously registered with the notification manager, the **ExNotRegistered** exception is raised.

list_known_notification_managers

The *list_known_notification_managers* operation returns the list of other notification manager objects of which the target notification manager is aware. It is anticipated that multiple notification managers in a distributed environment will cooperate to provide a distributed event service. Precisely how notification managers are distributed, and to what degree they share information about clients and event types is currently left as an implementation detail until more sufficient implementation experience is gained. But it is anticipated that within each implementation of this service, multiple notification managers distributed in some fashion among the hosts in a managed environment will cooperate to provide distributed event management across a particular scope of influence. This operation will return the list of notification managers that reside within the same scope of influence as the target notification manager, and thus participate in providing the distributed service.

Syntax

```
void list_known_notification_managers (
            out sequence<NotificationManager> managers)
        raises (ExNoManagerList);
```

Exceptions

If the target notification manager is currently not cooperating with any other notification managers, the **ExNoManagerList** exception is raised. The purpose of raising an exception in this case is to distinguish a true NULL return list from a failed invocation.

get_undelivered_events

The *get_undelivered_events* operation returns the list of *how_many* events that currently reside within the event repository of the target notification manager and that are still awaiting delivery to one or more consumers. The event repository within each notification manager serves as a persistent storage of all events for which there are registered consumers that should receive the events but have not yet done so. One possible reason for an event to reside in this repository for an extended period of time is if there exists one or more pull consumers registered to receive it that have not yet done so. Also, it is possible that the notification manager has lost contact with a registered push consumer that should receive the event. By default, a notification manager will not delete an event from the event repository until the event has been delivered to all consumers that are registered for a filter that evaluates to TRUE when applied to the event. This operation allows a management application to browse through events within the event repository of the notification manager that are currently in this state and make a decision of whether or not to remove these pending events.

The input to this operation is how many events to return and the name of a filter. The output is a list of events that returns at most, *how_many* events and an EventIterator instance. Should there be less than *how_many* events, the available events will be returned. An EventIterator instance is returned, which contains information so as to be able to step through the list of events using the *next_one* and *next_n* operations.

Syntax

```
void get_undelivered_events (
                in string filter_name,
                in unsigned long how_many,
                out event_list_t event_list,
                out EventIterator ei)
            raises (ExNoEventList, ExFilterNotFound);
```

Exceptions

If there are no events within the event repository of the target notification manager currently awaiting delivery, the **ExNoEventList** exception is raised. The purpose of raising an exception in this case is to distinguish a true NULL return list from a failed invocation.

delete_undelivered_event

The *delete_undelivered_events* operation deletes events from the event repository of the target notification manager. The input paramater specifies a filter_name that currently exists within the event filter repository of the target notification manager awaiting delivery to one or more clients.

There are a few possible reasons why an event could reside within the event repository for an extended period of time. By default, a notification manager will not delete any events from its event repository until they have been delivered to all consumers that are registered for a filter that evaluates to TRUE when applied to the event. It is thus conceivable that a consumer could lose interest in an event that it had registered for prior to receiving it. This operation provides a facility to remove events from the event repository that are no longer of interest to any consumer, and thus allows an administrator to prevent the event repository from growing to an unreasonable size.

Syntax

```
void delete_undelivered_events_by_filter (
                in string filter_name )
            raises (ExNoEventList, ExFilterNotFound);
```

Exceptions

If there are no events matching the the given filter or if there are no events within the event repository of the target notification manager currently awaying delivery, the **ExNoEventList** exception is raised. The purpose of raising this exception in this case is to distinguish a true NULL return list from a failed invocation. If the passed the name of a filter that does not exist, the **ExFilterNotFound** exception will be raised.

A.1.2 IDL

```
#include <CosEvents.idl>

module ems {

    // Universally unique entity identifier
    typedef string uuid_t;

    // Attribute structure
    typedef any attr_value_t;
    typedef struct attributes_s_t {
        string name;
        attr_value_t value;
    } attribute_t;

    // Event type structure
    typedef uuid_t event_type_t;
    typedef struct eventid_s_t {
        event_type_t type;
        uuid_t id;
    } eventid_t;

    // Event origin
    typedef enum nameservice_s_t {
        ems_ns_other,
        ems_ns_dns,
        ems_ns_dce,
        ems_ns_x500,
        ems_ns_nis,
        ems_ns_sna
    } nameservice_t;
```

```
typedef struct netaddr_s_t {
    sequence<octet> name;}
netaddr_t;

typedef struct netname_s_t {
    nameservice_t service;
    netaddr_t netaddr;
} netname_t;

union EventConsumer switch (
            enum ConsumerType {PULLCONSUMER, PUSHCONSUMER}) {
    case PULLCONSUMER:
        CosEventComm::PullConsumer pullc;
    case PUSHCONSUMER:
        CosEventComm::PushConsumer pushc;
};
union EventSupplier switch (
                enum SupplierType {PULLSUPPLIER, PUSHSUPPLIER}) {
    case PULLSUPPLIER:
        CosEventComm::PullSupplier pulls;
    case PUSHSUPPLIER:
        CosEventComm::PushSupplier pushs;
};
typedef struct origin_s_t {
    EventSupplier supplier;
    netname_t netname;
    string descname;
    unsigned long pid;
    unsigned long uid;
    unsigned long gid;
} origin_t;

// Event severity
typedef enum severity_e_t {
    ems_sev_info,
    ems_sev_fatal,
    ems_sev_error,
    ems_sev_warning,
    ems_sev_notice,
    ems_sev_notice_verbose,
    ems_sev_debug
} severity_t;
```

```
// Timestamp
typedef unsigned long utc_t;

// Event Header
typedef struct header_s_t {
    eventid_t eventid;
    origin_t origin;
    severity_t severity;
    utc_t received;
    utc_t delivered;
} hdr_t;

// Event structure
typedef struct event_s_t {
    hdr_t header;
    sequence<attribute_t> items;
} event_t;

// Event type schema
typedef struct event_type_schema_s_t {
    event_type_t type;
    string name;
    sequence<attribute_t> attributes;
} event_type_schema_t;
typedef struct event_type_list_s_t {
    sequence<event_type_schema_t> schemas;
} event_type_list_t;

// Comparison operators
typedef enum attr_op_e_t {
    ems_c_attr_op_eq,
    ems_c_attr_op_gt,
    ems_c_attr_op_lt,
    ems_c_attr_op_ge,
    ems_c_attr_op_le,
    ems_c_attr_op_ne,
    ems_c_attr_op_bitand,
    ems_c_attr_op_substr
} attr_op_t;

// Filter expressions
```

```
typedef struct filter_exp_s_t {
    string attr_name;
    attr_op_t attr_operator;
    attr_value_t attr_value;
} filter_exp_t;

typedef struct ems_filter_exp_list_s_t {
    sequence<filter_exp_t> filter_exps;
} filter_exp_list_t;

// Event filter structure
typedef struct filter_s_t {
    string filter_name;
    event_type_t type;
    filter_exp_list_t filter_exp_list;
} filter_t;

// Filter names
typedef struct filtername_list_s_t {
    sequence<string> filter_names;
} filtername_list_t;

// Useful lists
typedef struct filter_list_s_t {
    sequence<filter_t> filters;
} filter_list_t;
typedef struct consumer_list_s_t {
    sequence<EventConsumer> consumers;
} consumer_list_t;
typedef struct supplier_list_s_t {
    sequence<EventSupplier> suppliers;
} supplier_list_t;
typedef struct event_list_s_t {
    sequence<event_t> events;
} event_list_t;

// Exceptions
exception ExFilterExists {};
exception ExInvalidFilter { string attr_name; };
exception ExFilterNotFound { string filter_name; };
exception ExFilterInUse { EventConsumer consumer; };
exception ExNoFilterList {};
```

```
exception ExTypeAlreadyExists {};
exception ExInvalidType {};
exception ExNoTypeList {};
exception ExNoSuchEvent {};
exception ExNoEventList {};
exception ExAlreadyRegistered {};
exception ExNotRegistered {};
exception ExNoConsumerList {};
exception ExNoSupplierList {};
exception ExNoManagerList {};
exception ExNotAuthorized { event_type_t type };

// Filter repository interface
interface FilterRepository {
    void filter_add (in string filter_name, in event_type_t type,
            in filter_exp_list_t exp_list)
        raises (ExFilterExists, ExInvalidFilter, ExInvalidType);
    void filter_append (in string filter_name,
            in filter_exp_list_t exp_list)
        raises (ExFilterNotFound, ExInvalidFilter);
    void filter_get (in string filter_name, out event_type_t type,
            out filter_exp_list_t exp_list)
        raises (ExFilterNotFound);
    void filter_delete (in string filter_name)
        raises (ExFilterNotFound, ExFilterInUse);
    void filter_get_namelist ( out filtername_list_t name_list )
        raises (ExNoFilterList);
    void filter_get_list ( out filter_list_t filter_list )
        raises (ExNoFilterList);
};

// Consumer/Producer Registry interface
interface CPRegistry {
    void event_type_add ( in event_type_schema_t schema )
        raises (ExTypeAlreadyExists);
    void event_type_delete (in event_type_t name )
        raises (ExInvalidType);
    void event_type_get_list ( out event_type_list_t type_list )
        raises (ExNoTypeList);
    void push_consumer_register (
            in CosEventComm::PushConsumer consumer,
            in filtername_list_t filter_group )
```

```
        raises (ExAlreadyRegistered,
            ExFilterNotFound, ExNotAuthorized);
    void pull_consumer_register (
            in CosEventComm::PullConsumer consumer,
            in filtername_list_t filter_group )
        raises (ExAlreadyRegistered,
            ExFilterNotFound, ExNotAuthorized);
    void consumer_handler_register (
            in CosTypedEventComm::TypedPushConsumer consumer,
            in event_type_t event_type,
            in string fully_scoped_method_name )
        raises (ExNotRegistered, ExInvalidType, ExNotAuthorized);
    void consumer_unregister ( in EventConsumer consumer )
        raises (ExNotRegistered);
    void consumer_get_list ( out consumer_list_t push_consumers,
            out consumer_list_t pull_consumers )
        raises (ExNoConsumerList);
    void push_supplier_register (
            in CosEventComm::PushSupplier supplier )
        raises (ExAlreadyRegistered, ExNotAuthorized);
    void pull_supplier_register (
            in CosEventComm::PullSupplier supplier )
        raises (ExAlreadyRegistered, ExNotAuthorized);
    void supplier_unregister ( in EventSupplier supplier )
        raises (ExNotRegistered);
    void supplier_get_list ( out supplier_list_t push_suppliers,
            out supplier_list_t pull_suppliers )
        raises (ExNoSupplierList);
    void add_filter_to_group ( in EventConsumer consumer,
            in filtername_list_t filters )
        raises (ExNotRegistered, ExFilterNotFound, ExNotAuthorized);
    void delete_filter_from_group ( in EventConsumer consumer,
            in filtername_list_t filters )
        raises (ExNotRegistered, ExFilterNotFound);
    void get_filter_group ( in EventConsumer consumer,
            out filtername_list_t filters )
        raises (ExNotRegistered, ExNoFilterList);
};

// EventIterator interface
interface EventIterator {
    boolean next_one (out event_t);
```

```
        boolean next_n (in unsigned long how_many,
                out event_list_t events);
        void destroy();
    }

    // Notification Manager interface
    interface NotificationManager : FilterRepository, CPRegistry {
        void supplier_send ( in event_t event )
            raises (ExNotRegistered, ExNotAuthorized);
        void consumer_receive ( inout event_t event )
            raises (ExNotRegistered);

        // Management operations
        void list_known_notification_managers
                ( out sequence<NotificationManager> managers )
            raises (ExNoManagerList);
        void get_undelivered_events ( in string filter_name,
                in unsigned long how_many,
                out event_list_t event_list,
                out EventIterator ei)
            raises (ExNoEventList, ExFilterNotFound);
        void delete_undelivered_event_by_filter ( in string filter_name )
            raises (ExNoEventList, ExFilterNotFound);
    };
};
```

