Preliminary Specification

Systems Management: Event Management Service

The Open Group

© February 1997, The Open Group

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Preliminary Specification

Systems Management: Event Management Service

ISBN: 1-85912-132-2
Document Number: P437

Published in the U.K. by The Open Group, February 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

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

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

In a world of increasingly complex and distributed computer systems an effective Event Management Service (EMS) is a key part of the necessary systems management and administration infrastructure. The EMS must provide timely warning of impending problems, notify failing processes, identify problem areas in a system and possibly automatically fix them before service availability falls below acceptable levels. To achieve what is required, interoperability between systems in a distributed working environment is only part of the solution; inter-comprehension of the key event data is also necessary.

This Event Management Service (XEMS) specification defines a programming interface which receives notifications in the form of events, and transports them reliably to applications.

The specification is in 3 parts. The first part describes the generic XEMS API, including the model, architecture, positioning of XEMS in the systems management domain, the data formats used by XEMS, and the definitions for the XEMS interfaces (Registration, Event Type, Event Filter, Consumer, Supplier, Administration, Command Line). The second part describes reference implementations for DCE and CORBA environments. The third part describes event structures for the basic event set.

Structure

The structure of this specification is as follows:

- **Part 1: XEMS generic API specification**
  - **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 of the event management service general model. This is followed by an architectural view of XEMS, then a description of the role of XEMS in the systems management domain. The chapter continues with a survey of usage scenarios and deployment strategies, and an introduction to the data formats for XEMS data formats follows this 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, Registration Interface**, defines the registration interface.
  - **Chapter 5, Event Type Interface**, defines the event type interface.
  - **Chapter 6, Event Filter Interface**, defines the event filter interface.
  - **Chapter 7, Consumer Interface**, describes the API and commands available to register with and receive events from an XEMS.
  - **Chapter 8, Supplier Interface**, describes the API and commands available to register with and supply events to an XEMS.
  - **Chapter 9, Administration Interface**, describes the API and the administration commands used to configure and control an XEMS implementation.
  - **Chapter 10, Command Line Interface**, defines a CLI to assist wrapping legacy applications and shell scripts as event suppliers.
  - **Appendix A, <xems.h>**, is the header for XEMS.

- **Part 2: XEMS Implementations in Different Environments**
  - **Chapter 11, Reference Implementations**, explains the intentions behind providing the XEMS reference implementations.
  - **Chapter 12, DCE Implementation**, presents the DCE reference implementation.
  - **Chapter 13, CORBA Implementation**, is a placeholder for the normative CARBA reference implementation.
  - **Appendix B, CORBA Implementation**, presents the existing non-normative CORBA reference implementation.
- **Part 3: XEMS Structures for the Basic Event Set**
  - **Chapter 14, Event Objects**, describes event structures for the basic event set.

Trade Marks

Motif®, OSF/1® and UNIX® are registered trade marks and the ‘‘X Device’’™ and The Open Group™ are trade marks of The Open Group.

Referenced Documents

The following documents are referenced in this specification:

Base GSS-API
  CAE Specification, December 1995, Generic Security Service API (GSS-API) Base (ISBN: 1-85912-131-4, C441).

CORBA 1.2
  CAE Specification, July 1994, The Common Object Request Broker: Architecture and Specification (ISBN: 1-85912-044-X, C432), in conjunction with the Object Management Group (OMG).

COS, Volume 1
  Preliminary Specification, July 1994, Common Object Services, Volume 1 (ISBN: 1-85912-482-2, P432), in conjunction with the Object Management Group (OMG).

DCE Directory
  CAE Specification, December 1994, X/Open DCE: Directory Services (ISBN: 1-85912-078-4, C312).

DCE EMS
  DCE Event Management Service, OSF RFC 67.0, January 1996, R.Cohen (IBM), G Wilson (IBM), Open Software Foundation (The Open Group) Request for Comments, available from URL http://www.opengroup.org/otcgi/llscgi60.

DCE RPC
  CAE Specification, August 1994, X/Open DCE: Remote Procedure Call (ISBN: 1-85912-041-5, C309).

  This specification is now also ISO International Standard ISO/IEC 11578:1996, Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)

GSS-API Extensions
  Snapshot, January 1994, Generic Security Service API (GSS-API) Security Attribute and Delegation Extensions (ISBN: 1-85912-261-1, S307).

Internationalisation Guide
  Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304).

ISO/IEC 10165-2
  ISO/IEC 10165-2: 1992(E), Information Technology — Open Systems Interconnnect — Structure of Management Information: Definition of Management Information.

ISO/IEC 10165-4
  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
  ISO/IEC 10165-5: 1994(E), Information Technology — Open Systems Interconnnect — Structure of Management Information: Generic Management Information.

ISO/IEC 10165-6
  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
: ISO / IEC 10165-7: 1992(E), Information Technology — Open Systems Interconnnect — Structure of Management Information — Part 7: General Relationship Model.

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

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

XCMF-V1
: Preliminary Specification, June 1995, Systems Management: Common Management Facilities, Volume 1 (ISBN: 1-85912-047-4, P421).

XDS, Issue 3
: CAE Specification, May 1996, API to Directory Services (XDS), Issue 3 (ISBN: 1-85912-180-2, C608).

XDSF
: Guide, December 1994, Distributed Security Framework (ISBN: 1-85912-071-7, G410).

XGDMO
: Preliminary Specification, March 1994, Systems Management: GDMO to XOM Translation Algorithm (ISBN: 1-85912-023-7, P319).

XMP
: CAE Specification, March 1994, Systems Management: Management Protocol API (ISBN 1-85912-027-X, C306).

XOM, Issue 3
: CAE Specification, May 1996, OSI-Abstract-Data Manipulation API (XOM), Issue 3 (ISBN: 1-85912-175-6, C607).

Preliminary Specification

Part 1:

Event Management Service (XEMS) API

The Open Group

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 information technology (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), for example, 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 events. 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

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.

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 user interface 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

All categories (types) of 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 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 in the remainder of this subsection.

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 user interface 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 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.

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-balanci ng 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 (that is, implementation independent) API. This API permits source code compatibility across implementations. The API provides functions for consumers, producers, and administrators.

1.3.5 Extensibility

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.

1.3.7 Internationalization

The EMS must be compliant with internationalization (I18n) requirements.

1.3.8 Interoperability

The event management services from different vendors must interoperate.

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.

Part 2 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, for example, 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, for example, 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 the conceptual model shown in Figure 2-1.

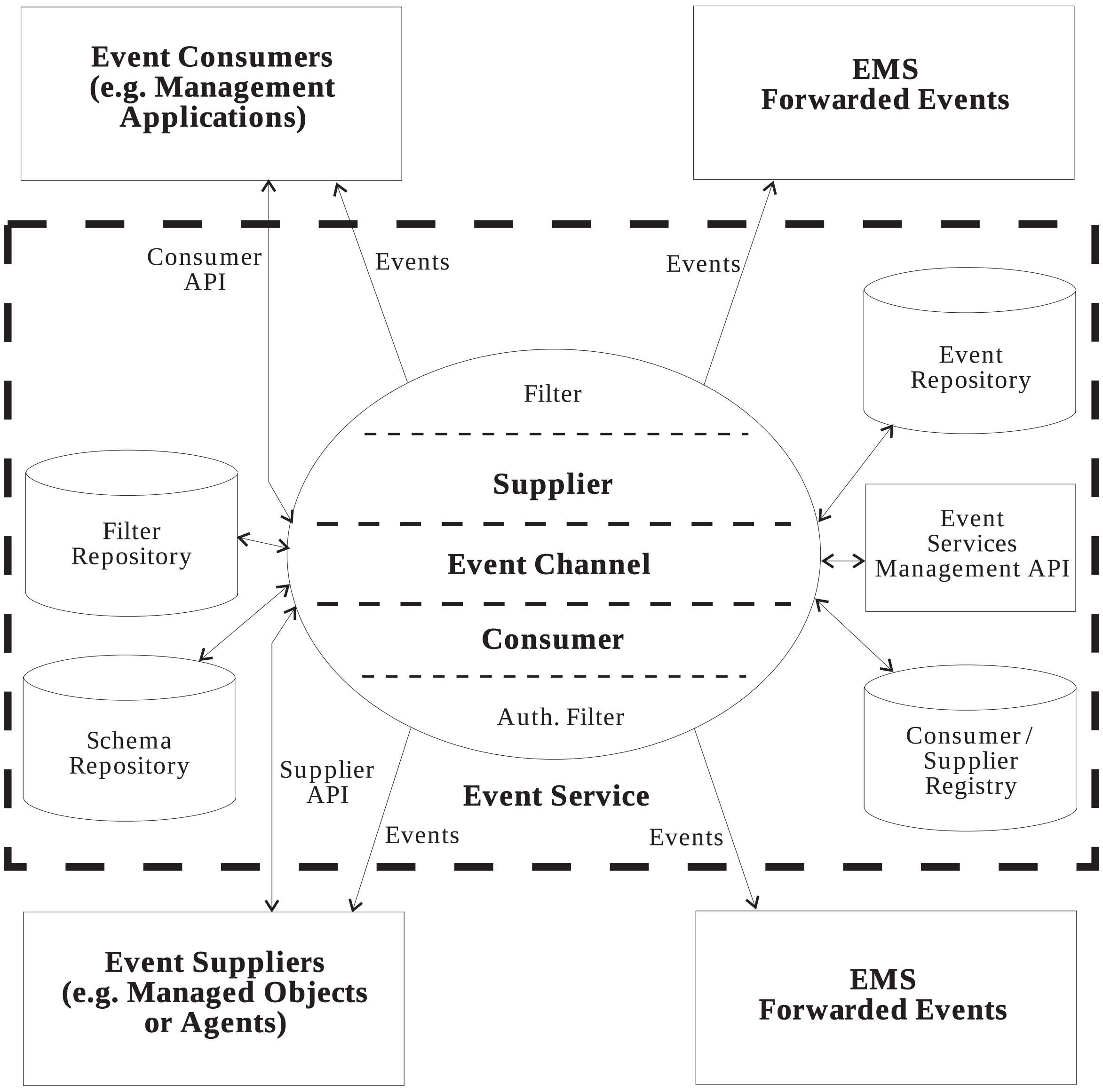


Figure 2-1 Event Management Service: Conceptual Model

At first glance, the core of the Event Service looks very similar to a COSSES Event Channel. There are, however, some notable differences:

- The EMS involves two or more 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 for which they are authorized. The [consumer-side] filter performs two major functions. First, it delineates the events in which the consumer is interested, both in terms of the event type, per se, and in terms of the criteria receiving a given event. Second, it provides a security mechanism, where the consumer can only receive events for which 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 must be a persistent store for reliable event delivery. It 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 must be a persistent store. It 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.
- Once events are collected for a consumer, they will persist across consumer connections, within the space restrictions set for the given XEMS instance. Should the value be exceeded, then events may be lost. The consumer may disconnect without losing events. In fact, events will continue to be gathered for the consumer. When the consumer reconnects, the events will be sent to the consumer.
- The schema repository contains information for typed events.
- The event services manager is responsible for orchestrating the activities of the Event Service.

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

The remaining sections of this chapter describe 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 ‘‘Example’’ section (see Section 2.8 on page 33) 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:

- configuration
- authorization
- registration
- connection
- delivery
- maintenance.

Configuration of an EMS

The configuration of EMS involves loading the package, allocating the repositories, and implementation setup. Subsequently, this EMS must be registered with other EMS instances. This will control the forwarding of events, etc. 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 configuration process will also instantiate the namespace (registry) for the basic event types.

Configuration of a Supplier

The configuration 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. When applying filters, the event channel assumes that all attributes in the event schema have values.

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 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 current h ints that may be received by a supplier are the number of consumers for an event type. This may be used by a supplier to (de)activate sending events to the EMS. Each change in the number of consumers of an event type causes a hint to be sent to the supplier. The hints are reference counts.

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, for example, 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_push_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 /or 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 time administrators may want to audit the repositories and registries. EMS is designed to be 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 than 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
- 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 Registration interface
- the Event Type interface
- the Supplier Interface
- the Filter Interface
- the Consumer Interface
- 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.

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 management service 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.

2.2.2 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.

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.

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*be*a

2.2.3 Schemas

Schemas are required for all typed 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.

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.

2.2.4 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 after a **get**.

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 indica tes to the EMS that it has received the event, the EMS removes the event from the persistent store.

A channel pair provides a means of decoupling consumers and suppliers. 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. 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_push_consumer_register( )
: Register a push consumer.

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.

ems_consumer_get_registration( )
: Retrieve consumer registration information associated with a consumer handle.

ems_consumer_pull( )
: Used by pull consumers to get an event from the event management service.

ems_consumer_try_pull( )
: Used by pull consumers to get an event from the event management service, but not block waiting.

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_supplier_register_handler( )
: Register a hint callback for the supplier. This requires the supplier to be both a client and a server.

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_secedit
: Alter the permissions of a subject with regard to a security object.

ems_mgmt_secsubjadd
: Add a subject to the EMS. The subject may act as a consumer or a supplier or both, based on the read/write permissions given on the call.

ems_mgmt_secsubjdelete
: Remove a subject from the EMS.

ems_mgmt_secsubjget
Given a principal, return the EMS defined subject.

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_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_get_undelivered_events( )
Retrieve a list of events that have not been delivered to interested consumers.

ems_mgmt_free_undelivered_events( )
Free the undelivered events in the list for interested consumers.

ems_mgmt_delete_undelivered_event( )
Delete an undelivered event from the EMS Event Log.

ems_mgmt_forward( )
Establish the forwarding of events described in a filter group from a given EMS to another EMS.