Appendix A.2 *DCE Implementation*

This appendix describes the data structures, APIs, and command line interfaces required for a DCE implementation of the EMS. The EMS generic APIs were described earlier. The information provided here describes additional features and/or facilities provided by an DCE implementation to operate with DCE consumers and producers. It, also, describes the adapters available for interoperability of EMS implementations, as well as those for SNMP, et al.

A.2.1 DCE EMS Data Structure IDL File

```
/***********************************************************************/
/*  DCE IDL File  -EMS Data structures                                 */
/***********************************************************************/
[
uuid(000b0e1e-c016-1ce3-b57e-10005ab14004),
pointer_default(ptr),
version(1.0)
]
interface  event_management
{
```

```
 import "dce/utctypes.idl";
 import "dce/aclbase.idl";

typedef [string] char *                 ems_string_t;
/**********************************************************************/
/* Event Severity                                                     */
/**********************************************************************/
typedef enum {
        ems_sev_info,                 /* Information event            */
        ems_sev_fatal,                /* Fatal event                  */
        ems_sev_error,                /* Alert event                  */
        ems_sev_warning,              /* Warning event                */
        ems_sev_notice,               /* Notice event                 */
        ems_sev_notice_verbose,       /* Notice Verbose event         */
        ems_sev_debug                 /* Debug event                  */
} ems_severity_t;

/**********************************************************************/
/* Event Attribute Types                                              */
/**********************************************************************/
typedef unsigned16 ems_attr_type_t;
const ems_attr_type_t ems_c_attr_small_int   = 0;
const ems_attr_type_t ems_c_attr_short_int = 1;
const ems_attr_type_t ems_c_attr_long_int    = 2;
const ems_attr_type_t ems_c_attr_hyper_int = 3;
const ems_attr_type_t ems_c_attr_usmall_int = 4;
const ems_attr_type_t ems_c_attr_ushort_int = 5;
const ems_attr_type_t ems_c_attr_ulong_int = 6;
const ems_attr_type_t ems_c_attr_uhyper_int = 7;
const ems_attr_type_t ems_c_attr_short_float = 8;
const ems_attr_type_t ems_c_attr_long_float = 9;
const ems_attr_type_t ems_c_attr_boolean   = 10;
const ems_attr_type_t ems_c_attr_uuid        = 11;
const ems_attr_type_t ems_c_attr_utc         = 12;
const ems_attr_type_t ems_c_attr_severity    = 13;
const ems_attr_type_t ems_c_attr_acl         = 14;
const ems_attr_type_t ems_c_attr_byte_string = 15;
const ems_attr_type_t ems_c_attr_char_string = 16;
const ems_attr_type_t ems_c_attr_bytes            = 17;

/**********************************************************************/
```

```
/* Event Attribute Values                                                  */
/**************************************************************************/
typedef struct ems_bytes_s_t {
    unsigned32                          size;   /* size of byte data    */
    [ptr, size_is(size)] byte *         data;   /* byte data            */
} ems_bytes_t;

typedef union switch (ems_attr_type_t format) {
        case ems_c_attr_small_int:
            small int small_int;
        case ems_c_attr_short_int:
            short int short_int;
        case ems_c_attr_long_int:
            long int long_int;
        case ems_c_attr_hyper_int:
            hyper int hyper_int;
        case ems_c_attr_usmall_int:
            unsigned small int usmall_int;
        case ems_c_attr_ushort_int:
            unsigned short int ushort_int;
        case ems_c_attr_ulong_int:
            unsigned long int ulong_int;
        case ems_c_attr_uhyper_int:
            unsigned hyper int uhyper_int;
        case ems_c_attr_short_float:
            float short_float;
        case ems_c_attr_long_float:
            double long_float;
        case ems_c_attr_boolean:
            boolean bool;
        case ems_c_attr_uuid:
            uuid_t uuid;
        case ems_c_attr_utc:
            utc_t * utc;
        case ems_c_attr_severity:
            ems_severity_t severity;
        case ems_c_attr_acl:
            sec_acl_t * acl;
        case ems_c_attr_byte_string:
            [string] byte * byte_string;
        case ems_c_attr_char_string:
```

```
            [string] char * char_string;
        case ems_c_attr_bytes:
            ems_bytes_t  bytes;
        default:
            ;
} ems_attr_value_t;

/*********************************************************************/
/* Event Attribute                                                   */
/*********************************************************************/
typedef struct ems_attribute_s_t {
        ems_string_t            name;      /* Event Attribute Name      */
        ems_attr_value_t        value;     /* Event Attribute Type      */
} ems_attribute_t;

/*********************************************************************/
/* Event Types                                                       */
/*********************************************************************/
typedef uuid_t                     ems_event_type_t;

/*********************************************************************/
/* Event Id                                                          */
/*********************************************************************/
typedef struct ems_eventid_s_t {
        ems_event_type_t        type;        /* event type              */
        uuid_t                  id;          /* unique event identifier */
} ems_eventid_t;

/*********************************************************************/
/* Network Name                                                      */
/*********************************************************************/
typedef enum {
        ems_ns_other,             /* name service other than listed     */
        ems_ns_dns,               /* DNS name service                   */
        ems_ns_dce,               /* DCE CDS name service               */
        ems_ns_x500,              /* X500                               */
        ems_ns_nis,               /* NIS                                */
        ems_ns_sna                /* SNA network                        */
} ems_nameservice_t;

typedef char ems_octet_t;
```

```
typedef struct ems_netaddr_s_t {
        unsigned long len;                /* length of netaddr name     */
        [size_is(len)] char name[];        /* netaddr name              */
} ems_netaddr_t;

typedef struct ems_netname_s_t {
       ems_nameservice_t      service;         /* netname name service  */
       [ptr] ems_netaddr_t *  netaddr;         /* network name/address  */
} ems_netname_t;

/**********************************************************************/
/* Event Origin                                                       */
/**********************************************************************/
typedef struct ems_origin_s_t {
        ems_netname_t      netname;    /* originator host network name  */
        [string] char *    descname;   /* supplier descriptive name     */
        unsigned32         pid;        /* originator process id         */
        unsigned32         uid;        /* originator user id            */
        unsigned32         gid;        /* originator group id           */
} ems_origin_t;

/**********************************************************************/
/* Event Header                                                       */
/**********************************************************************/
typedef struct ems_hdr_s_t {
        ems_eventid_t         eventid;      /* Event Identifier         */
        ems_origin_t          origin;       /* Event Origin             */
        ems_severity_t        severity;     /* Event Severity           */
        utc_t                 received;     /* Event received timestamp */
        utc_t                 delivered;    /* Event received timestamp */
} ems_hdr_t;

/**********************************************************************/
/* Event                                                              */
/**********************************************************************/
typedef struct ems_event_s_t {
        ems_hdr_t               header;       /* fixed event header     */
        unsigned32              count;        /* number of data items   */
           [size_is(count)]
        ems_attribute_t         item[];       /* data items             */
} ems_event_t;
```

```
/**********************************************************************/
/*  Event Schema                                                      */
/**********************************************************************/
typedef struct ems_event_schema_s_t {
        ems_event_type_t        type;        /* EMS event type          */
        [string] char *         name;        /* event type name         */
        long                    size;        /* number of attributes    */
        [size_is(size)]
        ems_attribute_t         attribute[]; /* event type attributes   */
} ems_event_schema_t;

/**********************************************************************/
/* Event Type List                                                    */
/**********************************************************************/
typedef [ptr] ems_event_schema_t *  ems_schema_ptr_t;
typedef struct ems_event_type_list_s_t {
       long                    size;       /* number of event types     */
       [size_is(size)]
       ems_schema_ptr_t        schema[];   /* event type schemas        */
} ems_event_type_list_t;
/**********************************************************************/
/*  Event Filters                                                     */
/**********************************************************************/

/**********************************************************************/
/* Attribute Operators                                                */
/**********************************************************************/
typedef unsigned16 ems_attr_op_t;
const ems_attr_op_t ems_c_attr_op_eq    = 0;
const ems_attr_op_t ems_c_attr_op_gt    = 1;
const ems_attr_op_t ems_c_attr_op_lt    = 2;
const ems_attr_op_t ems_c_attr_op_ge    = 3;
const ems_attr_op_t ems_c_attr_op_le    = 4;
const ems_attr_op_t ems_c_attr_op_ne    = 5;
const ems_attr_op_t ems_c_attr_op_bitand = 6;
const ems_attr_op_t ems_c_attr_op_substr = 7;

/**********************************************************************/
/* Event Filter Expressions                                           */
/**********************************************************************/
```

```
typedef struct ems_filter_exp_s_t {
        [string] char *       attr_name;       /* Attribute Name        */
        ems_attr_op_t         attr_operator;   /* Attribute Operator    */
        ems_attr_value_t      attr_value;      /* Attribute Value       */
} ems_filter_exp_t;

/*********************************************************************/
/* Event Filter Expression List                                      */
/*********************************************************************/
typedef struct ems_filter_exp_list_s_t {
        long                    size;           /* number of filter     */
        [size_is(size)]                         /* expressions          */
        ems_filter_exp_t        filter_exps[];  /* filter expressions   */
} ems_filter_exp_list_t;

/*********************************************************************/
/* Event Filter                                                      */
/*********************************************************************/
typedef struct ems_filter_s_t {
        ems_string_t            filter_name;      /* Event Filter Name  */
        ems_event_type_t        type;                  /* Event type    */
        ems_filter_exp_list_t   filter_exp_list;  /* filter exp list    */
} ems_filter_t;

/*********************************************************************/
/* Event Filter Name List                                            */
/*********************************************************************/
typedef struct ems_filtername_list_s_t {
        long                size;           /* number of event filter   */
        [size_is(size)]                     /* names in list            */
        ems_string_t        filter_names[]; /* filter names             */
} ems_filtername_list_t;

/*********************************************************************/
/* Event Filter List                                                 */
/*********************************************************************/
typedef [ptr] ems_filter_t *        ems_filter_ptr_t;
typedef struct ems_filter_list_s_t {
      long                 size;       /* number of filters             */
      [size_is(size)]
      ems_filter_ptr_t     filter[];   /* ptrs to the event filters     */
```

```
} ems_filter_list_t;

/**********************************************************************/
/* Consumer Context Handle                                            */
/**********************************************************************/
typedef [context_handle] void * cons_context_t;

/**********************************************************************/
/* Consumer                                                           */
/**********************************************************************/
typedef struct ems_consumer_s_t {
        [string] char *         name;       /* DCE name of consumer     */
        [ptr] ems_netname_t *   hostname;   /* DCE hostname of consumer */
        uuid_t                  uuid;       /* consumers uuid           */
} ems_consumer_t;

/**********************************************************************/
/* Consumer List                                                   */
/**********************************************************************/
typedef struct ems_consumer_list_s_t {
        long                    size;        /* # of consumers          */
        [size_is(size)]
        ems_consumer_t          consumer[];  /* consumer info           */
} ems_consumer_list_t;

/**********************************************************************/
/* Attribute List                                                     */
/**********************************************************************/
typedef struct ems_attrlist_s_t {
        long                    size;         /* number of server       */
        [size_is(size)]                       /* attributes             */
        ems_attribute_t         attr[];       /* server attributes      */
} ems_attrlist_t;

}
```