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 (for example, 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 (for example, 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.
- Interoperable across the network
  This attribute addresses the ability event management system products from different vendors to interoperate 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

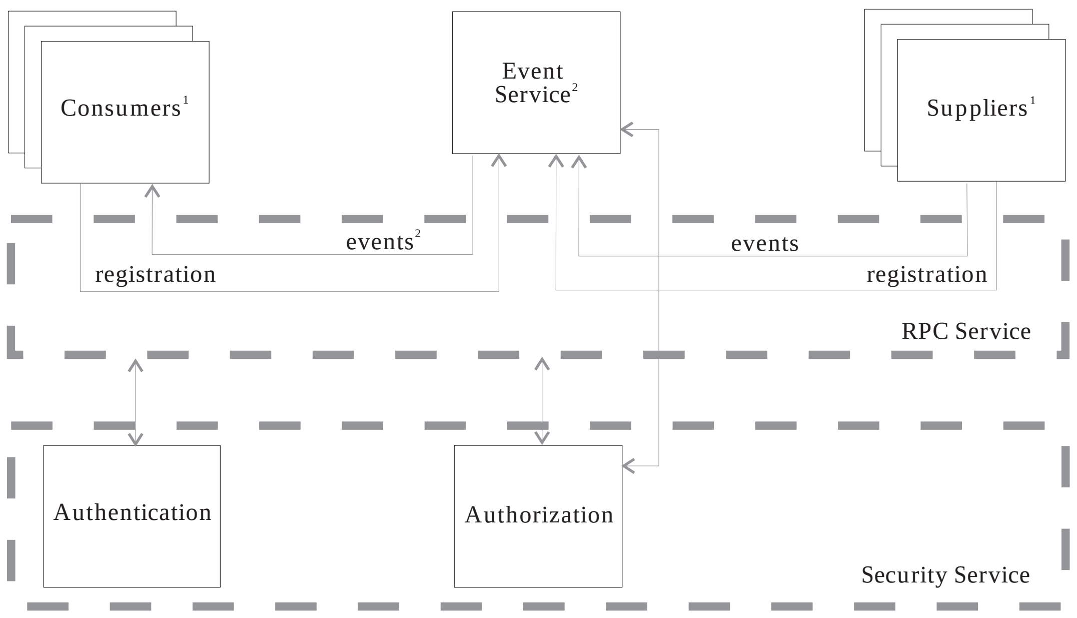
The EMS, optionally, uses security facilities in two distinct means. The EMS can use the security framework of the underlying system to obtain the principal name. The principal name may in turn be used to determine access permissions to EMS constructs, for example, the filter repository or event types. The access permissions to EMS constructs represents a multi-level access security model.

The EMS may use an external security mechanism to control the consumer's or supplier's ability to register (bind) with an EMS. Given a security principal, the EMS may recognize permissions for specific objects. 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 EMS 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 Figure 2-2.



1. The security context is derived from the normal means, for example, 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.

Figure 2-2 Security Service

The conceptual role of security for the DCE-based EMS is summarized as follows:

1. Neither consumers nor suppliers are aware of, that is, 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, that is, 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 (that is, 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.

— "*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 (for example, 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, for example, 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.

The 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 remainder of this section describes 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, for example, 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 (for example, 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, for example, 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 (for example, 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 interoperable. Consumers and suppliers would be capable of connecting to and operating with any conforming EMS. For implementations utilizing the same transport, for example, CORBA V2, the implementations are directly interoperable. For implementations utilizing different transports, they interoperate 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 (for example, SNMP traps).
- Event consumers based on different protocols (for example, 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. See Figure 2-3.

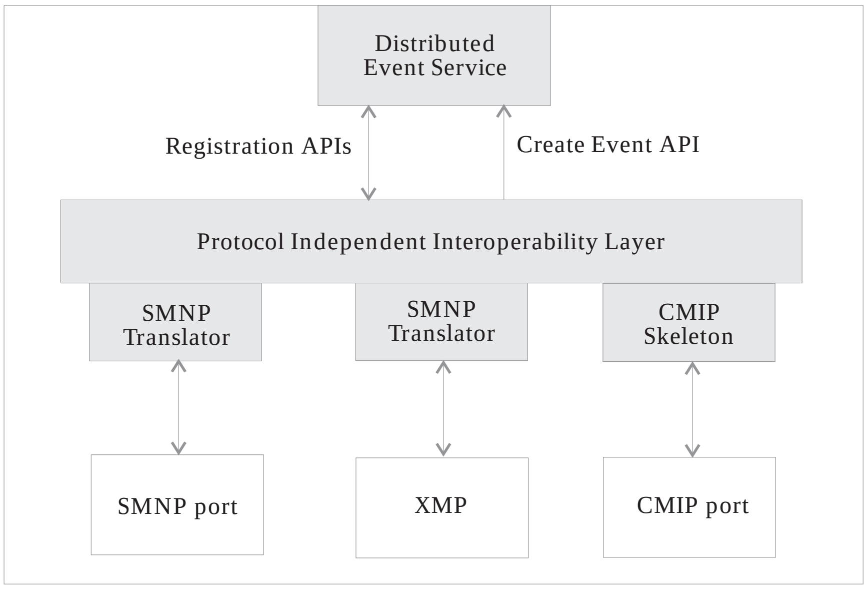


Figure 2-3 Managed Node

The translator modules establish access to the different event worlds by translating the particular event format (for example, 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. Figure 2-4 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.

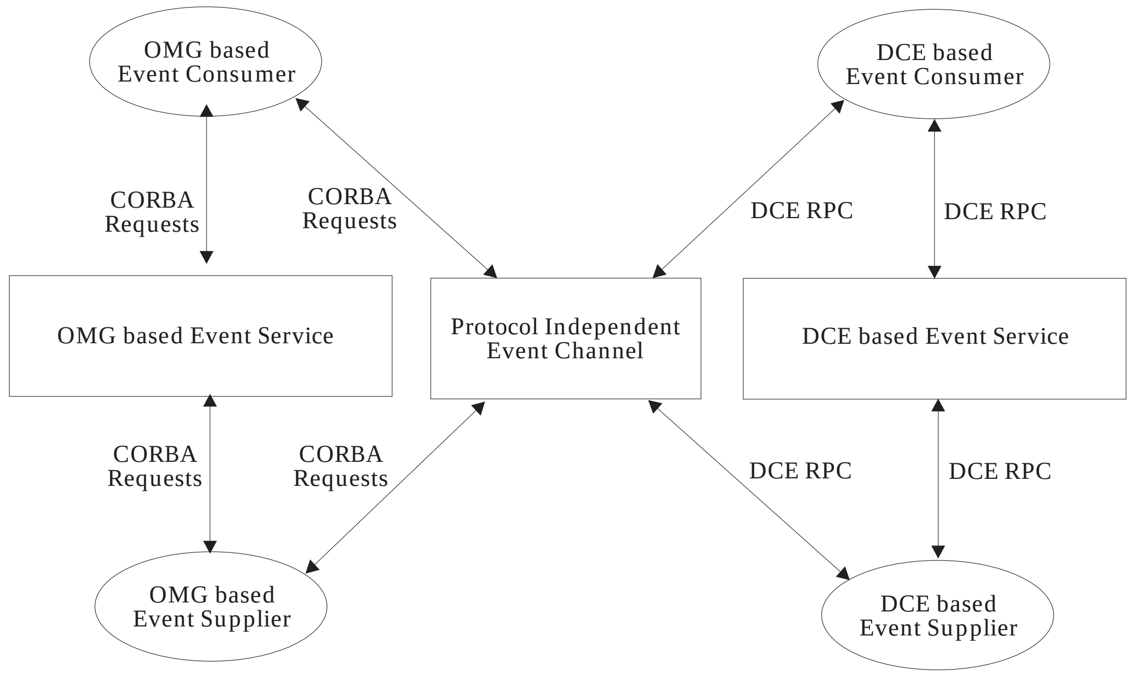


Figure 2-4 Interoperability: Protocol Independent Channel

Figure 2-4 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 Figure 2-4 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 Figure 2-5.

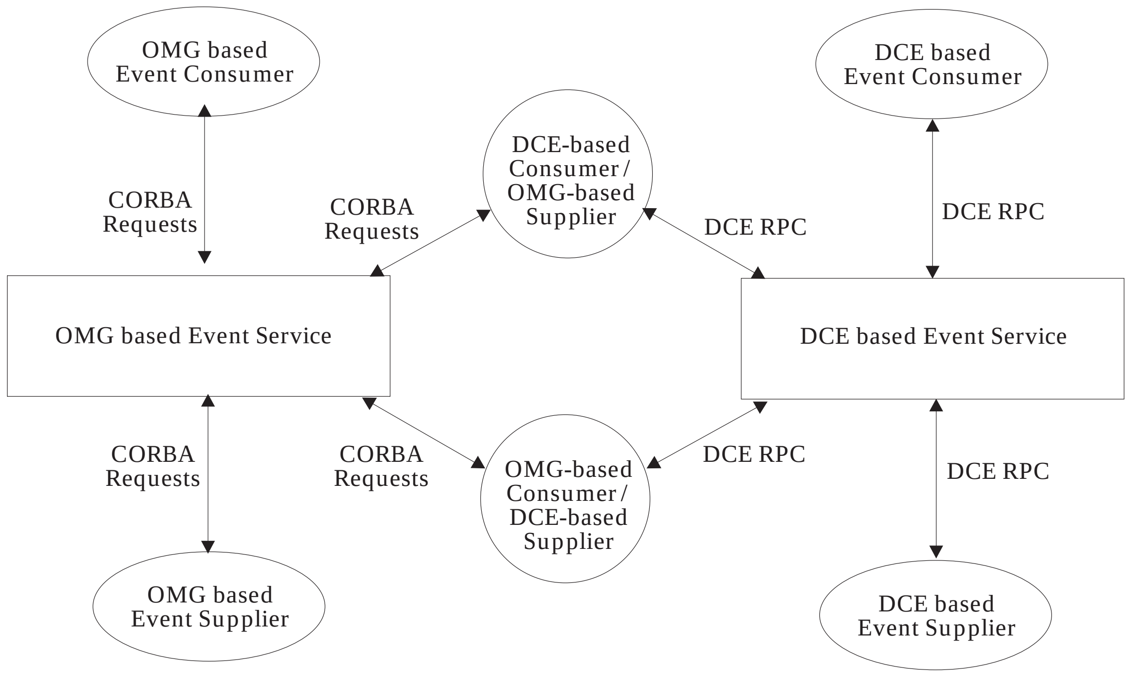


Figure 2-5 Interoperability: Dual Environment Channels

2.8 Examples

An example using the *intelligent agent* serves to demonstrate 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 be presumed to exist on the target node.

The process flow is then:

1. connect with the agent factory
2. set up the transfer of initialization information
3. transfer the agent and applettes
4. set up the supplier with the EMS
5. set up the consumer with the EMS
6. start the supplier
7. send completion status to the consumer with a reference to the EMS conversation.

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.

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.

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 could 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.

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 activates the agent in the context of the directory it previously built. The agent executes with the security credentials and permissions of the consumer.

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, for example, 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.

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.

Conversation Manager Remove Conversation

The consumer normally asks the conversation manager to clean up the agent and perform associated housekeeping tasks 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.

Conversation Manager Remove Consumer

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

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.

Conversation Manager Remove Event Type

Here, the conversation manager is attempting to remove the event type. This method will fail when either there are undelivered events of this type, or the event type does not exist.

Use *ems_event_type_delete*( ) to remove the event type. ,HU "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 effectively terminated by interrupting delivery of events to the consumer.

Chapter 3

Data Formats

The XEMS 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 XEMS. Some of these types define standard C data types and some machine dependant types so that implementations of XEMS 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 XEMS, 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 XEMS 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 XEMS 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 XEMS.

There is also an attribute list data structure which allows getting and setting attributes in XEMS. 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 XEMS routines. An **ems_handle** represents a connection to an Event Service on a particular host. When calling XEMS routines, one of the XEMS registration routines is called to unitize the handle, and it is used in all subsequent calls to XEMS routines that want to affect the Event Service on that host.

3.1 Generic Data Types

This data structure section first defines some generic types that are used throughout the definition of XEMS.

3.1.1 Scalar Types

XEMS defines generic C data types which can be defined per implementation. These types correspond to standard data types[1].

```
typedef unsigned_char          ems_boolean;     // 1 byte
#define ems_false              false
#define ems_true               true
typedef unsigned char          ems_byte;        // 1 byte
typedef unsigned char          ems_char;        // 1 byte
typedef signed char            ems_small_int;   // 1 byte
typedef unsigned char          ems_usmall_int;  // 1 byte
typedef short int              ems_short_int;   // 2 bytes
typedef unsigned short int     ems_ushort_int;  // 2 bytes
typedef long int               ems_long_int;    // 4 bytes
typedef unsigned long int      ems_ulong_int;   // 4 bytes

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; // 4 bytes
typedef double                 ems_long_float;  // 8 bytes
```

3.1.2 Strings

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

```
typedef char *     ems_string_t;
```

1. The underlying transport is responsible for the data representation changes between clients and servers.

3.1.3 Unique Identifier

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

```
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;
```

3.1.4 Time Stamp

The XEMS time structure contains a timestamp represented in Coordinated Universal Time (UTC). This is a 128-bit binary number. It is often referred to as a binary timestamp.

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

3.1.5 Error Status

The XEMS error status is used to return status to callers of XEMS 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 on page 56).

```
typedef ems_ulong_int  ems_error_t;
```

3.1.6 Event Type

XEMS event types are used to classify events.

```
typedef ems_uuid_t     ems_event_type_t;
```

3.1.7 Delivery Type

XEMS delivery model.

```
typedef enum {
    ems_delivery_push = 0,
    ems_delivery_pull
} ems_delivery_t;
```

3.1.8 Security Object

EMS security objects. These security objects are defined in Chapter 2.

```
typedef enum {
    ems_secobj_server = 0,
    ems_secobj_eventtypes,
    ems_secobj_filters,
    ems_secobj_consumers,
    ems_secobj_suppliers,
    ems_secobj_eventtype,
    ems_secobj_filter
} ems_secobjtype_t;

typedef struct {
    ems_secobjtype_t     secobjtype;
    ems_string_t         name;
    ems_uuid_t           uuid;
} ems_secobj_t;
```

3.1.9 Permissions Attributes

EMS permission attributes.

```
typedef struct {
    ems_usmall_int       control;
    ems_usmall_int       delete;
    ems_usmall_int       insert;
    ems_usmall_int       read;
    ems_usmall_int       write;
    ems_usmall_int       execute;
} ems_secperm_t;
```

3.1.10 Subject

EMS subject.

```
typedef struct {
    ems_string_t         name;
    ems_uuid_t           uuid;
} ems_secsubj_t;
```

3.1.11 Principal

EMS principal.

```
typedef struct {
    ems_ushort_int       len;
    ems_byte *           principal;
} ems_secprin_t;
```

3.2 Event Attributes

3.2.1 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:

Attribute Type	**Data Type**
ems_c_attr_small_int	**ems_small_int**
ems_c_attr_short_int	**ems_short_int**
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**

Table 3-1 Event Attribute Type Specifiers

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

The actual attribute format type values are:

```
#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)
```

3.2.2 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;
```

3.2.3 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 Event Structure

3.3.1 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;
```

3.3.2 Event Type

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

Event Type	Event Type Name
ems_c_generic_type	Generic

Table 3-2 Default Event Types

Events of type Generic, do not have event type schemas associated with them, and can only be filtered by expressions with header attributes in them (see Table 3-3 on page 47).

3.3.3 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. Structure **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 );
```

3.3.4 Event Origin

The event origin specifies where the event originated (that is, 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;
```

3.3.5 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;
```

3.3.6 Event Priority

The event priority specifies the priority of the event.