A.2.2 DCE EMS API - ems.h

```
/***********************************************************************/
/* ems.h - DCE EMS Interface                                           */
/***********************************************************************/
#ifndef _DCE_EMS_H
#define _DCE_EMS_H
#include <stdarg.h>
#include <dce/dce.h>
#include <pthread.h>
#include <dce/dce_svc.h>
#include <dce/utctypes.h>
#include <dce/emsif.h>
#include <dce/dceemsmsg.h>
#include <dce/dbif.h>

/*
**  Compilation controls.
*/
#define DCE_SVC_WANT__FILE__

#define EMS_COMPONENT_NAME        "ems"
/***********************************************************************/
/* SVC severities                                                      */
/***********************************************************************/
#define SVC_C_SEV_FATAL                 \
     ((svc_c_sev_fatal&svc__c_mask)>>svc__c_shift)
#define SVC_C_SEV_ERROR                 \
     ((svc_c_sev_error&svc__c_mask)>>svc__c_shift)
#define SVC_C_SEV_WARNING         \
     ((svc_c_sev_warning&svc__c_mask)>>svc__c_shift)
#define SVC_C_SEV_NOTICE         \
     ((svc_c_sev_notice&svc__c_mask)>>svc__c_shift)
#define SVC_C_SEV_NOTICE_VERBOSE \
     ((svc_c_sev_notice_verbose&svc__c_mask)>>svc__c_shift)

/***********************************************************************/
/* Event Handler                                                       */
/***********************************************************************/
```

```
typedef void   (*ems_handler_t) ( ems_event_t  *event,
                                error_status_t *status);

/**********************************************************************/
/* Event Service Handle                                               */
/**********************************************************************/
typedef struct ems_handle_priv_s_t *        ems_handle_t;

/**********************************************************************/
/*  External interfaces                                               */
/**********************************************************************/
#define EMS_C_EMSD_OBJECT_UUID \
                (unsigned char *)"84ff9d30-08a2-11cf-ba2a-10005a4f3556"

/**********************************************************************/
/* Register with EMS (non consumer)                                   */
/**********************************************************************/
extern void ems_register(
            ems_netname_t *        hostname,   /* DCE host name         */
            ems_handle_t *         handle,     /* ems handle            */
            error_status_t *       status);    /* mgmt request status   */

/**********************************************************************/
/* UnRegister with EMS (non consumer)                                 */
/**********************************************************************/
extern void ems_unregister(
            ems_handle_t *         handle,     /* ems handle            */
            error_status_t *       status);    /* unregister status     */

/********************************/
/* Event Type Interface         */
/********************************/
#define EMS_C_SVC_TYPE_UUID \
                (unsigned_char_t *)"7d18dd10-7807-11ce-bef6-000001758810"

#define EMS_C_GENERIC_TYPE_UUID \
                (unsigned_char_t *)"632c65ee-911a-11ce-84ad-000001758810"
#define EMS_C_SVC_TYPE_NAME "SVC"
#define EMS_C_GENERIC_TYPE_NAME "Generic"

extern const ems_event_type_t                ems_c_svc_type;
extern const ems_event_type_t                ems_c_generic_type;

/**********************************************************************/
/* Add an Event Type                                                  */
```

```
/**********************************************************************/
extern void ems_event_type_add(
        ems_handle_t          handle,      /* ems handle              */
        ems_event_schema_t *  schema,      /* event type schema to add */
        error_status_t *      status);     /* request status          */

/**********************************************************************/
/* Delete an Event Type                                               */
/**********************************************************************/
extern void ems_event_type_delete(
        ems_handle_t          handle,     /* ems handle               */
        char *                type_name,  /* event type name to delete */
        error_status_t *      status);    /* request status           */

/**********************************************************************/
/* Get List of Available Event Types                                  */
/**********************************************************************/
extern void ems_event_type_get_list(
       ems_handle_t               handle,      /* ems handle          */
       ems_event_type_list_t **    type_list,   /* list of event types */
       error_status_t *            status);     /* request status      */

/**********************************************************************/
/* Free Event Types List                                              */
/**********************************************************************/
extern void ems_event_type_free_list(
        ems_event_type_list_t **   type_list,  /* list of event types  */
        error_status_t *           status);    /* request status       */

/********************************/
/* Supplier Interface           */
/********************************/

/**********************************************************************/
/* Supplier Send                                                      */
/**********************************************************************/
extern void ems_supplier_send(
        ems_handle_t            handle,          /* handle to emsd     */
        ems_event_t *           event,           /* Event data         */
        error_status_t *        status);         /* send status        */

/********************************/
/* Filter Interface             */
/********************************/
/**********************************************************************/
```

```
/* Add an Event Filter                                                  */
/************************************************************************/
extern void ems_filter_add(
       ems_handle_t            handle,       /* ems handle             */
       ems_string_t            filter_name,  /* event filter name      */
       ems_event_type_t        type,         /* event type             */
       ems_filter_exp_list_t * exp_list,     /* filter exprs to add    */
       error_status_t *        status);      /* filter routine status  */

/************************************************************************/
/* Update an Event Filter                                               */
/************************************************************************/
extern void ems_filter_append(
        ems_handle_t            handle,       /* ems handle            */
        ems_string_t            filter_name,  /* Event Filter Name     */
        ems_filter_exp_list_t * exp_list,     /* exp list to append    */
        error_status_t *        status);      /* Filter routine status */

/************************************************************************/
/* Get an Event Filter                                                  */
/************************************************************************/
extern void ems_filter_get(
        ems_handle_t            handle,       /* ems handle            */
        ems_string_t            filter_name,  /* event filter name     */
        ems_event_type_t*       type,         /* event type            */
        ems_filter_exp_list_t ** filter_exprs, /* returned filter exprs */
        error_status_t *        status);      /* filter routine status */

/************************************************************************/
/* Free an Event Filter - free the filter expression list              */
/************************************************************************/
extern void ems_filter_free(
        ems_filter_exp_list_t **  list,          /* filter exps to free */
        error_status_t *          status);       /* return status       */

/************************************************************************/
/* Delete an Event Filter                                               */
/************************************************************************/
extern void ems_filter_delete(
       ems_handle_t            handle,       /* ems handle             */
       ems_string_t            filter_name,  /* filter name to delete  */
       error_status_t *        status);      /* filter routine status  */

/************************************************************************/
/* List Event Filter Names                                              */
```

```
/***********************************************************************/
extern void ems_filter_get_namelist(
        ems_handle_t              handle,     /* ems handle             */
        ems_filtername_list_t **  name_list,  /* event filter name list */
        error_status_t *          status);    /* filter routine status  */

/***********************************************************************/
/* Free a filter namelist                                              */
/***********************************************************************/
extern void ems_filter_free_namelist(
        ems_filtername_list_t **  name_list,     /* namelist to free    */
        error_status_t *          status);       /* status              */

/***********************************************************************/
/* Get an Event Filter Database                                        */
/***********************************************************************/
extern void ems_filter_get_list(
         ems_handle_t          handle,        /* ems handle             */
         ems_filter_list_t **  filter_list,   /* returned filter list   */
         error_status_t *      status);       /* filter routine status  */

/***********************************************************************/
/* Free Event Filter List                                              */
/***********************************************************************/
extern void ems_filter_free_list(
         ems_filter_list_t **  filter_list,        /* list to free      */
         error_status_t *      status);            /* routine status    */

/********************************/
/* Consumer Interface           */
/********************************/
/***********************************************************************/
/* Consumer Start                                                      */
/***********************************************************************/
extern void ems_consumer_start(
         char *                consumer,        /* consumer name      i */
         unsigned32            flags,           /* consumer start flags */
         error_status_t *      status);         /* start status         */

/***********************************************************************/
/* Consumer Stop                                                       */
/***********************************************************************/
extern void ems_consumer_stop(
         error_status_t        *status);        /* stop status          */
```

```
/**********************************************************************/
/* Consumer Event Handler Register                                    */
/**********************************************************************/
extern void ems_consumer_handler_register(
         ems_handler_t         hfunc,     /* event handler function     */
         error_status_t *      status);   /* handler register status    */

/**********************************************************************/
/* Consumer Register                                                  */
/**********************************************************************/
extern void ems_consumer_register(
         ems_netname_t *          hostname,       /* emsd hostname      */
         ems_filtername_list_t * filter_group,    /* event filter group */
         ems_handle_t *          handle,          /* ems handle         */
         error_status_t *        status);         /* Register status    */

/**********************************************************************/
/* Consumer Unregister                                                */
/**********************************************************************/
extern void ems_consumer_unregister(
         ems_handle_t *         handle,         /* EMS binding handle   */
         error_status_t *       status);        /* unregister status    */

/**********************************************************************/
/* Add Event Filter To Group                                          */
/**********************************************************************/
extern void ems_add_filter_to_group(
       ems_handle_t            handle,        /* ems handle             */
       ems_filtername_list_t * event_filters, /* filter name list to add*/
       error_status_t *        status);       /* filter request status  */

/**********************************************************************/
/* Delete an Event Filter From a Group                                */
/**********************************************************************/
extern void ems_delete_filter_from_group(
          ems_handle_t            handle,      /* ems handle            */
          ems_filtername_list_t * filter_name, /* event filter name(s)  */
          error_status_t *        status);     /* filter request status */

/**********************************************************************/
/* Get a Consumers Event Filter Group                                 */
/**********************************************************************/
extern void ems_get_filter_group(
       ems_handle_t              handle,        /* ems handle           */
       ems_filtername_list_t **  filter_group,  /* Event Filter Group   */
```

```
       error_status_t *          status);       /* filter request status*/

/********************************/
/* Management Interface         */
/********************************/
/*********************************************************************/
/* List EMS Hosts                                                    */
/*********************************************************************/
extern void ems_mgmt_list_ems(
         char ***           host_list,    /* list of hosts running ems  */
         error_status_t *   status);      /* mgmt request status        */

/*********************************************************************/
/* Free EMS Hosts list                                               */
/*********************************************************************/
extern void ems_mgmt_free_ems(
         char ***           host_list,     /* list of hosts running ems */
         error_status_t *   status);       /* mgmt request status       */

/*********************************************************************/
/* Management List Server Attributes                                 */
/*********************************************************************/
extern void ems_mgmt_list_attributes(
        ems_handle_t       h,              /* ems handle                */
        ems_attrlist_t **  list,           /* returned attributes       */
        error_status_t *   status);        /* mgmt request status       */

/*********************************************************************/
/* Management Free Server Attributes list                            */
/*********************************************************************/
extern void ems_mgmt_free_attributes(
        ems_attrlist_t **    list,               /* attribute list      */
        error_status_t *     status);            /* mgmt request status */

/*********************************************************************/
/* List Registered consumers                                        */
/********************************************************************/
extern void ems_mgmt_list_consumers(
         ems_handle_t            handle,      /* ems handle             */
         ems_consumer_list_t **  list,        /* returned consumer list */
         error_status_t *        status);     /* mgmt request status    */

/*********************************************************************/
/* Free Consumer list                                                */
/*********************************************************************/
```

```
extern void ems_mgmt_free_consumers(
         ems_consumer_list_t **     list,      /* consumer list to free */
         error_status_t *           status);   /* mgmt request status   */

/**********************************************************************/
/* Delete Registered consumer from EMS                                */
/**********************************************************************/
extern void ems_mgmt_delete_consumer(
          ems_handle_t          handle,         /* ems handle           */
          char *                consumer,       /* Consumer's name      */
          uuid_t *              uuid,           /* Consumer UUID        */
          error_status_t *      status);        /* mgmt request status  */

/**********************************************************************/
/* Delete an Event Filter from a Consumer's Filter Group              */
/**********************************************************************/
extern void ems_mgmt_delete_filter_from_group(
        ems_handle_t              handle,            /* ems handle      */
        char *                    consumer,          /* Consumer's name */
        uuid_t *                  uuid,              /* Consumer UUID   */
        ems_filtername_list_t *   filter_name,       /* names to delete */
        error_status_t *          status);           /* mgmt req status */

/**********************************************************************/
/* Add an Event Filter to a Consumer's Event Filter Group             */
/**********************************************************************/
extern void ems_mgmt_add_filter_to_group(
        ems_handle_t            handle,       /* ems handle             */
        char *                  consumer,     /* Consumer's name        */
        uuid_t *                uuid,         /* Consumer UUID          */
        ems_filtername_list_t * filter_name,  /* name of filter to add  */
        error_status_t *        status);      /* mgmt request status    */

/**********************************************************************/
/* EMS Management - Get a filter group                                */
/**********************************************************************/
extern void ems_mgmt_get_filter_group(
         ems_handle_t               handle,       /* ems handle         */
         char *                     consumer,     /* name of consumer   */
         uuid_t *                   uuid,         /* Consumer UUID      */
         ems_filtername_list_t **   filter_group, /* event filter group */
         error_status_t *           status);      /* mgmt request status*/

/**********************************************************************/
/* Free an EMS Event                                                  */
```

```
/***********************************************************************/
extern void ems_event_free(
       ems_event_t **            event,         /* pointer to event     */
       error_status_t *          status);       /* event free status    */
#endif /* _EMS_H */
```

A.2.3 DCE dcecp commands for EMS

Event Service Object

ems catalog

ems show [-host hostname]

ems modify [-host hostname] [-add attr_name:attr_value -change attr_name:attr_value |
-delete attr_name]

Consumer Object

emsconsumer catalog [-host hostname]

emsconsumer show <consumer_name> [-uuid consumer_uuid][-host hostname]

emsconsumer delete <consumer_name> [-uuid consumer_uuid][-host hostname]

emsconsumer modify <consumer_name> [-uuid consumer_uuid][-host hostname]
-add filter_name | -delete filter_name

Filter Object

emsfilter catalog [-host hostname]

emsfilter show <filter_name> [-host hostname]

emsfilter delete <filter_name>[-host hostname]

emsfilter modify <filter_name> [-host hostname] [-append filter_exp]

Appendix A.3

UMA Implementation

This appendix describes the data structures, APIs, and command line interfaces required for a UMA implementation of the EMS. The EMS generic APIs were described earlier. The information provided here describes additional features and/or facilities provided by an UMA implementation to operate with UMA consumers and producers. It, also, describes the adapters available for interoperability of EMS implementations, as well as those for SNMP, et al.

A.3.1 Data Structures

A.3.2 APIs

A.3.3 Command Line Interfaces

Appendix B

Event Objects

This appendix contains descriptions of the event objects required for all conforming implementations of the EMS. This appendix is organized by management arenas, e.g., SNMP, CMIP, UMA datapools, etc.

B.1 CMIP

B.2 SNMP

* An SNMP OBJECT IDENTIFER type which needs to be mapped to ARRAY of Integer. Currently XEMS does not support this construct.

A.4 UMA Datapools

TABLE 1-1.

Index	Attribute Name	Attribute Format	Description
0	version	ems_c_attr_ulong_int	SNMP Version number that generated this message.
1	community	ems_c_attr_byte_string	Community name
2	Trap-PDU	ems_c_attr_ulong_int	Indicates this message carries a Trap PDU.
3	enterprise	ems_c_attr_char_string	MIB II sysObjectID of object generating trap.
4	agent-addr	ems_c_attr_byte_string	NetworkAddress of object genrating this trap.
5	generic-trap	ems_c_attr_long_int	A generic SNMP trap type.

TABLE 1-1.

Index	Attribute Name	Attribute Format	Description
6	specific-trap	ems_c_attr_long_int	Specific code is present even if generic trap is not enterpriseSpecific.
7	time-stamp	ems_c_attr_ulong_int	Time elapsed since the last (re)initialization of entity and the generation of this trap.
8+ N	variable-bindings		Message specific VarBindList begins.

A.5 DCE SVC Events

DCE SVC is used to route error messages from DCE Applications. Messages can be routed by severity to any of the following locations:

- A Text File
- A Binary File
- STDOUT
- STDERR

An additional routing was added to also send messages to XEMS. This section shows the mapping from an SVC message to an XEMS Event.

Data declaration of the SVC data structure:

```
typedef struct dce_svc_prolog_s_t {
    dce_svc_handle_t   handle;
    int                version;
    utc_t              t;
    const char         *argtypes;
    unsigned32         table_index;
    unsigned32         attributes;
    unsigned32         message_index;
    char               *format;
    const char         *file;
    char               progname[dce_svc_c_progname_buffsize];
    int                line;
    pthread_t          thread_id;
} *dce_svc_prolog_t;
```

:

TABLE 1-1.

Index	Attribute Name	Attribute Format	Description
0	version	ems_c_attr_ulong_int	Version number of interface that generated message.
1	t	ems_c_attr_utc	Time message was written
2	argtypes	ems_c_attr_char_string	The format-specifier string for the message.
3	table_index	ems_c_attr_ulong_int	Subcomponent table index
4	attributes	ems_c_attr_ulong_int	Message attributes, OR'd together
5	message_index	ems_c_attr_ulong_int	Index number of message in message table
6	format	ems_c_attr_char_string	Format argument values for the message
7	file	ems_c_attr_char_string	Name of the source file where message came from
8	progname	ems_c_attr_char_string	Program name where message came from
9	line	ems_c_attr_ulong_int	Line number where message came from
10	threadid	ems_c_attr_ulong_int	Thread ID of the thread that output the message
11	component_name	ems_c_attr_char_string	SVC Component name of program that output message
12	sc_name	ems_c_attr_char_string	Name SVC SubComponent that output message
13	attribute.debug	ems_c_attr_ushort_int	Debug attribute of message
14	attribute.severity	ems_c_attr_ushort_int	SVC Severity of message
15	attribute.actroute	ems_c_attr_ulong_int	Action/Routing attribute of message
16			SVC Argument 1
...			
15+N			SVC Argument N

Appendix B Event Objects

This appendix contains descriptions of the event objects required for all conforming implementations of the XEMS. This appendix is organized by management arenas, e.g., SNMP, CMIP, UMA datapools, etc.

B.1 CMIP

B.2 SNMP

TABLE 2-1.

Index	Attribute Name	Attribute Format	Description
0	version	ems_c_attr_ulong_int	SNMP Version number that generated this message.
1	community	ems_c_attr_byte_string	Community name

TABLE 2-1.

Index	Attribute Name	Attribute Format	Description
2	Trap-PDU	ems_c_attr_ulong_int	Indicates this message carries a Trap PDU.
3	enterprise	ems_c_attr_char_string	MIB II sysObjectID of object generating trap.
4	agent-addr	ems_c_attr_byte_string	NetworkAddress of object genrating this trap.
5	generic-trap	ems_c_attr_long_int	A generic SNMP trap type.
6	specific-trap	ems_c_attr_long_int	Specific code is present even if generic trap is not enterpriseSpecific.
7	time-stamp	ems_c_attr_ulong_int	Time elapsed since the last (re)initialization of entity and the generation of this trap.
8+ N	variable-bindings		Message specific VarBindList begins.

* An SNMP OBJECT IDENTIFER type which needs to be mapped to ARRAY of Integer. Currently XEMS does not support this construct.

TABLE 2-2.

SNMP Primitive Types	XEMS Attribute Format	Description
INTEGER	Any ems_c_attr_xxxx_int	Variable. The ASN.1 encoding for each variable provides its size information .
OCTET STRING	ems_c_attr_byte_string	May contain value X’00’.
OBJECT IDENTIFIER	ems_c_attr_char_string	An ASN.1 OBJECT IDENTIFIER is a sequence of integer literals in dot notation which are used to traverse the ISO global object tree.
NULL	ems_c_attr_????	ANSI NULL. Currently XEMS does not include this type.
Constructor Types		
SEQUENCE	Any ems attribute	An SNMP list; Implicit in XEMS schema
SEQUENCE OF <entry>	Any ems attribute	Where <entry> resolves to an SNMP list. An SNMP table; Implicit in XEMS schema.
Defined Types		

TABLE 2-2.

SNMP	XEMS	Description
IpADDRESS	ems_c_attr_byte_string	Currently, only the Internet protocol family is defined for NetworkAddress CHOICE.
Counter	ems_c_attr_ulong_int	Non-negative integer which monotonically increases and wraps to 0 after reaching maximum value (2^32-1).
Gauge	ems_c_attr_ulong_int	Non-negative integer which may increase or decrease, but which latches at a maximum value (2^32-1).
TimeTicks	ems_c_attr_ulong_int	Non-negative integer which counts the time in hundredths of a second since some epoch.
Opaque	ems_c_attr_bytes	Supports the capability to pass arbitrary ASN.1 syntax.
Common Constructs		
RequestId	ems_c_attr_long_int	Used by SNMP application entities to correlate incoming responses with outstanding requests.
ErrorStatus	ems_c_attr_long_int	A non-zero instance is used to indicate an exception.
ErrorIndex	ems_c_attr_ulong_int	May provide additional information by indicating which variable in a list caused the exception.
VarBind	ems attribute	Name Value pair. An SNMP name implies the data type. XEMS ems_attr_value_t component of ems_c_attribute_t stores value and data type.
VarBindList	XEMS Event Schema	Simple list of Name Value pairs.
PDUs		
GetRequest-PDU	ems_c_attr_ulong_int	0
GetNextRequest-PDU	ems_c_attr_ulong_int	1
GetResponse-PDU	ems_c_attr_ulong_int	2
SetRequest-PDU	ems_c_attr_ulong_int	3
Trap-PDU	ems_c_attr_ulong_int	4
MIB II		

TABLE 2-2.

SNMP	XEMS	Description
DisplayString	ems_c_attr_char_string	Printable display string restricted to the NVT ASCII character set defined in RFC 854.
PhysAddress	ems_c_attr_byte_string	Used to represent media- or physical-level addresses.