```
typedef ems_ulong_int    ems_priority_t;
```

3.3.7 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 XEMS, 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.

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**
received.wday	**ems_c_attr_ushort_int**
received.yday	**ems_c_attr_ushort_int**

Table 3-3 Event Header Attributes

3.3.8 Event

The **ems_event_t** structure contains a fixed part, the event header, and a variable part, the event data items. 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 Event Types

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

3.4.1 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;
```

3.4.2 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 Event Filters

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

3.5.1 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;
```

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*

Table 3-4 Attribute Operators

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)
```

3.5.2 Event Filter Grammar

The event filter grammar specifies which grammar the event filter is using to specify a filter expression. Support for the default grammar is required.

```
typedef unsigned16 ems_filter_grammar_t;
const ems_filter_grammar_t ems_c_fg_default   =  0;
const ems_filter_grammar_t ems_c_fg_OQL       =  1;
const ems_filter_grammar_t ems_c_fg_other     =  2;
```

OQL stands for Object Query Language.

3.5.3 Default Event Filter Grammar

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

```
typedef struct ems_default_fg_s_t  {
    ems_string_t        attr_name;
    ems_attr_op_t       attr_operator;
    ems_attr_value_t    attr_value;
} ems_default_fg_t;
```

3.5.4 Event Filter Expression

The event filter expression structure contains an event filter expression. This structure is a tagged union whose type (or tag) defines which grammar the filter expression is using, and that value is the filter expression itself.

```
typedef struct ems_filter_exp_s_t  {
    ems_filter_grammar_t    grammar;
    union  {
        /* case: ems_c_fg_default */
        ems_default_fg_t    def_filter;
        /* case: ems_c_fg_OQL */
        ems_string_t        oql_filter;
        /* case: ems_c_fg_other */
        ems_string_t        other_filter;
    } tagged_union;
} ems_filter_exp_t;
```

3.5.5 Event Filter Expression List

An event filter expression list groups a list of filter expressions together in a list to form an anded 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;
```

3.5.6 Event Filter

An event filter specifies a series of event filter expressions that will be anded 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 3-3 on page 47).

```
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;
```

3.5.7 Event Filter Name List

An event filter list contains a list of size event_filter_names;

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

3.5.8 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 Consumer Data Structures

3.6.1 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_delivery_t        type;
} ems_consumer_t;
```

3.6.2 Consumer List

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

```
typedef struct ems_consumer_list_s_t {
    ems_long_list         size;
    ems_consumer_t        consumer[1];
} ems_consumer_list_t;
```

3.6.3 Event Handler

The consumer provides a set of event handler functions in the ems_consumer_start method. These routines are callbacks. They are associated with hosts and event filter groups through the ems_push_consumer_register method.

```
typedef void (*ems_handler)(void * arg, ems_event_t * event,
              ems_error_t * error);
```

3.7 Supplier Data Structures

3.7.1 Supplier Event Handler

The supplier may provide a handler function (the *ems_push_supplier_register_handler*( ) method. This routine is a callback. It is called each time the number of consumers of the event type changes. This allows the supplier to gauge the need to create and send events to the XEMS.

```
typedef void (*ems_supplier_count_handler_t)(ems_event_type_t type,
                           ems_long_int count, ems_error_t * error);
```

3.7.2 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_delivery_t         type;
} ems_supplier_t;
```

3.7.3 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 Attribute and Event list

3.8.1 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;
```

3.8.2 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 Event Service Handle

An ems_handle represents a connection to an Event Service on a particular host. When calling XEMS routines, one of the XEMS 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.

3.9.1 Event Service Handle

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

```
typedef void *ems_handle_t;
```

3.10 Status Codes

All XEMS routines return status codes which contain values which indicate whether the call to that routine was successful or not.

Rather than list the specific status codes for each routine, the following summary lists all the status codes and their meanings.

ems_s_already_registered
: Consumer with this name is already registered.

ems_s_consumer_already_started
: Consumer already started.

ems_s_consumer_not_started
: Consumer not started.

ems_s_empty_filter_db
: The listed filters could not be returned because the filter database is empty.

ems_s_event_type_exists
: The event type to be added already exists.

ems_s_event_type_not_found
: The specified event type was not found.

ems_s_filter_exists
: The given filter name already exists.

ems_s_filter_in_use
: The filter cannot be deleted because it is currently in use.

ems_s_filter_not_found
: The requested filter does not exist.

ems_s_forwarding_event_service_not_there
: The event service to forward to is not available.

ems_s_forwarding_event_loop
: The hostname introduces a loop condition, where XEMS would be forwarding events to itself.

ems_s_insufficient_permission
: Caller does not have sufficient permission to perform operation.

ems_s_invalid_event_type
: The schema for the event type is not valid.

ems_s_invalid_filter
: The input parameters specifies an invalid filter.

ems_s_invalid_handle
: The handle parameter is not valid.

ems_s_invalid_name
: The name parameter contains invalid characters.

ems_s_no_consumers
: No consumers are registered.

ems_s_no_event
: Tried to pull an event of a specified type, but there are no events to pull.

ems_s_no_events
: There are no undelivered events.

ems_s_no_memory
: An XEMS handle cannot be allocated.

ems_s_no_suppliers
: No suppliers are registered.

ems_s_no_type_list
: There was no type list in the function invocation.

ems_s_status_ok
: Success.

ems_s_unknown_consumer
: Tried to unregister a consumer that was not registered.

ems_s_unknown_supplier
: Tried to unregister a supplier that was not registered.

ems_s_unsupported_nameservice
: Unsupported nameservice on host name.

Chapter 4

Registration Interface

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

NAME

ems_register — register with XEMS

SYNOPSIS

```
#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
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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_unsupported_nameservice

NAME

ems_unregister — unregister with XEMS.

SYNOPSIS

```
#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 XEMS 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.

RETURN VALUE

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.

NAME

ems_event_type_add — Add an Event Type.

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_handle
ems_s_event_type_exists
ems_s_insufficient_permission
ems_s_invalid_event_type

NAME

ems_event_type_delete — Delete an Event Type

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_event_type_get — Get an Event Type

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_event_type_get_list — Get Event Types List

SYNOPSIS

```
#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 *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.

RETURN VALUE

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

NAME

ems_event_type_free_list — Free Event Types List

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

Chapter 6

Event Filter Interface

Event filters can be created using the Event Filter Interface.

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

NAME

ems_filter_add — Add an Event Filter

SYNOPSIS

```
#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 XEMS 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.

RETURN VALUE

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

NAME

ems_filter_append — Append to an Event Filter

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_filter_get — Get an Event Filter

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_filter_delete — Delete an Event Filter

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_filter_free — Free an Event Filter

SYNOPSIS

```
#include <xems.h>

void ems_filter_free(
    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

exp_list

the list of filter expressions which are part 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_filter_get_namelist — List Event Filter Names

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_insufficient_permission
ems_s_invalid_handle
ems_s_empty_filter_db

NAME

ems_filter_free_namelist — Free Event Filter Names

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_filter_get_list — Get Event Filter List

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_insufficient_permission
ems_s_invalid_handle
ems_s_empty_filter_db

NAME

ems_filter_free_list — Free Event Filter List

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

Chapter 7

Consumer Interface

The XEMS 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 XEMS.

All event consumers have to make calls to the XEMS event consumer setup routines before receiving XEMS events. These routines perform required setup, and are XEMS implementation dependent. The setup routines are designed to work with the register routines to save the state of a consumer environment so that the consumer can be restarted with a call to the *ems_consumer_start*( ) routine by passing the consumer *uuid* obtained from the initial call to *ems_consumer_start*( ).

The event consumer interface allows event consumers to register and unregister with XEMS. Once registered, consumers can add and delete event filters define what events they are interested in. When XEMS 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.

NAME

ems_consumer_start — Consumer Start

SYNOPSIS

```
#include <xems.h>

void ems_consumer_start(
    ems_string_t        consumer,
    ems_ulong_int       flags,
    ems_handler_t       hfunc[],
    ems_uuid_t **       uuid,
    ems_handle_t *      handle[],
    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 a **ems_uuid_t** to uniquely identify this event consumer and perform any local consumer initialization required. The routine can be called the first time a consumer starts, or when a consumer is restarting and wishes to reestablish the environment already established. If a new environment is being established, then the uuid parameter should contain NULL. When reestablishing an environment, then the uuid from the initial call should be passed in.

PARAMETERS

Input

consumer
specifies the consumer name. This parameter can be null if the *uuid* is specified.

flags
reserved for future use.

hfunc
null terminated array of event handler routines. All event handler routines that will be used in an *ems_consumer_register*( ) call must be in this array.

uuid
the unique consumer id returned from a previous call to *ems_consumer_start*( ).

Output

uuid
the unique consumer id for this consumer environment.

handle
returns an XEMS handle which can be used on subsequent calls to XEMS routines. This handle also represents a consumer filter_group /event handler association.

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

RETURN VALUE

The possible ems status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_consumer_already_started

NAME

ems_consumer_stop — Consumer Stop

SYNOPSIS

```
#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.

RETURN VALUE

The possible XEMS status codes are:

ems_s_status_ok
ems_s_consumer_not_started

NAME

ems_push_consumer_register — Push Consumer Register

SYNOPSIS

```
#include <xems.h>

void ems_push_consumer_register(
    ems_netname_t *              hostname,
    ems_filtername_list_t *      filter_group,
    int                          hfunc_index,
    ems_handle_t *               handle,
    void *                       arg,
    ems_error_t *                status);
```

DESCRIPTION

This routine is used by XEMS event consumers to register as a push consumer with XEMS. This routine contacts an event service, and registers this filter group/handler association with that event service. This routine may be called multiple times per consumer with different hostnames, filter_groups, and handler functions. The handler function specified is started up the in the consumer process. *ems_consumer_start*( ) has to have been called before this routine (to establish the consumer name and uuid).

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 XEMS will not forward any events to this consumers until the consumer makes a call to *ems_add_event_to_group*( ).

hfunc_index
the index into the event handler array of the event handler function to call if an event passes the filter group and is sent to the consumer. The event handler array is the *hfunc[]* parameter to the *ems_consumer_start*( ) routine.

Output

handle
returns an XEMS handle which can be used on subsequent calls to XEMS routines. This handle also represents a consumer filter_group/event handler association.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_already_registered

NAME

ems_pull_consumer_register — Pull Consumer Register

SYNOPSIS

```
#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 XEMS event consumers to register as a pull consumer with XEMS. 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 XEMS will not forward any events to this consumers until the consumer makes a call to *ems_add_event_to_group*( ).

Output

handle
returns an XEMS handle which can be used on subsequent calls to XEMS routines.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_already_registered

NAME

ems_consumer_unregister — Consumer Unregister

SYNOPSIS

```
#include <xems.h>

void ems_consumer_unregister(
    ems_handle_t *          handle,
    ems_error_t *           status);
```

DESCRIPTION

This routine is used by XEMS event consumers to unregister with XEMS. This routine should be called once for each call to *ems_push_consumer_register*( ) 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_unknown_consumer

NAME

ems_add_filter_to_group — Add Event Filter to Group

SYNOPSIS

```
#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 XEMS 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_delete_filter_from_group — Delete Event Filter From Group

SYNOPSIS

```
#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 XEMS 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_get_filter_group — Get Filter Group

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_consumer_get_registration — Get Consumer Registration

SYNOPSIS

```
#include <xems.h>

void ems_consumer_get_registration(
    ems_handle_t                   handle,
    ems_netname_t **               hostname,
    ems_filtername_list_t **       filter_group,
    int*                           hfunc_index,
    ems_error_t *                  status);
```

DESCRIPTION

This routine returns the consumer registration information associated with a consumer handle.

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 event filter group which is associated with this consumer registration handle. It is up to the caller to free the storage allocated for *filter_group*.

hostname
will contain the hostname of the event service associated with this consumer registration handle. It is up to the caller to free the storage allocated for hostname.

hfunc_index
will contain the handler function index associated with this consumer registration handle. If the consumer is a *pull* consumer, then this value will be -1.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_consumer_pull — Consumer Pull

SYNOPSIS

```
#include <xems.h>

void ems_consumer_pull(
    ems_handle_t            handle,
    ems_event_t *           event,
    ems_error_t *           status);
```

DESCRIPTION

This routine is called by pull consumers to get an event from the event service. The event has to pass through the filter group set up by the pull consumer in order to receive the event. This routine does not return until an event is available.

PARAMETERS

Input

handle
must contain a valid consumer handle obtained from *ems_pull_consumer_register*( ).

Output

event
will contain the event received from the pull operation from the event service.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_consumer_try_pull — Consumer Try Pull

SYNOPSIS

```
#include <xems.h>

void ems_consumer_try_pull(
    ems_handle_t            handle,
    ems_event_t *           event,
    ems_error_t *           status);
```

DESCRIPTION

This routine is called by pull consumers to get an event from the event service. The event has to pass through the filter group set up by the pull consumer in order to receive the event. This routine returns with a status of *ems_s_no_event*( ) when no event is available.

PARAMETERS

Input

handle
must contain a valid consumer handle obtained from *ems_pull_consumer_register*( ).

Output

event
will contain the event received from the pull operation from the event service.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_event

Chapter 8

Supplier Interface

The supplier interface provides a mechanism for managed objects to convey events to the XEMS.

NAME

ems_push_supplier_register ( )
Push Supplier Register

SYNOPSIS

```
#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 XEMS event suppliers to register with XEMS. 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 XEMS handle which can be used on subsequent calls to XEMS routines.

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

RETURN VALUE

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

NAME

ems_supplier_register_handler — Supplier Register Handler

SYNOPSIS

```
#include <xems.h>

void ems_supplier_register_handler(
    ems_event_type_t                type,
    ems_supplier_count_handler_t    handler,
    ems_handle_t *                  handle,
    ems_error_t *                   status);
```

DESCRIPTION

This routine is used by XEMS event suppliers to register per event type handlers with XEMS. This routine should be called once for each event type that this supplier wants to provide hints for.

PARAMETERS

Input

type
: is the type of event to associate with the handler.

handler
: is the callback method to be invoked with XEMS consumer count information.

handle
: an XEMS handle.

Output

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

RETURN VALUE

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

NAME

ems_pull_supplier_register — Pull Supplier Register

SYNOPSIS

```
#include <xems.h>

void ems_pull_supplier_register(
    ems_netname_t *         hostname,
    ems_ushort_int          interval,
    ems_handle_t *          handle,
    ems_error_t *           status);
```

DESCRIPTION

This routine is used by XEMS event suppliers to register with XEMS. 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.

interval
is the suggested polling interval in seconds. This represents the interval that XEMS should use to get events from the supplier. An interval of 0 is a hint to the XEMS to use *pull* rather than *try-pull* semantics.

Output

handle
returns an XEMS handle which can be used on subsequent calls to XEMS routines.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_already_registered
ems_s_unsupported_nameservice

NAME

ems_supplier_unregister — Supplier Unregister

SYNOPSIS

```
#include <xems.h>

void ems_supplier_unregister(
    ems_handle_t *          handle,
    ems_error_t *           status);
```

DESCRIPTION

This routine is used by XEMS event suppliers to unregister with XEMS. 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_unknown_supplier
ems_s_invalid_handle

NAME

ems_supplier_send — Supplier Send

SYNOPSIS

```
#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.