*Types derived from IETF RFC 1155 (SMI Primitive , Constructor and Defined Types), RFC 1157 (SNMP Common Constructs), and RFC 1213 (MIB II).*Types derived from IETF RFC 1155 (SMI Primitive , Constructor and Defined Types), RFC 1157 (SNMP Common Constructs), and RFC 1213 (MIB II).

B.3 UMA Datapools

B.4 DCE SVC Events

DCE SVC is used to route error messages from DCE Applications. Messages can be routed by severity to any of the following locations:

- A Text File
- A Binary File
- STDOUT
- STDERR

An additional routing was added to also send messages to XEMS. This section shows the mapping from an SVC message to an XEMS Event.

Data declaration of the SVC data structure:

```
typedef struct dce_svc_prolog_s_t {
    dce_svc_handle_t   handle;
    int                version;
    utc_t              t;
    const char         *argtypes;
    unsigned32         table_index;
    unsigned32         attributes;
    unsigned32         message_index;
    char               *format;
    const char         *file;
    char               progname[dce_svc_c_progname_buffsize];
    int                line;
    pthread_t          thread_id;
} *dce_svc_prolog_t;
```

:

TABLE 2-1. XEMS Mapping for a DCE SVC Message

Index	Attribute Name	Attribute Format	Description
0	version	ems_c_attr_ulong_int	Version number of interface that generated message.
1	t	ems_c_attr_utc	Time message was written
2	argtypes	ems_c_attr_char_string	The format-specifier string for the message.
3	table_index	ems_c_attr_ulong_int	Subcomponent table index
4	attributes	ems_c_attr_ulong_int	Message attributes, OR'd together
5	message_index	ems_c_attr_ulong_int	Index number of message in message table
6	format	ems_c_attr_char_string	Format argument values for the message
7	file	ems_c_attr_char_string	Name of the source file where message came from
8	progname	ems_c_attr_char_string	Program name where message came from
9	line	ems_c_attr_ulong_int	Line number where message came from
10	threadid	ems_c_attr_ulong_int	Thread ID of the thread that output the message
11	component_name	ems_c_attr_char_string	SVC Component name of program that output message
12	sc_name	ems_c_attr_char_string	Name SVC SubComponent that output message
13	attribute.debug	ems_c_attr_ushort_int	Debug attribute of message
14	attribute.severity	ems_c_attr_ushort_int	SVC Severity of message
15	attribute.actroute	ems_c_attr_ulong_int	Action/Routing attribute of message
16			SVC Argument 1
...			
15+N			SVC Argument N