RETURN VALUE

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

NAME

ems_mgmt_list_ems — List Event Service Hosts

SYNOPSIS

```
#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 .Fn 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_free_ems — Free Event Service Host List

SYNOPSIS

```
#include <xems.h>

void 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_list_attributes — List Event Service Attributes

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_free_attributes — Free Event Service Attributes

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_list_consumers — Management List Consumers

SYNOPSIS

```
#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 XEMS.

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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_no_consumers

NAME

ems_mgmt_free_consumers — Management Free Consumers List

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_secedit — Management Security Edit

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_secedit(
    ems_handle_t          handle,
    ems_secobj_t          secobj,
    ems_secsubj_t         subject,
    ems_secperm_t         newperm,
    ems_secperm_t *       oldperm,
    ems_error_t *         status);
```

DESCRIPTION

Alters the permission attributes for a subject, that is, a principal or client with regard to an XEMS object. The effect of an edit operation may be to permit a subject to use an EMS object. The effect of an edit operation may be to revoke permission to use an EMS object. If the subject is not associated with the security object, then the addition of permissions will instantiate the subject for the security object. The removal of all permissions will remove the subject from the security object.

PARAMETERS

Input

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

secobj
specifies the targeted security object.

subject
specifies the subject involved in the edit operation.

newperm
specifies the permissions to be applied to the subject.

Output

oldperm
returns the permissions associated with the subject before the edit operation.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
ems_s_insufficient_permission

NAME

ems_mgmt_secread — Management Security Read

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_secread(
    ems_handle_t          handle,
    ems_secobj_t          secobj,
    ems_secsubj_t         subject,
    ems_secperm_t *       oldperm,
    ems_error_t *         status);
```

DESCRIPTION

Retrieves the permission attributes for a subject, that is, a principal or client with regard to an XEMS.

PARAMETERS

Input

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

secobj
specifies the targeted security object.

subject
specifies the subject involved in the edit operation.

Output

oldperm
returns the permissions associated with the subject.

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
ems_s_insufficient_permission

NAME ems_mgmt_secsubjadd — Management Security Add Subject

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_secsubjadd(
    ems_handle_t          handle,
    ems_secsubj_t         subject,
    ems_secprin_t         principal,
    ems_error_t *         status);
```

DESCRIPTION:

Identifies a principal as an XEMS subject.

PARAMETERS

Input

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

subject
: specifies the subject involved in the edit operation.

principal
: specifies an opaque identifier that represents the principal from the system perspective.

Output

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
ems_s_insufficient_permission

NAME

ems_mgmt_secsubjdelete — Management Security Delete Subject

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_secsubjdelete(
    ems_handle_t          handle,
    ems_secsubj_t         subject,
    ems_error_t *         status);
```

DESCRIPTION

Identifies a principal as an XEMS subject.

PARAMETERS

Input

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

subject
specifies the subject involved in the edit operation.

Output

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
not valid
ems_s_insufficient_permission

NAME

ems_mgmt_secsubjget — Management Security Get Subject

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_secsubjget(
    ems_handle_t          handle,
    ems_secprin_t         principal,
    ems_secsubj_t *       subject,
    ems_error_t *         status);
```

DESCRIPTION

Identifies a principal as an XEMS subject.

PARAMETERS

Input

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

principal
specifies an opaque identifier that represents the principal from the system perspective.

subject
returns the corresponding subject.

Output

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
ems_s_insufficient_permission

NAME

ems_mgmt_delete_consumer — Management Delete Consumer

SYNOPSIS

```
#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, that is, principal, from the XEMS consumer database. After this call, the specified consumer will not receive any events unless it reregisters with the Event Service. The principal may not be the same as a consumer, for example, the principal may be a group (dbadmin) and the consumer is a member of the group.

Implementations may provide an alternate mechanism, for example, command line or a global security mechanism, for removing principals. The alternate mechanism may be part of a global security scheme.

PARAMETERS

Input

handle
must contain a valid consumer handle obtained from routine.

consumer
specifies the consumer, that is, principal, 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_invalid_name
ems_s_insufficient_permission

NAME

ems_mgmt_delete_filter_from_group — Management Delete Event Filter From Group

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_delete_filter_from_group(
    ems_handle_t                handle,
    ems_string_t                consumer,
    ems_uuid_t *                uuid,
    int                         hfunc_index,
    ems_filtername_list_t *     filter_names,
    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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_add_filter_to_group — Management Add Event Filter to Group

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_add_filter_to_group(
    ems_handle_t                 handle,
    ems_string_t                 consumer,
    ems_uuid_t *                 uuid,
    int                          hfunc_index,
    ems_filtername_list_t *      filter_names,
    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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_get_filter_group — Management Get Filter Group

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_get_filter_group(
    ems_handle_t                  handle,
    ems_string_t                  consumer,
    ems_uuid_t *                  uuid,
    int                           hfunc_index,
    ems_filtername_list_t **      filter_names,
    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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_list_suppliers — Management List Suppliers

SYNOPSIS

```
#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 XEMS.

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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_no_suppliers

NAME

ems_mgmt_free_suppliers — Management Free Suppliers List

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_delete_supplier — Management Delete Supplier

SYNOPSIS

```
#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 XEMS about the specified supplier. The principal may not be the same as a supplier, for example, the principal may be a group (dbadmin) and the supplier is a member of the group.

Implementations may provide an alternate mechanism, for example, command line or a global security mechanism, for adding principals. The alternate mechanism may be part of a global security scheme.

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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_get_undelivered_events — Management Get Undelivered Events

SYNOPSIS

```
#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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_no_events

NAME

ems_mgmt_free_undelivered_events — Management Free Undelivered Events

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_free_undelivered_events(
    ems_event_list_t **    event,     /* undelivered events     */
    ems_error_t *          status);   /* event get status       */
```

DESCRIPTION

Free the undelivered events for the interested consumer.

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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok

NAME

ems_mgmt_delete_undelivered_event — Management Delete Undelivered Event

SYNOPSIS

```
#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 XEMS 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.

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_no_memory
ems_s_no_events

NAME

ems_mgmt_forward — Management Forward Events

SYNOPSIS

```
#include <xems.h>

void ems_mgmt_forward(
    ems_handle_t                 handle,
    ems_filtername_list_t *      filter_group,
    ems_netname_t *              hostname,
    ems_string_t *               name,
    ems_uuid_t *                 uuid,
    ems_error_t *                status);
```

DESCRIPTION

This call tells the XEMS identified by *ems_handle* to forward all events that pass through the *filter_group* specified to the XEMS specified by *hostname*.

Once this call is made, then the event service identified as hostname will be treated like any other consumer, and the filter group can be manipulated by the XEMS management filter group routines.

Forwarding can be stopped by using the *ems_mgmt_consumer_delete*( ).

PARAMETERS

Input

handle

ems handle of the event service that is being asked to forward events. Handle must contain a valid consumer handle obtained from the *ems_register*( ) routine.

filter_group

is a list of event filter names which will define the event filter group that controls which events will be forwarded. If filter_group is empty, no filter group is specified, and XEMS will not forward any events to the specified event service until a call is make to *ems_mgmt_add_filter_to_group*( ).

hostname

is the name of the host machine where the Event Service is running that will receive the forwarded events. If the hostname is NULL, then the local host is assumed. This hostname cannot be the same as the host referred to in the handle.

Output

name

returns the consumer name associated with the event service that the events will be forwarded to. This name can be used in calls to the *ems_mgmt_XXX_filter_group*( ) routines so that the event service can be treated as any other consumer.

uuid

returns the consumer uuid which uniquely identifies the event service to forward to so that it can be treated like as any other event consumer.

status

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

RETURN VALUE

The possible status codes are:

ems_s_status_ok
ems_s_forwarding_event_service_not_there
ems_s_forwarding_event_loop

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 Management Service 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.

NAME

Supplier Send

SYNOPSIS

```
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[2]:

— other
— dns
— dce
— x500
— nis
— sna.

Default value for this parameter is the local node[3].

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.

severity

severity of the event; one of[4]:

— info
— emergency
— alert
— critical
— warning
— notice
— debug.

2. The keywords that are allowed for *service* match the value of XES_NameService.
3. It is necessary that both service and address can be determined on a node.
4. The keywords that are allowed for *severity* match the values of XES_Severity.

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 EMS Command Line Interface

The Event Management Service 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 objects defined are:

- **ems**
- **emsconsumer**
- **emssupplier**
- **emsfilter**
- **emsevent-type**
- **emslog**.

The management objects and command line syntax are defined in the remainder of this Chapter.

NAME

Event Service Object

SYNOPSIS

```
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 (for example, 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.

NAME

Consumer Object

SYNOPSIS

```
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.

NAME

Supplier Object

SYNOPSIS

```
emssupplier catalog [-host hostname]
emssupplier show <supplier_name> [-uuid supplier_uuid][-host hostname]
emssupplier delete <supplier_name> [-uuid supplier_uuid][-host hostname]
```

DESCRIPTION

This command is used to manage registered suppliers in the EMS Supplier Database. The **emssupplier** object represents an EMS supplier.

OPERATIONS

catalog
returns a list registered suppliers in the supplier Database.

show
shows the filter group for supplier *supplier_name*.

delete
delete supplier *supplier_name* from the supplier Database.

modify
modify supplier *supplier_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.

supplier_name
specifies the name of the supplier to perform the operation on.

-*uuid supplier_uuid*
specifies the suppliers unique identifier to uniquely identify *supplier_name*.

NAME

Filter Object

SYNOPSIS

```
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.

NAME

Event Type Object

SYNOPSIS

```
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.

NAME

Event Log Object

SYNOPSIS

```
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.

Appendix A

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  {
    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 delivery types                            */
/************************************************/
typedef enum {
    ems_delivery_push=0,
    ems_delivery_pull
} ems_delivery_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;
s_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 grammars            */
/***********************************************/
typedef unsigned16 ems_filter_grammar_t;
const ems_filter_grammar_t ems_c_fg_default = 0;
const ems_filter_grammar_t ems_c_fg_OQL     = 1;
const ems_filter_grammar_t ems_c_fg_other   = 2;

typedef struct ems_default_fg_s_t {
    ems_string_t           attr_name;
    ems_attr_op_t          attr_operator;
    ems_attr_value_t       attr_value;
} ems_default_fg_t;

/***********************************************/
/* XEMS filter expression                      */
/***********************************************/
typedef struct ems_filter_exp_s_t {
    ems_filter_grammar_t    grammar;
    union  {
        /* case: ems_c_fg_default */
        ems_default_fg_t    def_filter;
        /* case: ems_c_fg_OQL */
        ems_string_t        oql_filter;
        /* case: ems_c_fg_other */
        ems_string_t        other_filter;
    } tagged_union;
} 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_delivery_t              type;
} ems_consumer_t;

typedef struct ems_consumer_list_s_t {
    ems_long_int           size;
    ems_consumer_t         consumer[1];
} ems_consumer_list_t;

/*************************************************/
/* Event Handler                                 */
/*************************************************/
typedef void (*ems_handler_t)(
    ems_event_t * event,
    ems_error_t * error);

/*************************************************/
/* XEMS supplier structures                      */
/*************************************************/
typedef void (*ems_supplier_count_handler_t)(
    ems_event_type_t type,
    ems_long_int count,
    ems_error_t * error);
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 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_ushort_int      interval;      /* Recommended poll interval*/
    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        */

/**********************************************/
/* Push Supplier Register Handler               */
/**********************************************/
extern
void ems_supplier_register_handler(
    ems_event_type_t             type,   /* event type for handler */
    ems_supplier_count_handler_t handler,   /* handler function    */
    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_handler_t         hfunc[],    /* handler functions         */
    ems_uuid_t **         uuid,       /* unique consumer id        */
    ems_handle_t *        handle[],   /* array of consumer handles */
    ems_error_t *         status);    /* start status              */