Appendix C *xems.h*

```
#ifndef _XEMS_H

/*----------------------------------------------------------------------*/
/* XEMS Data Structures                                                 */
/*----------------------------------------------------------------------*/

/***********************************************/
/* XEMS generic data types                     */
/***********************************************/
typedef unsigned char          ems_boolean ;
#define ems_false              false
#define ems_true               true
typedef unsigned char          ems_byte ;
typedef unsigned char          ems_char ;
typedef signed char            ems_small_int ;
typedef unsigned char          ems_usmall_int ;
typedef short int              ems_short_int ;
typedef unsigned short int     ems_ushort_int ;
typedef long int               ems_long_int ;
typedef unsigned long int      ems_ulong_int ;
struct ems_hyper_int_rep_s_t {
         ems_long_int          high;
         ems_ulong_int         low;
```

```
} ems_hyper_int;
struct ems_uhyper_int_rep_s_t {
         ems_ulong_int          high;
         ems_ulong_int          low;
} ems_uhyper_int;
typedef float                  ems_short_float ;
typedef double                 ems_long_float ;

typedef char                   *ems_string_t;
typedef struct uuid_t {
    ems_ulong_int          time_low;
    ems_ushort_int         time_mid;
    ems_ushort_int         time_hi_and_version;
    ems_usmall_int         clock_seq_hi_and_reserved;
    ems_usmall_int         clock_seq_low;
        ems_byte               node[6];
} ems_uuid_t;

typedef struct utc {
        ems_byte               char_array[16];
} ems_utc_t;

typedef ems_ulong_int          ems_error_t;

typedef ems_uuid_t             ems_event_type_t;

/************************************************/
/* XEMS severity                                */
/************************************************/
typedef enum {
        ems_sev_info,
        ems_sev_fatal,
        ems_sev_error,
        ems_sev_warning,
        ems_sev_notice,
        ems_sev_notice_verbose,
        ems_sev_debug
} ems_severity_t;

/************************************************/
/* XEMS priority                                */
/************************************************/
```

```
typedef ems_ulong_int          ems_priority_t;

/*********************************************/
/* XEMS Attributes                           */
/*********************************************/
typedef ems_ushort_int         ems_attr_type_t;

#define ems_c_attr_small_int    (0)
#define ems_c_attr_short_int    (1)
#define ems_c_attr_long_int     (2)
#define ems_c_attr_hyper_int    (3)
#define ems_c_attr_usmall_int   (4)
#define ems_c_attr_ushort_int   (5)
#define ems_c_attr_ulong_int    (6)
#define ems_c_attr_uhyper_int   (7)
#define ems_c_attr_short_float  (8)
#define ems_c_attr_long_float   (9)
#define ems_c_attr_boolean      (10)
#define ems_c_attr_uuid         (11)
#define ems_c_attr_utc          (12)
#define ems_c_attr_severity     (13)
#define ems_c_attr_byte_string  (15)
#define ems_c_attr_char_string  (16)
#define ems_c_attr_bytes        (17)

typedef struct ems_bytes_s_t {
        ems_ulong_int   size;
        ems_byte        *data;
} ems_bytes_t;

typedef struct  {
        ems_attr_type_t format;
        union  {
            /* case(s): ems_c_attr_small_int */
            ems_small_int small_int;
            /* case(s): ems_c_attr_short_int */
            ems_short_int short_int;
            /* case(s): ems_c_attr_long_int */
            ems_long_int long_int;
            /* case(s): ems_c_attr_hyper_int */
            ems_hyper_int hyper_int;
            /* case(s): ems_c_attr_usmall_int */
```

```
            ems_usmall_int usmall_int;
            /* case(s): ems_c_attr_ushort_int */
            ems_ushort_int ushort_int;
            /* case(s): ems_c_attr_ulong_int */
            ems_ulong_int ulong_int;
            /* case(s): ems_c_attr_uhyper_int */
            ems_uhyper_int uhyper_int;
            /* case(s): ems_c_attr_short_float */
            ems_short_float short_float;
            /* case(s): ems_c_attr_long_float */
            ems_long_float long_float;
            /* case(s): ems_c_attr_boolean */
            ems_boolean bool;
            /* case(s): ems_c_attr_uuid */
            ems_uuid_t uuid;
            /* case(s): ems_c_attr_utc */
            ems_utc_t *utc;
            /* case(s): ems_c_attr_severity */
            ems_severity_t severity;
            /* case(s): ems_c_attr_byte_string */
            ems_byte *byte_string;
            /* case(s): ems_c_attr_char_string */
            char *char_string;
            /* case(s): ems_c_attr_bytes */
            ems_bytes_t bytes;
            } tagged_union;
        } ems_attr_value_t;

typedef struct ems_attribute_s_t {
        ems_string_t           name;
        ems_attr_value_t       value;
} ems_attribute_t;

/***********************************************/
/* XEMS event id                               */
/***********************************************/
typedef struct ems_eventid_s_t {
        ems_event_type_t       type;
        ems_uuid_t             id;
} ems_eventid_t;

/***********************************************/
```

```
/* XEMS network name structure                    */
/*************************************************/
typedef enum {
        ems_ns_other,
        ems_ns_dns,
        ems_ns_dce,
        ems_ns_x500,
        ems_ns_nis,
        ems_ns_sna
} ems_nameservice_t;

typedef char ems_octet_t;
typedef struct ems_netaddr_s_t {
        ems_ulong_int               len;
        ems_octet_t                 name[1];
} ems_netaddr_t;

typedef struct ems_netname_s_t {
        ems_nameservice_t      service;
        ems_netaddr_t          *netaddr;
} ems_netname_t;

/*************************************************/
/* XEMS event origin                              */
/*************************************************/
typedef struct ems_origin_s_t {
        ems_netname_t          netname;
        ems_string_t           descname;
        ems_ulong_int          pid;
        ems_ulong_int          uid;
        ems_ulong_int          gid;
} ems_origin_t;

/*************************************************/
/* XEMS event header                              */
/*************************************************/
typedef struct ems_hdr_s_t {
        ems_eventid_t          eventid;
        ems_origin_t           origin;
        ems_severity_t         severity;
        ems_utc_t              received;
        ems_utc_t              delivered;
```

```
        ems_priority_t          priority;
} ems_hdr_t;

/*********************************************/
/* XEMS event structure                      */
/*********************************************/
typedef struct ems_event_s_t {
        ems_hdr_t               header;
        ems_ulong_int           count;
        ems_attribute_t         item[1];
} ems_event_t;

/*********************************************/
/* XEMS event schema                         */
/*********************************************/
typedef struct ems_event_schema_s_t {
        ems_event_type_t        type;
        ems_string_t            name;
        ems_long_int            size;
        ems_attribute_t         attribute[1];
} ems_event_schema_t;

typedef ems_event_schema_t      *ems_schema_ptr_t;
typedef struct ems_event_type_list_s_t {
        ems_long_int            size;
        ems_schema_ptr_t        schema[1];
} ems_event_type_list_t;

/*********************************************/
/* XEMS filter expression operators          */
/*********************************************/
typedef ems_ushort_int          ems_attr_op_t;

#define ems_c_attr_op_eq        (0)
#define ems_c_attr_op_gt        (1)
#define ems_c_attr_op_lt        (2)
#define ems_c_attr_op_ge        (3)
#define ems_c_attr_op_le        (4)
#define ems_c_attr_op_ne        (5)
#define ems_c_attr_op_bitand    (6)
#define ems_c_attr_op_substr    (7)
```

```
/*********************************************/
/* XEMS filter expression                   */
/*********************************************/
typedef struct ems_filter_exp_s_t {
        ems_string_t           attr_name;
        ems_attr_op_t          attr_operator;
        ems_attr_value_t       attr_value;
} ems_filter_exp_t;

/*********************************************/
/* XEMS filter expression list              */
/*********************************************/
typedef struct ems_filter_exp_list_s_t {
        ems_long_int            size;
        ems_filter_exp_t        filter_exps[1];
} ems_filter_exp_list_t;

/*********************************************/
/* XEMS event filter structures             */
/*********************************************/
typedef struct ems_filter_s_t {
        ems_string_t            filter_name;
        ems_event_type_t       type;
        ems_filter_exp_list_t  filter_exp_list;
} ems_filter_t;

typedef struct ems_filtername_list_s_t {
        ems_long_int           size;
        ems_string_t           filter_names[1];
} ems_filtername_list_t;

typedef ems_filter_t *ems_filter_ptr_t;
typedef struct ems_filter_list_s_t {
        ems_long_int           size;
        ems_filter_ptr_t       filter[1];
} ems_filter_list_t;

/*********************************************/
/* XEMS consumer structures                 */
/*********************************************/
typedef struct ems_consumer_s_t {
        ems_string_t           name;
```

```
        ems_netname_t           *hostname;
        ems_uuid_t              uuid;
} ems_consumer_t;

typedef struct ems_consumer_list_s_t {
        ems_long_int            size;
        ems_consumer_t          consumer[1];
} ems_consumer_list_t;

/************************************************/
/* XEMS supplier structures                     */
/************************************************/
typedef struct ems_supplier_s_t {
        ems_string_t            name;
        ems_netname_t           *hostname;
        ems_uuid_t              uuid;
} ems_supplier_t;

typedef struct ems_supplier_list_s_t {
        ems_long_int            size;
        ems_supplier_t          supplier[1];
} ems_supplier_list_t;

/************************************************/
/* XEMS attribute list                          */
/************************************************/
typedef struct ems_attrlist_s_t {
        ems_long_int            size;
        ems_attribute_t         attr[1];
} ems_attrlist_t;

/************************************************/