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

```
/* Consumer Stop                                */
/************************************************/
extern
void ems_consumer_stop(
    ems_error_t *         status);           /* stop 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 */
    int                  hfunc_index, /* index of handler function */
    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 */

/***********************************************/
/* Get Consumer Registration                    */
/***********************************************/
extern
void ems_consumer_get_registration(
    ems_handle_t             handle,         /* XEMS handle           */
    ems_netname_t **         hostname,   /* hostname of the assoc EMS */
    ems_filtername_list_t ** filter_group,   /* Event Filter Group    */
    int *                    hfunc_index, /* associated handler index */
    ems_error_t *            status);        /* filter request status */

/***********************************************/
/* Get Consumer Pull                            */
/***********************************************/
extern
void ems_consumer_pull(
    ems_handle_t             handle,         /* XEMS handle           */
    ems_event_t *            event,          /* received event        */
    ems_error_t *            status);        /* filter request status */
/***********************************************/
/* Get Consumer tRY Pull                        */
/***********************************************/
extern
void ems_consumer_try_pull(
    ems_handle_t             handle,         /* XEMS handle           */
    ems_event_t *            event,          /* received event        */
    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     */

/************************************************/
/* Managaement Security Edit                   */
/************************************************/
extern
void ems_mgmt_secedit(
    ems_handle_t          handle,
    ems_secobj_t          secobj,
    ems_secsubj_t         subject,
    ems_secperm_t         newperm,
    ems_secperm_t *       oldperm,
    ems_error_t *         status);

/************************************************/
/* Managaement Security Read                    */
/************************************************/
extern
void ems_mgmt_secread(
    ems_handle_t          handle,
    ems_secobj_t          secobj,
    ems_secsubj_t         subject,
    ems_secperm_t *       oldperm,
    ems_error_t *         status);

/************************************************/
/* Managaement Security Add Subject             */
/************************************************/
extern
void ems_mgmt_secsubjadd(
    ems_handle_t          handle,
    ems_secsubj_t         subject,
    ems_secprin_t         principal,
    ems_error_t *         status);

/************************************************/
/* Managaement Security Delete Subject          */
```

```
/************************************************/
extern
void ems_mgmt_secsubjdelete(
    ems_handle_t           handle,
    ems_secsubj_t          subject,
    ems_error_t *          status);

/************************************************/
/* Managaement Security Get Subject             */
/************************************************/
extern
void ems_mgmt_secsubjget(
    ems_handle_t           handle,
    ems_secprin_t          principal,
    ems_secsubj_t *        subject,
    ems_error_t *          status);

/************************************************/
/* Add consumer to XEMS                         */
/************************************************/
extern
void ems_mgmt_add_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   */

/************************************************/
/* Add consumer of event to XEMS                */
/************************************************/
extern
void ems_mgmt_add_consumer_of_event(
    ems_handle_t           handle,           /* XEMS handle            */
    ems_string_t           consumer,         /* Consumer’s name        */
    ems_uuid_t *           uuid,             /* Consumer UUID          */
    ems_event_type_t       type,             /* associated event type */
    ems_error_t *          status);          /* mgmt request status    */

/************************************************/
/* Delete consumer of event for XEMS            */
/************************************************/
extern
void ems_mgmt_delete_consumer_of_event(
    ems_handle_t           handle,           /* XEMS handle            */
    ems_string_t           consumer,         /* Consumer’s name        */
    ems_uuid_t *           uuid,             /* Consumer UUID          */
    ems_event_type_t       type,             /* associated event type */
    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 */

/*************************************************/
/* Add supplier                                 */
/*************************************************/
extern
void ems_mgmt_add_supplier(
    ems_handle_t               handle,       /* XEMS handle         */
    ems_string_t               supplier,     /* supplier name       */
    ems_uuid_t *               uuid,         /* supplier UUID       */
    ems_error_t *              status);      /* mgmt request status */

/*************************************************/
/* Add supplier of event                        */
/*************************************************/
extern
void ems_mgmt_add_supplier_of_event(
    ems_handle_t               handle,     /* XEMS handle           */
    ems_string_t               supplier,   /* supplier name         */
    ems_uuid_t *               uuid,       /* supplier UUID         */
    ems_event_type_t           type,       /* associated event type */
    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  */

/************************************************/
/* Delete supplier of event                     */
/************************************************/
extern
void ems_mgmt_delete_supplier_of_event(
    ems_handle_t            handle,        /* XEMS handle            */
    ems_string_t            supplier,      /* supplier name          */
    ems_uuid_t *            uuid,          /* supplier UUID          */
    ems_event_type_t        type,          /* associated event type */
    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_handle_t            handle,        /* XEMS handle            */
    ems_eventid_t *         event_id,      /* event id to delete     */
    ems_error_t *           status);       /* event delete status    */
```

```
/***********************************************/
/* Management Forward Events                   */
/***********************************************/
extern
void ems_mgmt_forward(
    ems_handle_t            handle,       /* XEMS handle            */
    ems_filtername_list_t * filter_group, /*list of event filter names */
    ems_netname_t *         hostname,     /* receiver of events     */
    ems_string_t *          name,         /* assoc consumer name    */
    ems_uuid_t *            uuid,         /* consumer uuid          */
    ems_error_t *           status);      /* event delete status    */

/***********************************************/
/* Status Codes                                   */
/***********************************************/

#define ems_status_modid    (0x10*16777216)
#define ems_status_compid   (0x10*65536)
#define ems_status_subid    (0x10*256)
#define ems_status_base (sms_status_modid+ems_status_compid
                                                    +ems_status_subid)

#define ems_s_already_registered                    ems_status_base+1
#define ems_s_consumer_already_started              ems_status_base+2
#define ems_s_consumer_not_started                  ems_status_base+3
#define ems_s_empty_filter_db                       ems_status_base+4
#define ems_s_event_type_exists                     ems_status_base+5
#define ems_s_event_type_not_found                  ems_status_base+6
#define ems_s_filter_exits                          ems_status_base+7
#define ems_s_filter_in_use                         ems_status_base+8
#define ems_s_filter_not_found                      ems_status_base+9
#define ems_s_forwarding_event_service_not_there    ems_status_base+10
#define ems_s_forwarding_event_loop                 ems_status_base+11
#define ems_s_insufficient_permission               ems_status_base+12
#define ems_s_invalid_event_type                    ems_status_base+13
#define ems_s_invalid_filter                        ems_status_base+14
#define ems_s_invalid_handle                        ems_status_base+15
#define ems_s_invalid_name                          ems_status_base+16
#define ems_s_no_consumers                          ems_status_base+17
#define ems_s_no_event                              ems_status_base+18
#define ems_s_no_events                             ems_status_base+19
#define ems_s_no_memory                             ems_status_base+20
#define ems_s_no_suppliers                          ems_status_base+21
#define ems_s_no_type_list                          ems_status_base+22
#define ems_s_status_ok                             0
#define ems_s_unknown_consumer                      ems_status_base+23
#define ems_s_unknown_supplier                      ems_status_base+24
#define ems_s_unsupported_nameservice               ems_status_base+25

#endif /* _XEMS_H */
```

Part 2:

Implementations in Different Environments

The Open Group

Chapter 11

Reference Implementations

11.1 Introduction to Reference Implementations

This XEMS Part 2 contains reference implementations of the XEMS, initially for DCE and for CORBA environments.

These are intended to provide a grounding for implementors of the XEMS, in these environments. Each implementation supports the generic XEMS 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 other parts of this XEMS specification are:

- **Part 1**, which describes the XEMS generic specification
- **Part 3**, which describes event object structures for the basic event set.

Chapter 12

DCE Implementation

This chapter describes the data structures, APIs, and command line interfaces required for a DCE implementation of the XEMS.

The XEMS generic APIs are described in XEMS Part 1.

The information provided here describes additional features and facilities provided by a DCE implementation to operate with DCE consumers and producers.

12.1 DCE XEMS Data Structure IDL File

```
/*  DCE IDL File  - XEMS Data structures  */
[
uuid(000b0e1e-c016-1ce3-b57e-10005ab14004),
pointer_default(ptr),
version(2.0)
]
interface  event_management
{
 import "dce/utctypes.idl";
 import "dce/aclbase.idl";

typedef [string] char *                    ms_string_t;

/* XEMS delivery types */
typedef enum {
            ems_delivery_push=0,
            ems_delivery_pull
} ems_delivery_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;

/* XEMS priority */
typedef  unsigned long int                      ems_priority_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 networkn */
} 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_priority_t          priority;       /* event priority */
} 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;

/* XEMS filter expression grammars */
typedef unsigned16 ems_filter_grammar_t;
const ems_filter_grammar_t ems_c_fg_default     = 0;
const ems_filter_grammar_t ems_c_fg_OQL         = 1;
const ems_filter_grammar_t ems_c_fg_other       = 2;

typedef struct ems_default_fg_s_t {
        ems_string_t            attr_name;
        ems_attr_op_t           attr_operator;
        ems_attr_value_t        attr_value;
} ems_default_fg_t;

/* Event Filter Expressions */
typedef struct ems_filter_exp_s_t {
        ems_filter_grammar_t    grammar;
        union  {
                /* case: ems_c_fg_default */
                ems_default_fg_t        def_filter;
                /* case: ems_c_fg_OQL */
                ems_string_t            oql_filter;
                /* case: ems_c_fg_other */
                ems_string_t            other_filter;
        } tagged_union;
} 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_delivery_t          type;               /* consumer delivery type */
} 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;

/* Supplier */
typedef struct ems_supplier_s_t {
            ems_string_t            name;               /* DCE name of supplier */
            ems_netname_t           *hostname;          /* DCE hostname of supplier */
            uuid_t                  uuid;               /* supplier UUID */
            ems_delivery_t          type;               /* supplier delivery type */
} ems_supplier_t;

/* Supplier List */
typedef struct ems_supplier_list_s_t {
            ems_long_int            size;               /* number of suppliers */
            [size-is(size)]
            ems_supplier_t          supplier[];         /* supplier info */
} ems_supplier_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;

/* 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
}
```

12.2 DCE XEMS API: <ems.h>

```
/* <ems.h> - DCE XEMS 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)
edefine 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);

typedef void (*ems_supplier_count_handler_t)(
        ems_event_type_t type
        long int count,
        error_status_t * error);

/* 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 XEMS (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 XEMS (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 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 */
        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 */

/* Pull Supplier Register */
extern void ems_pull_supplier_register(
        ems_netname_t *         hostname,       /* event service hostname */
        ems_ushort_int          interval;       /* recommended poll interval */
        ems_handle_t *          handle,         /* XEMS handle */
        error_status_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 */
         error_status_t *       status);        /* Register status */

/* Push Supplier Register Handler */
extern void ems_supplier_register_handler(
        ems_event_type_t                type,           /* event type for handler */
        ems_supplier_count_handler_t    handler,        /* handler function */
        ems_handle_t *                  handle,         /* XEMS handle*/
        error_status_t *                status);        /* register status */

/* Supplier Unregister */
extern void ems_supplier_unregister(
        ems_handle_t *          handle,         /* XEMS handle */
        error_status_t          *status);       /* unregister status */

/* 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 */
        unsigned32                flags,          /* consumer start flags */
        ems_handler_t             hfunc[],        /* handler functions */
        uuid_t **                 uuid,           /* unique consumer id */
        ems_handle_t *            handle[],       /* array of consumer handles */
        error_status_t *          status);        /* start status */

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

/* Push Consumer Register */
extern void ems_push_consumer_register(
        ems_netname_t *           hostname,       /* emsd hostname */
        ems_filtername_list_t *   filter_group,   /* event filter group */
        int                       hfunc_index,    /* index of handler function */
        ems_handle_t *            handle,         /* ems handle */
        error_status_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 */
        error_status_t *          status);        /* register status */

/* Consumer Unregister */
extern void ems_consumer_unregister(
        ems_handle_t *            handle,         /* XEMS 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 */

/* Get Consumer Registration */
extern void ems_consumer_get_registration(
              ems_handle_t               handle,           /* XEMS handle */
              ems_netname_t **           hostname,         /* hostname of the assoc XEMS */
              ems_filtername_list_t **  filter_group,      /* event filter group */
              int *                      hfunc_index,      /*associated handler index */
              error_status_t *           status);          /* filter request status */

/* Consumer Pull */
extern void ems_consumer_pull(
              ems_handle_t               handle,           /* XEMS handle */
              ems_event_t *              event,            /* received event */
              error_status_t *           status);          /* filter request status */

/* Consumer Try Pull */
extern void ems_consumer_try_pull(
              ems_handle_t               handle,           /* XEMS handle */
              ems_event_t *              event,            /* received event */
              error_status_t *           status);          /* filter request status */

/* Management Interface */

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

/* Free XEMS 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 */

/* The following APIs are not required for the DCE implementation, and */
/* handled by the DCE Registry and ACL Management */
/*          ems_mgmt_add_consumer */
/*          ems_mgmt_add_consumer_of_event */
/*          ems_mgmt_delete_consumer_of_event */

/* Delete Registered Consumer from XEMS */
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 */

/* XEMS 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 */

/* The following APIs are not required for the DCE implementation, and  */
/* handled by the DCE Registry and ACL Management */
/*          ems_mgmt_add_supplier */
/*          ems_mgmt_add_supplier_of_event */
/*          ems_mgmt_delete_supplier_of_event */

/* List Registered Suppliers */
extern
void ems_mgmt_list_suppliers(
            ems_handle_t            handle,             /* XEMS handle */
            ems_supplier_list_t **  list,               /* returned supplier list */
            error_status_t *        status);            /* mgmt request status */

/* Free Consumer List */
extern
void ems_mgmt_free_suppliers(
            ems_supplier_list_t **  list,               /* supplier list to free  */
            error_status_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 */
        error_status_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 */
        error_status_t *        status);        /* event get status */

/* Free Undelivered Events */
extern void ems_mgmt_free_undelivered_events(
        ems_event_list_t **     event,          /* undelivered events */
        error_status_t *        status);        /* event get status */

/* Delete Undelivered Events */
extern void ems_mgmt_delete_undelivered_event(
        ems_handle_t            handle,         /* XEMS handle */
        ems_eventid_t *         event_id,       /* event id to delete */
        error_status_t *        status);        /* event delete status */

/* Management Forward Events */
extern void ems_mgmt_forward(
        ems_handle_t            handle,         /* XEMS handle */
        ems_filtername_list_t * filter_group,   /* list of event filter names */
        ems_netname_t *         hostname,       /* receiver of events */
        ems_string_t *          name,           /* assoc consumer name */
        uuid_t *                uuid,           /* consumer uuid */
        error_status_t *        status);        /* event delete status */

#endif /* _EMS_H */
```

12.3 DCE dcecp commands for XEMS

12.3.1 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]
```

12.3.2 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
```

12.3.3 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]
```

Chapter 13

CORBA Implementation

The CORBA implementation for the Event Management Service is not yet sufficiently proven to be presented in this specification as normative information.

However, the intent remains to present the XEMS CORBA implementation information as a key part of the normative content of this XEMS specification.

Accordingly, to show this intent and to make visible the existing XEMS CORBA implementation information:

- this Chapter is presented as a placeholder for the CORBA implementation
- the existing CORBA implementation is included in this document, as ‘‘appended’’ (that is, *non-normative*) information, in Appendix B.

Appendix B

CORBA Implementation

B.1 Interface Descriptions

This appendix describes the interfaces along with rationale for the implementation, then gives the IDL for the CORBA implementation.

The CORBA implementation adheres stylistically to the specification given in Part 1. The type definitions for EMS data primitives are incorporated in the IDL. The IDL is specified using the C++ mapping. EMS is couched in a module to distinguish it from other constructs. A notification manager and iterator interfaces have been added for the CORBA implementation.

This implementation uses the OMG Common Objects Services (COS) Event Service interface (see reference **COS V1**) to provide de-coupled communications.

B.2 Primitive Data Types

These data types use a prefix to avoid keyword and namespace collisions. The remaining EMS IDL is given in terms of these primitives.

```
typedef boolean          ems_boolean;         // 1 byte
typedef octet            ems_byte;            // 1 byte
typedef  char            ems_char;            // 1 byte
typedef char             ems_small_int;       // 1 byte
typedef char             ems_usmall_int;      // 1 byte
typedef short            ems_short_int;       // 2 bytes
typedef unsigned short   ems_ushort_int;      // 2 bytes
typedef long             ems_long_int;        // 4 bytes
typedef unsigned long    ems_ulong_int;       // 4 bytes

typedef struct ems_hyper_int_rep_s_t {
        ems_long_int     high;
        ems_ulong_int    low;
} ems_hyper_int;

typedef struct ems_uhyper_int_rep_s_t {
        ems_ulong_int    high;
        ems_ulong_int    low;
} ems_uhyper_int;

typedef float            ems_short_float;     // 4 bytes
typedef double           ems_long_float;      // 8 bytes
```

B.3 Composite Data Types

This implementation is based on the COS facilities described in the referenced **COS** publication. These COS facilities are commercially available from a number of vendors. A given CORBA implementation may use the additional facilities, assuming they will interoperate with the less robust implementations. For example, there is a COS specification for Universal Coordinated Time (utc). This implementation does not use the utc specification, because it had not been widely available in commercial COS implementations at the time of this writing.

A universal unique identifier (uuid) is a tag that can be associated with an entity. The tag is unique. Specifically, the time and sequencing elements provide a unique specification on a given node. The node specification provides uniqueness in a network.

```
typedef struct uuid {
        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];
} uuid_t;
```

The CORBA string is used for EMS strings.

```
typedef string                  string_t;

typedef struct string_list_s_t {
        sequence<string_t> strings;
} string_list_t;
```

The timestamp is from the X/Open DCE Time Service as articulated in COS V1[5].