/* XEMS event list                              */
/************************************************/
typedef ems_event_t             *ems_event_ptr_t;
typedef struct ems_event_list_s_t {
        ems_long_int            size;
        ems_event_ptr_t         event[1];
} ems_event_list_t;

/************************************************/
/* Event Handler                                */
```

```
/***********************************************/
typedef void   (*ems_handler_t) ( ems_event_t  *event,
                                  ems_error_t *status);

/***********************************************/
/* Event Service Handle                        */
/***********************************************/
typedef struct ems_handle_priv_s_t *ems_handle_t;

/*----------------------------------------------------------------------*/
/* XEMS Registration API                                                */
/*----------------------------------------------------------------------*/
/***********************************************/
/* Register with XEMS                          */
/***********************************************/
extern
void ems_register(
        ems_netname_t *        hostname,   /* Event Service hostname */
        ems_handle_t *         handle,     /* XEMS handle            */
        ems_error_t *          status);    /* operation status       */

/***********************************************/
/* UnRegister with XEMS                     */
/***********************************************/
extern
void ems_unregister(
        ems_handle_t *         handle,     /* XEMS handle          */
        ems_error_t *          status);    /* operation status      */

#define EMS_C_GENERIC_TYPE_UUID \
               (ems_string_t)"632c65ee-911a-11ce-84ad-000001758810"

/*----------------------------------------------------------------------*/
/* XEMS Event Type API                                                  */
/*----------------------------------------------------------------------*/
/***********************************************/
/* Add an Event Type                           */
/***********************************************/
extern
void ems_event_type_add(
        ems_handle_t          handle,      /* XEMS handle          */
        ems_event_schema_t *  schema,      /* event type schema to add */
```

```
        ems_error_t *         status);     /* request status           */

/**********************************************/
/* Delete an Event Type                       */
/**********************************************/
extern
void ems_event_type_delete(
        ems_handle_t           handle,     /* XEMS handle          */
        ems_string_t           type_name,  /* event type name to delete */
        ems_event_type_t *     type,       /* event type id to delete   */
        ems_error_t *          status);    /* request status            */

/**********************************************/
/* Get an Event Type                          */
/**********************************************/
extern
void ems_event_type_get(
        ems_handle_t           handle,     /* XEMS handle          */
        ems_string_t           type_name,  /* event type name to get */
        ems_event_type_t *     type,       /* event type id to get   */
        ems_event_schema_t **  schema,     /* event type schema      */
        ems_error_t *          status);    /* request status         */

/**********************************************/
/* Get List of Available Event Types          */
/**********************************************/
extern
void ems_event_type_get_list(
         ems_handle_t             handle,        /* XEMS handle          */
         ems_event_type_list_t ** type_list,     /* list of event types  */
         ems_error_t *            status);       /* request status       */

/**********************************************/
/* Free Event Types List                    */
/**********************************************/
extern
void ems_event_type_free_list(
        ems_event_type_list_t **   type_list,  /* list of event types  */
        ems_error_t *              status);    /* request status       */

/*-------------------------------------------------------------------*/
/* XEMS Supplier Interface                                           */
```

```
/*----------------------------------------------------------------------*/
/************************************************/
/* Pull Supplier Register                   */
/************************************************/
extern
void ems_pull_supplier_register(
        ems_netname_t *        hostname,        /* Event Service hostname */
         ems_handle_t *         handle,         /* XEMS handle          */
         ems_error_t *          status);         /* Register status    */

/************************************************/
/* Push Supplier Register                   */
/************************************************/
extern
void ems_push_supplier_register(
        ems_netname_t *        hostname,        /* Event Service hostname */
         ems_handle_t *         handle,         /* XEMS handle          */
         ems_error_t *          status);         /* Register status    */

/************************************************/
/* Supplier Unregister                          */
/************************************************/
extern
void ems_supplier_unregister(
         ems_handle_t *        handle,           /* XEMS handle          */
         ems_error_t *         status);          /* unregister status    */

/************************************************/
/* Supplier Send                                */
/************************************************/
extern
void ems_supplier_send(
         ems_handle_t           handle,         /* XEMS handle          */
         ems_event_t *          event,          /* Event data           */
         ems_error_t *          status);        /* send status          */

/*----------------------------------------------------------------------*/
/* XEMS Filter Interface                                                */
/*----------------------------------------------------------------------*/
/************************************************/
/* Add an Event Filter                          */
/************************************************/
```

```
extern
void ems_filter_add(
        ems_handle_t             handle,       /* XEMS handle          */
        ems_string_t             filter_name,  /* event filter name      */
        ems_event_type_t         type,         /* event type             */
        ems_filter_exp_list_t *  exp_list,     /* filter exprs to add    */
        ems_error_t *            status);      /* filter routine status  */

/***********************************************/
/* Update an Event Filter                      */
/***********************************************/
extern
void ems_filter_append(
        ems_handle_t             handle,       /* XEMS handle          */
        ems_string_t             filter_name,  /* Event Filter Name        */
        ems_filter_exp_list_t *  exp_list,     /* exp list to append       */
        ems_error_t *            status);      /* Filter routine status    */

/***********************************************/
/* Get an Event Filter                         */
/***********************************************/
extern
void ems_filter_get(
        ems_handle_t             handle,       /* XEMS handle          */
        ems_string_t             filter_name,  /* event filter name     */
        ems_event_type_t *       type,         /* event type            */
        ems_filter_exp_list_t ** filter_exprs, /* returned filter exprs */
        ems_error_t *            status);      /* filter routine status */

/***********************************************/
/* Free an Event Filter -                      */
/*       free the filter expression list       */
/***********************************************/
extern
void ems_filter_free(
        ems_filter_exp_list_t **   list,         /* filter exps to free */
        ems_error_t *              status);      /* return status       */

/***********************************************/
/* Delete an Event Filter                      */
/***********************************************/
extern
```

```
void ems_filter_delete(
        ems_handle_t            handle,       /* XEMS handle          */
        ems_string_t            filter_name,  /* filter name to delete   */
        ems_error_t *           status);      /* filter routine status   */

/**********************************************/
/* List Event Filter Names                   */
/**********************************************/
extern
void ems_filter_get_namelist(
        ems_handle_t             handle,     /* XEMS handle          */
        ems_filtername_list_t ** name_list,  /* event filter name list */
        ems_error_t *            status);    /* filter routine status  */

/**********************************************/
/* Free a filter namelist                    */
/**********************************************/
extern
void ems_filter_free_namelist(
        ems_filtername_list_t ** name_list,       /* namelist to free   */
        ems_error_t *            status);         /* status             */

/**********************************************/
/* Get an Event Filter Database              */
/**********************************************/
extern
void ems_filter_get_list(
        ems_handle_t             handle,        /* XEMS handle          */
        ems_filter_list_t **     filter_list,   /* returned filter list   */
        ems_error_t *            status);       /* filter routine status  */

/**********************************************/
/* Free Event Filter List                    */
/**********************************************/
extern
void ems_filter_free_list(
        ems_filter_list_t **     filter_list,     /* list to free     */
        ems_error_t *            status);         /* routine status   */

/*-------------------------------------------------------------------*/
/* XEMS Consumer Interface                                           */
/*-------------------------------------------------------------------*/
```

```
/**********************************************/
/* Consumer Start                             */
/**********************************************/
extern
void ems_consumer_start(
         ems_string_t          consumer,      /* consumer name          */
         ems_ulong_int         flags,         /* consumer start flags  */
         ems_error_t *         status);       /* start status           */

/**********************************************/
/* Consumer Stop                              */
/**********************************************/
extern
void ems_consumer_stop(
         ems_error_t *          status);      /* stop status         */

/**********************************************/
/* Consumer Event Handler Register            */
/**********************************************/
extern
void ems_consumer_handler_register(
         ems_handler_t        hfunc,     /* event handler function   */
         ems_error_t *        status);   /* handler register status  */

/**********************************************/
/* Push Consumer Register                     */
/**********************************************/
extern
void ems_push_consumer_register(
         ems_netname_t *         hostname,     /* Event Service hostname */
         ems_filtername_list_t * filter_group, /* event filter group     */
         ems_handle_t *          handle,       /* XEMS handle            */
         ems_error_t *           status);      /* Register status        */

/**********************************************/
/* Pull Consumer Register                     */
/**********************************************/
extern
void ems_pull_consumer_register(
         ems_netname_t *         hostname,     /* Event Service hostname */
         ems_filtername_list_t * filter_group, /* event filter group     */
         ems_handle_t *          handle,       /* XEMS handle            */
```

```
        ems_error_t *           status);      /* Register status        */

/*************************************************/
/* Consumer Unregister                           */
/*************************************************/
extern
void ems_consumer_unregister(
        ems_handle_t *         handle,           /* XEMS handle          */
        ems_error_t *          status);          /* unregister status    */

/*************************************************/
/* Add Event Filter To Group                     */
/*************************************************/
extern
void ems_add_filter_to_group(
        ems_handle_t            handle,        /* XEMS handle              */
        ems_filtername_list_t * event_filters, /* filter name list to add */
        ems_error_t *           status);       /* filter request status   */

/*************************************************/
/* Delete an Event Filter From a Group           */
/*************************************************/
extern
void ems_delete_filter_from_group(
        ems_handle_t             handle,      /* XEMS handle           */
        ems_filtername_list_t *  filter_name, /* event filter name(s)  */
        ems_error_t *            status);     /* filter request status */

/*************************************************/
/* Get a Consumers Event Filter Group            */
/*************************************************/
extern
void ems_get_filter_group(
        ems_handle_t              handle,        /* XEMS handle           */
        ems_filtername_list_t **  filter_group,  /* Event Filter Group    */
        ems_error_t *             status);       /* filter request status */

/*------------------------------------------------------------------*/
/* XEMS Management Interface                                        */
/*------------------------------------------------------------------*/
/*************************************************/
/* List XEMS Hosts                               */
```

```
/**********************************************/
extern
void ems_mgmt_list_ems(
        ems_string_t **        host_list,   /* Event Service hosts list */
        ems_error_t *          status);     /* mgmt request status      */

/**********************************************/
/* Free XEMS Hosts list                       */
/**********************************************/
extern
void ems_mgmt_free_ems(
        ems_strint_t **    host_list,       /* Event Service hosts list */
        ems_error_t *      status);         /* mgmt request status      */

/**********************************************/
/* Management List Server Attributes          */
/**********************************************/
extern
void ems_mgmt_list_attributes(
        ems_handle_t          h,              /* XEMS handle          */
        ems_attrlist_t **     list,           /* returned attributes  */
        ems_error_t *         status);        /* mgmt request status  */

/**********************************************/
/* Management Free Server Attributes list     */
/**********************************************/
extern
void ems_mgmt_free_attributes(
        ems_attrlist_t **      list,          /* attribute list       */
        ems_error_t *          status);       /* mgmt request status  */

/**********************************************/
/* List Registered consumers                   */
/**********************************************/
extern
void ems_mgmt_list_consumers(
        ems_handle_t            handle,      /* XEMS handle              */
        ems_consumer_list_t **  list,        /* returned consumer list  */
        ems_error_t *           status);     /* mgmt request status     */

/**********************************************/
/* Free Consumer list                          */
```

```
/***********************************************/
extern
void ems_mgmt_free_consumers(
        ems_consumer_list_t **   list,      /* consumer list to free   */
        ems_error_t *            status);   /* mgmt request status     */

/***********************************************/
/* Delete Registered consumer from XEMS         */
/***********************************************/
extern
void ems_mgmt_delete_consumer(
        ems_handle_t          handle,        /* XEMS handle          */
        ems_string_t          consumer,      /* Consumer's name      */
        ems_uuid_t *          uuid,          /* Consumer UUID        */
        ems_error_t *         status);       /* mgmt request status  */

/***********************************************/
/* Delete an Event Filter from a Consumer's     */
/*      Filter Group                            */
/***********************************************/
extern
void ems_mgmt_delete_filter_from_group(
        ems_handle_t             handle,          /* XEMS handle        */
        char *                   consumer,        /* Consumer's name    */
        ems_uuid_t *             uuid,            /* Consumer UUID      */
        ems_filtername_list_t *  filter_name,     /* names to delete    */
        ems_error_t *            status);         /* mgmt req status    */

/***********************************************/
/* Add an Event Filter to a Consumer's Event    */
/*      Filter Group                            */
/***********************************************/
extern
void ems_mgmt_add_filter_to_group(
         ems_handle_t              handle,       /* XEMS handle           */
         char *                    consumer,     /* Consumer's name       */
         ems_uuid_t *              uuid,         /* Consumer UUID         */
         ems_filtername_list_t *   filter_name,  /* name of filter to add */
         ems_error_t *             status);      /* mgmt request status   */

/***********************************************/
/* XEMS Management - Get a filter group         */
```

```
/*********************************************************/
extern
void ems_mgmt_get_filter_group(
        ems_handle_t              handle,       /* XEMS handle          */
        char *                    consumer,     /* name of consumer     */
        ems_uuid_t *              uuid,         /* Consumer UUID        */
        ems_filtername_list_t **  filter_group, /* event filter group   */
        ems_error_t *             status);      /* mgmt request status */

/*********************************************************/
/* List Registered suppliers                             */
/*********************************************************/
extern
void ems_mgmt_list_suppliers(
        ems_handle_t              handle,     /* XEMS handle             */
        ems_supplier_list_t **    list,       /* returned supplier list */
        ems_error_t *             status);    /* mgmt request status    */

/*********************************************************/
/* Free Consumer list                                    */
/*********************************************************/
extern
void ems_mgmt_free_suppliers(
        ems_supplier_list_t **     list,      /* supplier list to free  */
        ems_error_t *              status);   /* mgmt request status    */

/*********************************************************/
/* Delete Registered supplier from XEMS                  */
/*********************************************************/
extern
void ems_mgmt_delete_supplier(
        ems_handle_t          handle,         /* XEMS handle          */
        ems_string_t          supplier,       /* Supplier's name      */
        ems_uuid_t *          uuid,           /* Supplier's UUID      */
        ems_error_t *         status);        /* mgmt request status  */

/*********************************************************/
/* Get Undelivered Events                                */
/*********************************************************/
extern
void ems_mgmt_get_undelivered_events(
        ems_handle_t           handle,        /* XEMS handle          */
```

```
        ems_event_list_t **    event,         /* undelivered events   */
        ems_error_t *          status);       /* event get status     */

/*********************************************/
/* Free Undelivered Events                    */
/*********************************************/
extern
void ems_mgmt_free_undelivered_events(
        ems_event_list_t **    event,         /* undelivered events   */
        ems_error_t *          status);       /* event get status     */

/*********************************************/
/* Delete Undelivered Events                  */
/*********************************************/
extern
void ems_mgmt_delete_undelivered_event(
        ems_eventid_t *        event_id,      /* event id to delete     */
        ems_error_t *          status);       /* event delete status    */

#endif /* _XEMS_H */
```