```
typedef Time::UtcT              utc_t;
```

The EMS specification de-couples the security mechanism for the CORBA transport from the EMS security characteristics. This maximizes implementation flexibility, providing avenues for security adapters within an implementation. Such an adapter may bridge the CORBA security model with that of a given operating system.

```
typedef enum secobjtype_e_t {
        secobj_server,
        secobj_eventtypes,
        secobj_filters,
        secobj_consumers,
        secobj_suppliers,
        secobj_eventtype,
        secobj_filter
} secobjtype_t;

typedef struct secobj {
        secobjtype_t            secobjtype;
        string_t                name;
        uuid_t                  uuid;
} secobj_t;
```

5. The MAScOTTE project (see Glossary entry for **MAScOTTE** on page 233) has proposed that this should be the definition for CORBA Universal Time. This definition is found in **#include <CosTime.idl>**.

```
typedef struct secperm {
        ems_usmall_int          control;
        ems_usmall_int          delete;
        ems_usmall_int          insert;
        ems_usmall_int          read;
        ems_usmall_int          write;
        ems_usmall_int          execute;
} secperm_t;

typedef struct secsubj {
        string_t                name;
        uuid_t                  uuid;
} secsubj_t;

typedef struct secprin {
        Principal               principal;
} secprin_t;
```

Events are composed of attributes (data elements). This implementation uses an untyped event model. The any contains the type code for the attribute, eliminating the need for a separate **struct attribute_s_t** and **attr_value_t**:

```
typedef any                     attribute_t;
```

An event identifier is a composite consisting of the event type and an instance identifier, that is, a *uuid*:

```
typedef struct eventid_s_t {
        event_type_t            type;
        uuid_t                  id;
} eventid_t;
```

Suppliers may be adapters, extracting events from one domain and couching them in terms of EMS events. As such, a variety of network naming schemes may be used to describe the event origin:

```
typedef enum nameservice_e_t {
        ns_other,
        ns_dns,
        ns_dce,
        ns_x500,
        ns_nis,
        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;
```

The specification of the consumer is unique to the CORBA implementation. This is due to the use of the COS Event Service. This implementation provides support for both push and pull consumers:

```
enum ConsumerType {
        PULLCONSUMER,
        PUSHCONSUMER
} ;

typedef union EventConsumer switch(ConsumerType) {
        case PULLCONSUMER:
                CosEventComm::PullConsumer pullc;
        case PUSHCONSUMER:
                CosEventComm::PushConsumer pushc;
} consumer_t;

typedef struct consumer_list_s_t {
        sequence<consumer_t> consumer;
} consumer_list_t;
```

The specification of the supplier is unique to the CORBA implementation. This is due to the use of the COS Event Service. This implementation provides support for both push and pull suppliers:

```
enum SupplierType {
        PULLSUPPLIER,
        PUSHSUPPLIER
} ;

typedef union EventSupplier switch(SupplierType) {
        case PULLSUPPLIER:
                CosEventComm::PullSupplier pulls;
        case PUSHSUPPLIER:
                CosEventComm::PushSupplier pushs;
} supplier_t;

typedef struct supplier_list_s_t {
        sequence<supplier_t> supplier;
} supplier_list_t;
```

The event origin retains the structure given in Part 1[6]. The notions of and data type primitives for process, user, and group identifiers vary by operating system. For this implementation, these are assumed to be supplementary fields:

```
typedef struct ems_origin_s_t {
        supplier_t          supplier;
        string_t            descname;
        ems_ulong_int       pid;            // supplementary field
        ems_ulong_int       uid;            // supplementary field
        ems_ulong_int       gid;            // supplementary field
} origin_t;
```

There are competing notions of event severity and priority classification. Some notions are problem domain specific. This implementation uses the specification in Part 1 for lack of a clear alternative:

```
typedef enum severity_e_t {
        sev_info,
        sev_fatal,
        sev_error,
```

6. The MAScoTTE project (see Glossary entry for **MAScOTTE** on page 233) has proposed that the process identifier, the user identifier and the group identifier should be replaced by the **SysAdminLifeCycle::Location** for the CORBA location description, as defined in the referenced **XCMF-V1** specification. This would provide a fixed origin format, enabling interoperability.

```
            sev_warning,
            sev_notice,
            sev_notice_verbose,
            sev_debug
    } severity_t;

    typedef ems_ulong_int    priority_t;
```

These type definitions describe the layout of events. They are in accordance with Part 1 of this specification:

```
    typedef struct hdr_s_t {
            eventid_t                eventid;
            origin_t                 origin;
            severity_t               severity;
            utc_t                    received;
            utc_t                    delivered;
            priority_t               priority;
    } hdr_t;

    typedef struct event_s_t {
            hdr_t                    header;
            sequence<attribute_t> item;
    } event_t;

    typedef struct event_list_s_t {
            sequence<event_t>        event;
    } event_list_t;

    typedef struct event_schema_s_t {
            event_type_t             type;
            string_t                 name;
            sequence<attribute_t> attr;
    } event_schema_t;

    typedef struct event_type_list_s_t {
            sequence<event_schema_t> schema;
    } event_type_list_t;

    typedef enum attr_op_e_t {
            c_attr_op_eq,
            c_attr_op_gt,
            c_attr_op_lt,
            c_attr_op_ge,
            c_attr_op_le,
            c_attr_op_ne,
            c_attr_op_bitand,
            c_attr_op_substr
    } attr_op_t;

    typedef struct attrlist_s_t {
            sequence<attribute_t> attr;
    } attrlist_t;

    typedef enum filter_grammar_e_t {
            c_fg_default,
            c_fg_OQL,
            c_fg_other
    } filter_grammar_t;
```

```
typedef struct default_fg_s_t {
            string_t                    attr_name;
            attr_op_t                   attr_operator;
            attribute_t                 attr_value;
} default_fg_t;

typedef struct filter_exp_s_t {
            union tagged switch(filter_grammar_t) {
                        case c_fg_default:
                                    default_fg_t def_filter;
                        case c_fg_OQL:
                                    string_t oql_filter;
                        case c_fg_other:
                                    string_t other;
            } filter;
} filter_exp_t;

typedef struct filter_exp_list_s_t {
            sequence<filter_exp_t> filter_exp;
} filter_exp_list_t;

typedef struct filter_s_t {
            string_t                    filter_name;
            event_type_t                type;
            filter_exp_list_t           filter_exp_list;
} filter_t;

typedef struct filter_list_s_t {
            sequence<filter_t>          filter;
} filter_list_t;

typedef struct filtername_list_s_t {
            sequence<string_t>          filter_names;
} filtername_list_t;
```

B.4 Exceptions

The CORBA implementation uses CORBA user exceptions rather than returning status through and output argument. This retains the spirit of Part 1 of this specification, while conforming to CORBA C++ common practice of using exceptions for status delivery. These are mapped to the status codes specified in Part 1.

ExAlreadyRegistered	a consumer with this name is already registered.
ExConsumerAlreadyStarted	the consumer is already started.
ExConsumerNotStarted	the consumer has not started.
ExEmptyFilterDB	the listed filters could not be returned because the filter database is empty.
ExEventTypeExists	the event type to be added already exists.
ExEventTypeNotFound	the specified event type was not found.
ExFilterExists	the given filter name already exists.
ExFilterInUse	the filter cannot be deleted because it is currently in use.
ExFilterNotFound	the requested filter does not exist.
ExForwardingEventServiceNotThere	the event service to forward to is not available.
ExForwardingEventLoop	the host name introduces a loop condition, where EMS would be forwarding events to itself.
ExInsufficientPermission	the caller does not have sufficient permission to perform the operation.
ExInvalidEventType	the schema for the event type is not valid.
ExInvalidFilter	the input parameters specify an invalid filter.
ExInvalidHandle	the handle parameter is not valid.
ExInvalidName	the name parameter contains invalid characters.
ExNoConsumers	no consumers are registered.
ExNoEvent	tried to pull an event of a specified type, but there are no events to pull.
ExNoEvents	there are no undelivered events.
ExNoMemory	an EMS object could not be allocated.
ExNoSuppliers	no suppliers are registered.
ExNoTypeList	there was no type list in the function invocation.
ExUnknownConsumer	tried to unregister a consumer that was not registered.
ExUnsupportedNameService	unsupported name service on host name.
ExNotRegistered	the user, for example, the consumer, of the service is not registered.
ExNoFilters	no filters exist.

B.5 Registration Interface

The registration interface is not required for the CORBA implementation. The CORBA IDL-generated stubs have a *bind* method used for connecting to a specific host. In addition, each bind is interface-specific, using the CORBA COS Event Service for the connection registration.

B.6 Event Type Interface

The event type interface provides support for manipulating the event type repository.

B.6.1 Add

The *Add* operation adds a new type to the repository. This makes the new event type known to the EMS.

The input parameter contains the schema describing a new event type. This structure includes a string uniquely identifying the event type within the repository and a sequence containing the attributes supported by the new event type. Each attribute in the sequence consists of a name for the attribute, accompanied by an *any* value, indicating the attribute's type. Note that each event type automatically supports the attributes defined for an event header, so the list of attibutes supplied for the new event type should include any additional attributes.

Syntax

```
void Add(in event_schema_t schema)
        raises(ExEventTypeExists, ExInsufficientPermission);
```

Exceptions

If the event type (based on the unique identifier) already exists in the repository, the **ExEventTypeExists** exception is raised.

If the caller is not permitted to add event types, the **ExInsufficientPermission** exception is raised.

B.6.2 Delete

The *Delete* operation removes an event type from the repository.

The input parameters should be the common name or the unique identifier for an existing event type. The first parameter should be the null string or the human readable name for the event type. The second parameter should be the null unique identifier or the unique identifier for the event type. At least one of the parameters must contain a non-null value. The common name should only be used when the common names are unique within the repository. The action taken for non-unique common names, when a non-unique common name is specified, is not specified. When both the common name and the unique identifier are specified, they must refer to the same event type.

Syntax

```
void Delete(in string_t type_name,
                in event_type_t type)
        raises(ExEventTypeNotFound, ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the given event type does not exist, then the **ExEventTypeNotFound** exception is raised.

If both input parameters are specified and they do not refer to an event type, then the **ExInvalidName** exception is raised.

If the caller is not permitted to remove event types, the **ExInsufficientPermission** exception is raised.

B.6.3 Get

The *Get* operation retrieves the schema for the given event type from the repository.

The input parameters should be the common name or the unique identifier for an existing event type. The first parameter should be the null string or the human readable name for the event type. The second parameter should be the null unique identifier or the unique identifier for the event type. At least one of the parameters must contain a non-null value. The common name should only be used when the common names are unique within the repository. The action taken for non-unique common names, when a non-unique common name is specified, is not specified. When both the common name and the unique identifier are specified, they must refer to the same event type.

Syntax

```
void Get(in string_t type_name,
                 in event_type_t type,
                 out event_schema_t schema)
         raises(ExEventTypeNotFound, ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the given event type does not exist, then the **ExEventTypeNotFound** exception is raised.

If both input parameters are specified and they do not refer to an event type, then the **ExInvalidName** exception is raised.

If the caller is not permitted to remove event types, the **ExInsufficientPermission** exception is raised.

B.6.4 GetList

The *GetList* operation returns the list of all event type schemas currently maintained within the repository. Each element within the returned list is a structure of the same type described above for the *Add* operation.

Syntax

```
void GetList(out event_type_list_t type_list)
         raises(ExNoTypeList, ExInsufficientPermission);
```

Exceptions

If no event type schema exist in the repository, the **ExNoTypeList** exception is raised.

If the caller is not permitted to retrieve schema from the repository, the **ExInsufficientPermission** exception is raised.

B.7 Event Filter Interface

As described in the general model, the filter repository contains registered filters. Consumers can specify a list of names of registered filters, when registering with the EMS. The filter list is used to determine which events should be forwarded to a given consumer. The event filter interface defines methods supporting the management of filters by an EMS. The *FreeFilter*, *FreeFilterList* and *FreeNameList* operations described in Part 1 are not required for the CORBA implementation.

B.7.1 Add

The *Add* operation adds a new filter to the repository.

The first parameter should be the name for the event filter. This name must be unique among all filters in the repository. The second parameter should be the identifier for an existing event type. It associates the filter with the event type. The content of this parameter can either be the unique identifier of an event type or the common name of the event type. The third parameter contains the list of filter expressions comprising the new filter.

Syntax

```
void Add(in string_t filter_name,
                in event_type_t type,
                in filter_exp_list_t exp_list)
        raises(ExInsufficientPermission, ExFilterExists,
                ExInvalidFilter, ExEventTypeNotFound, ExInvalidName);
```

Exceptions

If the caller is not permitted to add the filter, the **ExInsufficientPermission** exception is raised.

If the filter already exists, the **ExFilterExists** exception is raised.

If the filter list is not properly composed, the **ExInvalidFilter** exception is raised.

If the given event type does not exist, the **ExEventTypeNotFound** exception is raised.

If the filter name contains invalid characters, the **ExInvalidName** exception is raised.

B.7.2 Append

The *Append* operation adds a list of filters to an existing filter expression in the filter repository.

The first 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. 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 Append(in string_t filter_name,
                in filter_exp_list_t exp_list)
        raises(ExInsufficientPermission, ExInvalidFilter, ExFilterNotFound,
                ExInvalidName);
```

Exceptions

If the caller is not permitted to add the filter, the **ExInsufficientPermission** exception is raised.

If the filter list is not properly composed, the **ExInvalidFilter** exception is raised.

If the filter does not exist in the repository, the **ExFilterNotFound** exception is raised.

If the filter name contains invalid characters, the **ExInvalidName** exception is raised.

B.7.3 Delete

The *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 are no users of the filter.

Syntax

```
void Delete(in string_t filter_name)
        raises(ExInsufficientPermission, ExFilterNotFound, ExFilterInUse, ExInvalidName);
```

Exceptions

If the caller is not permitted to delete the filter, the **ExInsufficientPermission** exception is raised.

If the filter does not exist in the repository, the **ExFilterNotFound** exception is raised.

If the filter name contains invalid characters, the **ExInvalidName** exception is raised.

B.7.4 Get

The *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. The third parameter is an output parameter that, upon successful, completion of the operation, will contain the list of filter expressions associated with the given filter.

Syntax

```
void Get(in string_t filter_name,
                in event_type_t type,
                out filter_exp_list_t filter_exprs)
        raises(ExInsufficientPermission, ExFilterNotFound, ExInvalidName);
```

Exceptions

If the caller is not permitted to get (read) the filter, the **ExInsufficientPermission** exception is raised.

If the filter does not exist in the repository, the **ExFilterNotFound** exception is raised.

If the filter name contains invalid characters, the **ExInvalidName** exception is raised.

B.7.5 GetList

The GetList *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 related event type, and the list of filter expressions associated with the filter.

Syntax

```
void GetList(out filter_list_t filter_list)
        raises(ExInsufficientPermission, ExEmptyFilterDB);
```

Exceptions

If the caller is not permitted to read the list of filter names, the **ExInsufficientPermission** exception is raised.

If there are no filters in the filter repository, the **ExEmptyFilterDB** exception is raised.

B.7.6 GetNameList

The *GetNameList* operation returns the names of the filters in the filter repository.

Syntax

```
void GetNameList(out filtername_list_t name_list)
        raises(ExInsufficientPermission, ExEmptyFilterDB);
```

Exceptions

If the caller is not permitted to read the list of filter names, the filter, the **ExInsufficientPermission** exception is raised.

If there are no filters in the filter repository, the **ExEmptyFilterDB** exception is raised.

B.8 Consumer Interface

The consumer interface allows event consumers to register and unregister with EMS. Once registered, consumers can add and delete filters, define the interesting events, etc. When EMS receives events from event suppliers, the event is filtered, using the event filter, and only matching events are forwarded to interested consumers.

B.8.1 PushConsumerRegister

The *PushConsumerRegister* operation registers a new push consumer with the EMS.

The first parameter specifies the object reference of an object which supports the **CosEventComm::PushConsumer** interface. As described in COS V1 (see reference **COS 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 parameter specifies a list of the names of filters that describe which events the new consumer is interesting in receiving. Each filter name in the list should correspond to a filter that exists in the filter repository associated with the EMS. 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 EMS repository.

Note that from the client's perspective, this operation is invoked on the *Consumer* interface supported by a particular EMS. Effectively, this operation forms a connection between the target EMS and a consumer. It should be recalled that internally, each EMS manages one or more event channels of the type described COS V1 (see reference **COS V1**). It is anticipated that during the course of performing this operation the EMS 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 EMS 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 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 PushConsumerRegister(in CosEventComm::PushConsumer consumer,
                          in filtername_list_t filter_group)
        raises(ExAlreadyRegistered, ExFilterNotFound, ExInsufficientPermissions,
               ExNoMemory);
```

Exceptions

If the input consumer is already registered with the target EMS, 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 EMS, the **ExFilterNotFound** exception is raised. The filter name field of the exception is set to the name of the missing filter from the input list.

If the consumer being registered is not authorized to register with the EMS, the **ExInsufficientPermission** exception is raised. This exception is also raised when the consumer is not permitted to receive one or more event types contained in the input list.

If there is insufficient memory to allocate to register the connection, the **ExNoMemory** exception is raised.

B.8.2 PullConsumerRegister

The *PullConsumerRegister* operation registers a new pull consumer with the EMS.

The first input parameter specifies the object reference of an object which supports the **CosEventComm::PullConsumer** interface. As described in COS V1 (see reference **COS V1**), an object supporting this interface will form a connection with 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 through the event channel.

The second input parameter 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 in the repository. 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 EMS filter repository.

Note that from the client's perspective, this operation is invoked on the *Consumer* interface of the specific EMS. Effectively, this operation forms a connection between the EMS and the consumer. It should be recalled that internally, each EMS manages one or more event channels of the type described in COS V1 (see reference **COS V1**). It is anticipated that during the course of performing this operation the EMS will invoke the appropriate event channel and the input consumer. Any event received by the EMS for which one of the filters in the consumer's filter group evaluates to TRUE will be made available for the pulling consumer. This can be done by having the client invoke the *Receive* operation in the *Consumer* interface. This operation is likely to be implemented in terms of the *pull* or *try_pull* operations supported by an event channel managed by the EMS. The precise relationship between the EMS and the event channels, and how the 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 PullConsumerRegister(in CosEventComm::PullConsumer consumer,
                          in filtername_list_t filter_group)
        raises(ExAlreadyRegistered, ExFilterNotFound, ExInsufficientPermission,
               ExNoMemory);
```

Exceptions

If the input consumer is already registered with the target EMS, 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 EMS, the **ExFilterNotFound** exception is raised. The filter name field of the exception is set to the name of the missing filter from the input list.

If the consumer being registered is not authorized to register with the EMS, the **ExInsufficientPermission** exception is raised. This exception is also raised when the consumer is not permitted to receive one or more event types contained in the input list.

If there is insufficient memory to allocate to register the connection, the **ExNoMemory** exception is raised.

B.8.3 Unregister

The *Unregister* operation removes knowledge of the consumer from the repository of the targeted EMS.

The input parameter specifies the object reference of the consumer to be unregistered.

Syntax

```
void Unregister(in consumer_t consumer)
        raises(ExNotRegistered);
```

Exceptions

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

B.8.4 AddFilterToGroup

The *AddFilterToGroup* 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 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 as a consumer of the target EMS.

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 of an event filter that currently exists in the filter repository maintained by the target EMS.

Syntax

```
void AddFilterToGroup(in consumer_t consumer,
                in filtername_list_t event_filters)
        raises(ExInsufficientPermission, ExNotRegistered, ExFilterNotFound);
```

Exceptions

If the first 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, that currently exists in the filter repository, the **ExFilterNotFound** exceptions 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 on one or more types indicated by the filters being registered for, the **ExInsufficientPermission** exception is raised.

B.8.5 DeleteFilterFromGroup

The *DeleteFilterFromGroup* operation removes a filter from the group of filters of the currently registered consumer.

The first input parameter is the object reference of the consumer whose event filter grop is being updated. This consumer should be currently registered in the target EMS repository.

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 that currently exists in the target EMS filter repository, and is currently one of the event filters in the input consumers filter group.

Syntax

```
void DeleteFilterFromGroup(in consumer_t consumer,
                in filtername_list_t filter_name)
        raises(ExInsufficientPermission, ExNotRegistered, ExFilterNotFound);
```

Exceptions

If the input consumer is not authorized to receive events on one or more types indicated by the filters being registered for, the **ExInsufficientPermission** exception is raised.

If the first 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, that currently exists in the filter repository, the **ExFilterNotFound** exceptions is raised with the filter name field of the exception structure set to the first such name encountered in the sequence.

B.8.6 GetFilterGroup

The *GetFilterGroup* operationreturns the list of filters for which the consumer is currently registered.

The first parameter is the object reference of an event consumer that should be currently registered with the target EMS.

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 GetFilterGroup(in consumer_t consumer,
                out filtername_list_t filter_group)
        raises(ExNotRegistered, ExNoFilters);
```

Exceptions

If the first parameter is not the object reference of a currently registered consumer, the **ExNotRegistered** exception is raised.

If the consumer is not curently registered for any event filters, the **ExNoFilters** exception is raised.

B.8.7 GetRegistration

The *GetRegistration* operation returns object references for current consumers.

Syntax

```
void GetRegistration(out consumer_list_t push_consumers,
                     out consumer_list_t pull_consumers)
          raises(ExNoConsumers);
```

Exceptions

If there are no registered consumers, the **ExNoConsumers** exception is raised.

B.8.8 Receive

The *Receive* operation returns an event for a currently registered *pull* consumer.

The first input parameter is an area to receive the event for a *pull* consumer.

Syntax

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

Exceptions

If the current consumer is not currently registered as a *pull* consumer, the **ExNotRegistered** exception is raised.

B.9 ConsumerAdmin Interface

The **ConsumerAdmin** interface is the part of the **Administration** interface pertaining to the management of consumers. This interface inherits the **Consumer** interface, permitting the administrator to perform consumer operations with a **ConsumerAdmin** object reference.

B.9.1 ListConsumers

The *ListConsumers* operation lists the consumers registered with the target EMS.

Upon successful return, the output parameter contains a sequence of consumers in the consumer repository for the target EMS.

Syntax

```
void ListConsumers(out consumer_list_t consumers)
        raises(ExNoConsumers);
```

Exceptions

If there are no consumers in the consumer repository, the **ExNoConsumers** exception is raised.

B.9.2 DeleteConsumer

The *DeleteConsumer* operation removes a consumer from the consumer repository of the target EMS.

At least one of the input parameters must be specified. If both parameters are specified, they must reference the same consumer. The first input parameter specifies the name of the consumer. This parameter is not specified by referencing the null string as the input parameter. The second parameter specifies the identifier of the consumer. This parameter is not specified by referencing the null identifier as the input parameter.

Syntax

```
void DeleteConsumer(in string_t consumer,
                in uuid_t uuid)
        raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input specification does not reference a consumer in the consumer repository, or the input parameters are both specified and reference different consumers, the **ExInvalidName** exception is raised.

If the user is not authorized to remove consumers from the consumer repository, the **ExInsufficientPermission** exception is raised.

B.9.3 AdminDeleteFilterFromGroup

The *AdminDeleteFilterFromGroup* operation removes a sequence of filters from a registered consumer's filter group. The registered consumer may be retrieved through the **Consumer::GetRegistration** operation. The sequence of filter names for the registered consumer may be obtained through the **Consumer::GetFilterGroup** operation.

At least one of the input parameters for the consumer name must be specified. If both parameters are specified, they must reference the same consumer. The first input parameter specifies the name of the consumer. This parameter is not specified by referencing the null string as the input parameter. The second parameter specifies the identifier of the consumer. This parameter is not specified by referencing the null identifier as the input parameter. The third input parameter is the sequence of filter names to be removed from the consumer's filter group. The filter names in the sequence must be members of the consumer's filter group.

Syntax

```
void AdminDeleteFilterFromGroup(in string_t consumer,
                                in uuid_t uuid,
                                in filtername_list_t filter_name)
        raises(ExInvalidName, ExInvalidFilter, ExInsufficientPermission);
```

Exceptions

If the input specification does not reference a registered consumer for the target EMS, or the input parameters are both specified and reference different consumers, the **ExInvalidName** exception is raised.

If a filter name is not a member of the registered consumers filter group, the **ExInvalidFilter** exception is raised.

If the user is not authorized to remove filters from the registered consumers filter group, the **ExInsufficientPermission** exception is raised.

B.9.4 AdminAddFilterToGroup

The *AdminAddFilterToGroup* operation inserts the set of filters into the registered consumer's filter group. The registered consumer may be retrieved through the **Consumer::GetRegistration** operation. The sequence of filter names for the registered consumer may be obtained through the **Consumer::GetFilterGroup** operation.

At least one of the input parameters for the consumer name must be specified. If both parameters are specified, they must reference the same consumer. The first input parameter specifies the name of the consumer. This parameter is not specified by referencing the null string as the input parameter. The second parameter specifies the identifier of the consumer. This parameter is not specified by referencing the null identifier as the input parameter. The third input parameter is the sequence of filter names to be added to the consumer's filter group. The filter names in the sequence must not be members of the consumer's filter group.

Syntax

```
void AdminAddFilterToGroup(in string_t consumer,
                           in uuid_t uuid,
                           in filtername_list_t filter_name)
          raises(ExInvalidName, ExInvalidFilter, ExInsufficientPermission);
```

Exceptions

If the input specification does not reference a registered consumer for the target EMS, or the input parameters are both specified and reference different consumers, the **ExInvalidName** exception is raised.

If a filter name is a member of the registered consumers filter group, the **ExInvalidFilter** exception is raised.

If the user is not authorized to add filters to the registered consumers filter group, the **ExInsufficientPermission** exception is raised.

B.9.5 AdminGetFilterGroup

The *AdminGetFilterGroup* operation retrieves a sequence of filter names representing the filter group for the given consumer. The registered consumer may be retrieved through the **Consumer::GetRegistration** operation.

At least one of the input parameters for the consumer name must be specified. If both parameters are specified, they must reference the same consumer. The first input parameter specifies the name of the consumer. This parameter is not specified by referencing the null string as the input parameter. The second parameter specifies the identifier of the consumer. This parameter is not specified by referencing the null identifier as the input parameter. The third parameter is the sequence of filter names representing the consumer's filter group.

Syntax

```
void AdminGetFilterGroup(in string_t consumer,
                         in uuid_t uuid,
                         out filtername_list_t filter_name)
          raises(ExInvalidName, ExInsufficientPermission, ExNoFilters);
```

Exceptions

If the input specification does not reference a registered consumer for the target EMS, or the input parameters are both specified and reference different consumers, the **ExInvalidName** exception is raised.

If there are no members in the registered consumers filter group, the **ExNoFilters** exception is raised.

If the user is not authorized to view the filter group of the registered consumer, the **ExInsufficientPermission** exception is raised.

B.10 Supplier Interface

The Supplier interface provides a means for managed objects to convey events to the EMS.

B.10.1 PushSupplierRegister

The *PushSupplierRegister* operation registers a new push supplier with the target EMS.

The input parameter specifies the object reference of an object which supports the **CosEventComm::PushSupplier** interface. As described in COS V1 (see reference **COS V1**), an object supporting this interface will form a connection with 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 perspective, this operation is invoked on the *Supplier* interface for the target EMS. Effectively, this operation forms a connection between the target EMS and a supplier. It should be recalled that internally, each EMS manages one or more event channels of the type described in COS V1 (see reference **COS V1**). It is anticipated that during the course of performing this operation the target EMS 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 target EMS. This is performed transparently whenever the supplier invokes the i.I send operation. The precise relationship between the target EMS and event channels is left as an implementation detail since these aspects will be largely dependent on the specific implementation of the underlying event channels.

Syntax

```
void PushSupplierRegister(in CosEventComm::PushSupplier supplier)
        raises(ExAlreadyRegistered, ExInsufficientPermission);
```

Exceptions

If the input supplier is already registered with the target EMS, the **ExAlreadyRegistered** exception is raised.

If the input supplier in not authorized to register with the target EMS, the **ExInsufficientPermission** exception is raised.

B.10.2 PullSupplierRegister

The *PullSupplierRegister* operation registers a new *pull* supplier with the target EMS.

The input parameter specifies a reference to an object, supporting the **CosEventComm::PullSupplier** interface. As described in COS V1 (see reference **COS V1**), an object supporting this interface will support *pull* and *try_pull* operations which are invoked by an event consumer in order to transmit events to the consumer of an event.

Precisely how an EMS will pull events from suppliers registered using this operation is a detail left up to the implementers of this specification. It is envisioned that not all implementations will support this feature.

Syntax

```
void PullSupplierRegister(in CosEventComm::PullSupplier supplier)
        raises(ExAlreadyRegistered, ExInsufficientPermission);
```

Exceptions

If the input supplier is already registered with the target EMS, the **ExAlreadyRegistered** exception is raised.

If the input supplier is not authorized to register with the target EMS, the **ExInsufficientPermission** exception is raised.

B.10.3 Unregister

The *Unregister* operation removes registration knowledge of the supplier from the target EMS.

The input parameter specifies the object reference of the supplier to be unregistered.

Syntax

```
void UnRegister(in supplier_t supplier)
        raises(ExNotRegistered, ExInsufficientPermission);
```

Exceptions

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

If the user is not authorized to unregister the supplier, the **ExInsufficientPermission** exception is raised.

B.10.4 Send

The *Send* operation transmits a message from a push supplier to an EMS, 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 EMS 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 issued when registering with the target EMS as a *push* supplier.

The input parameter specifies the event to be forwarded to the EMS.

Syntax

```
void Send(in event_t event)
        raises(ExNotRegistered, ExInsufficientPermission);
```

Exceptions

If the supplier invoking this operation is not currently registered as a *push* supplier with the target EMS, the **ExNotRegistered** exception is raised.

If the supplier sending the event is not authorized to send events of the type being generated, the **ExInsufficientPermission** exception is raised.

B.10.5 GetRegistration

The *GetRegistration* operation returns object references for current suppliers.

Syntax

```
void GetRegistration(out supplier_list_t push_suppliers,
                     out supplier_list_t pull_supplier)
         raises(ExNoSuppliers);
```

Exceptions

If there are no registered suppliers, the **ExNoSuppliers** exception is raised.

B.11 SupplierAdmin

The **SupplierAdmin** interface is the part of the **Administration**interface**pertaining**to**the** management of suppliers. This interface inherits the **Supplier** interface, permitting the administrator to perform supplier operations with a **SupplierAdmin** object reference.

B.11.1 ListSuppliers

The *ListSuppliers* operation lists the suppliers registered with the target EMS.

Upon successful return, the output parameter contains a sequence of suppliers in the supplier repository for the target EMS.

Syntax

```
void ListSuppliers(out supplier_list_t suppliers)
        raises(ExInsufficientPermission, ExNoSuppliers);
```

Exceptions

If the user is not authorized to retrieve the sequence of suppliers, the **ExInsuficientPermission** exception is raised.

If there are no suppliers in the suppliers repository, the **ExNoSuppliers** exception is raised.

B.11.2 DeleteSupplier

The *DeleteSupplier* operation removes a supplier from the supplier repository of the target EMS.

At least one of the input parameters must be specified. If both parameters are specified, they must reference the same supplier. The first input parameter specifies the name of the supplier. This parameter is not specified by referencing the null string as the input parameter. The second parameter specifies the identifier of the supplier. This parameter is not specified by referencing the null identifier as the input parameter.

Syntax

```
void DeleteSupplier(in string_t supplier,
                    in uuid_t uuid)
        raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input specification does not reference a supplier in the supplier repository, or the input parameters are both specified and reference different suppliers, the **ExInvalidName** exception is raised.

If the user is not authorized to remove suppliers from the supplier repository, the **ExInsufficientPermission** exception is raised.

B.12 EventIterator

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** interface 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 interface.

The result of using this interface is increased scalability, usability, and performance.

B.12.1 NextOne

The *NextOne* 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

```
ems_boolean NextOne(out event_t event);
```

Exceptions

Standard CORBA exceptions.

B.12.2 NextN

The *NextN* 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

```
ems_boolean NextN(in ems_ulong_int how_many,
                  out event_list_t events);
```

Exceptions

Standard CORBA exceptions.

B.12.3 Destroy

The *Destroy* operation destroys the iterator.

Syntax

```
void Destroy( );
```

Exceptions

Standard CORBA exceptions.

B.13 Registry

The **Registry** interface is a factory for the **Supplier** and **Consumer** interfaces.

B.13.1 ForSupplier

The **ForSupplier** operation returns a *Supplier* object reference.

Syntax

```
Supplier ForSupplier( );
```

Exceptions

Standard CORBA exceptions.

B.13.2 ForConsumer

The *ForConsumer* operation returns a *Consumer* object reference.

Syntax

```
Consumer ForConsumer( );
```

Exceptions

Standard CORBA exceptions.

B.14 RegistryAdmin

The **RegistryAdmin** interface allows the administration of the target EMS and associated repositories. This interface inherits the **Registry** interface, permitting consumer and supplier operations from an object reference for this interface.

B.14.1 ListAttributes

The *ListAttributes* operation allows the users to retrieve EMS implementation defined registry attributes.

Syntax

```
void ListAttributes(out attrlist_t attributes);
```

Exceptions

Standard CORBA exceptions.

B.14.2 GetUndeliveredEvents

The *GetUndeliveredEvents* operation permits the retrieval of undelivered events under control of an event filter from the event repository.

The first input parameter specifies the name of an event filter. The null string may be used to indicate no filtering.

The second input parameter indicates the number of events to fetch. This is used in conjunction with the EventIterator object reference returned as the last parameter.

The third parameter is the sequence of events.

The fourth parameter is the object reference for an event iterator object.

Syntax

```
void GetUndeliveredEvents(in string filter_name,
                          in unsigned long how_many,
                          out event_list_t events,
                          out EventIterator ei)
        raises(ExNoEvents, ExFilterNotFound, ExInsufficientPermission);
```

Exceptions

If there are no undelivered events meeting the filter criteria, the **ExNoEvents** exception is raised.

If the filter name does not specify the name of a filter for the target EMS, the **ExFilterNotFound** exception is raised.

If the user is not authorized to retrieve undelivered events, the **ExInsufficientPermission** exception is raised.

B.14.3 DeleteUndeliveredEventsByFilter

The *DeleteUndeliveredEventsByFilter* operation provides a mechanism for the removal of the set undelivered events meeting the evaluation criteria given by the filter.

Syntax

```
void DeleteUndeliveredEventsByFilter(in  string filter_name)
        raises(ExNoEvent, ExFilterNotFound, ExInsufficientPermission);
```

Exceptions

If there are no events meeting the filter criteria (or no undelivered events), the **ExNoEvent** exception is raised.

If the filter does not exist in the target EMS, the **ExFilterNotFound** exception is raised.

If the user is not authorized to remove undelivered events, the **ExInsufficientPermission** exception is raised.

B.14.4 DeleteUndeliveredEvent

The *DeleteUndeliveredEvent* operation removes the specified event from the target EMS.

Syntax

```
void DeleteUndeliveredEvent(in  eventid_t event)
        raises(ExNoEvent, ExInsufficientPermission);
```

Exceptions

If the specified event does not exist in the target EMS, the **ExNoEvent** exception is raised.

If the user is not authorized to retrieve undelivered events, the **ExInsufficientPermission** exception is raised.

B.14.5 Forward

The *Forward* operation indicates that all events satisfying the specified filter at the current EMS are to be forwarded to the EMS at the specified object reference. In effect, this operation creates a *supplier*in*the*current*EMS*.The current EMS in its *forwarding supplier*.

The first input parameter specifies the sequence of filters to be applied against the forwarding request. The second input parameter specifies the object reference of an object which supports the **CosEventComm::PushConsumer** interface. As described in COS V1 (see reference **COS**V1**),** an object supporting this interface will support a *push* operation which is invoked by the current EMS (using an event supplier interface) in order to send the target EMS (using a *consumer* interface) an event.

Syntax

```
void Forward(in filtername_list_t filter_group,
             in consumer_t xems_forward_consumer,
             out string_t name,
             out uuid_t uuid)
      raises(ExForwardingEventServiceNotThere,
             ExForwardingEventLoop, ExInsufficientPermission);
```

Exceptions

If the input host specification does not refer to an EMS or the EMS is not accessible from the current EMS, the **ExForwardEventServiceNotThere** exception is raised.

If the adding the specified host in conjunction with the given filter specification would cause an event forwarding loop, the **ExForwardingEventLoop** is raised.

If the user is not authorized to retrieve undelivered events, the **ExInsufficientPermission** exception is raised.

B.15 Security

The **Security** interface encapsulates the security administrative operations. This interface is security implementation neutral. It may be used with an implementation of the CORBA security service. It may used with other security implementations.

B.15.1 Edit

The *Edit* operation alters the permission attributes for a subject with regard to an EMS object.

The first input parameter specifies the EMS security object. The second input parameter specifies the security subject, for example, a *consumer* or a *supplier*. The third input parameter describes the new permissions. The output parameter contains the old permissions.

Syntax

```
void Edit(in secobj_t secobj,
              in secsubj_t subject,
              in secperm_t newperm,
              out secperm_t oldperm)
     raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input security object, security subject, or security permissions do not exist, the **ExInvalidName** exception is raised.

If the user is not authorized to edit the security permissions, the **ExInsufficientPermission** exception is raised.

B.15.2 Read

The *Read* operation retrieves the security permissions for the specified object and subject.

The first input parameter specifies the EMS security object. The second input parameter specifies the security subject, for example, a *consumer* or a *supplier*. The output parameter contains the current permissions.

Syntax

```
void Read(in secobj_t secobj,
              in secsubj_t subject,
              out secperm_t oldperm)
     raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input security object or security subject do not exist, the **ExInvalidName** exception is raised.

If the user is not authorized to retrieve security permissions, the **ExInsufficientPermission** exception is raised.

B.15.3 SubjAdd

The *SubjAdd* operation identifies a principal as an EMS subject. The first input parameter specifies the EMS subject. The second input parameter specifies the principal.

Syntax

```
void SubjAdd(in secsubj_t subject,
                in secprin_t principal)
        raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input security subject already exists, the **ExInvalidName** exception is raised.

If the user is not authorized to add security subject/principal associations, the **ExInsufficientPermission** exception is raised.

B.15.4 SubjDelete

The *SubjDelete* operation removes the association of a principal with an EMS subject.

The input parameter specifies the name of an existing subject.

Syntax

```
void SubjDelete(in secsubj_t subject)
        raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the input security subject does not exist, the **ExInvalidName** exception is raised.

If the user is not authorized to remove security subject/principal associations, the **ExInsufficientPermission** exception is raised.

B.15.5 SubjGet

The *SubjGet* operation retrieves the subject associated with the specified principal.

The input parameter specifies the principal. The principal may have been obtained through the CORBA **get_principal** operation. The output parameter contains the EMS subject associated with the principal.

Syntax

```
void SubjGet(in secprin_t principal,
        out secsubj_t subject)
raises(ExInvalidName, ExInsufficientPermission);
```

Exceptions

If the principal is not associated with a security subject, the **ExInvalidName** exception is raised.

If the user is not authorized to retrieve security subject/principal associations, the **ExInsufficientPermission** exception is raised.

B.16 Management Interface

The **Management** interface provides a means to administer various operational aspects of an EMS.

B.16.1 Systems Attribute

The *Systems* attribute provides a means of identifying a federation of EMS.

Syntax

```
typedef sequence<Management> EventManagementSystems;

readonly attribute EventManagementSystems Systems;
```

B.16.2 ObtainRegistry

The *ObtainRegistry* operation returns an object reference to the **Registry** interface.

Syntax

```
Registry ObtainRegistry( )
        raises(ExInsufficientPermission);
```

Exceptions

If the user is not authorized to retrieve the registry object reference, the **ExInsufficientPermission** exception is raised.

B.16.3 ObtainSecurity

The *ObtainSecurity* operation returns an object reference to the Security interface.

Syntax

```
Security ObtainSecurity( )
        raises(ExInsufficientPermission);
```

Exceptions

If the user is not authorized to retrieve the security object reference, the **ExInsufficientPermission** exception is raised.

B.16.4 ObtainTypeRepository

The *ObtainTypeRepository* operation returns the **EventType** interface object reference.

Syntax

```
EventType ObtainTypeRepository( )
        raises(ExInsufficientPermission);
```

Exceptions

If the user is not authorized to retrieve the **EventType** interface object reference, the **ExInsufficientPermission** exception is raised.

B.16.5 ObtainFilterRepository

The *ObtainFilterRepository* operation returns the **EventFilter** interface object reference.

Syntax

```
EventFilter ObtainFilterRepository( )
        raises(ExInsufficientPermission);
```

Exceptions

If the user is not authorized to retrieve the **EventFilter** interface object reference, the **ExInsufficientPermission** exception is raised.

B.17 IDL

```
//
// Event Management Service (EMS)
//

#include <CosEventComm.idl>              // COS Event Communications
#include <CosTime.idl>                   // COS Time Specification
// #include <SysAdminLifeCycle.idl> EMS Location Specification

//
// eMS data primitives
//
// Note: These are outside the module definition to permit names like ems_char.
//

typedef boolean                 ems_boolean;            // 1 byte
typedef octet                   ems_byte;               // 1 byte
typedef  char                   ems_char;               // 1 byte
typedef char                    ems_small_int;          // 1 byte
typedef char                    ems_usmall_int;         // 1 byte
typedef short                   ems_short_int;          // 2 bytes
typedef unsigned short          ems_ushort_int;         // 2 bytes
typedef long                    ems_long_int;           // 4 bytes
typedef unsigned long           ems_ulong_int;          // 4 bytes

typedef struct ems_hyper_int_rep_s_t {
        ems_long_int            high;
        ems_ulong_int           low;
} ems_hyper_int;

typedef struct ems_uhyper_int_rep_s_t {
        ems_ulong_int           high;
        ems_ulong_int           low;
} ems_uhyper_int;

typedef float                   ems_short_float;// 4 bytes
typedef double                  ems_long_float; // 8 bytes

module ems {

        //
        // Universal unique identifier
        //

        typedef struct uuid {
                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];
        } uuid_t;

        //
        // String representation
        //

        typedef string                  string_t;
        typedef struct string_list_s_t {
                sequence<string_t> strings;
        } string_list_t;

        //
        // Timestamp representation
        //

        typedef Time::UtcT                    utc_t;

        //
        // Error Status representation
        //

        typedef ems_ulong_int                 error_t;

        //
        // Event Type
        //

        typedef uuid_t                        event_type_t;

        //
        // Delivery Type
        //

        typedef enum delivery_s_t {
                delivery_push,
                delivery_pull
        } delivery_t;

        //
        // Security Types
        //

        typedef enum secobjtype_e_t {
                secobj_server,
                secobj_eventtypes,
                secobj_filters,
                secobj_consumers,
                secobj_suppliers,
                secobj_eventtype,
                secobj_filter
        } secobjtype_t;

        typedef struct secobj {
                secobjtype_t              secobjtype;
                string_t                  name;
                uuid_t                    uuid;
        } secobj_t;

        //
        // Permissions Attributes
        //

        typedef struct secperm {
                ems_usmall_int            control;
                ems_usmall_int            delete;
                ems_usmall_int            insert;
```

```
        ems_usmall_int          read;
        ems_usmall_int          write;
        ems_usmall_int          execute;
} secperm_t;

//
// Subject Type
//

typedef struct secsubj {
        string_t                name;
        uuid_t                  uuid;
} secsubj_t;

//
// Principal Type
//

typedef struct secprin {
        Principal               principal;
} secprin_t;

//
// Attribute Representation
//

typedef any                     attribute_t;

//
// Event Identifier
//

typedef struct eventid_s_t {
        event_type_t            type;
        uuid_t                  id;
} eventid_t;

//
// Network Name Types
//

typedef enum nameservice_e_t {
        ns_other,
        ns_dns,
        ns_dce,
        ns_x500,
        ns_nis,
        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;
```

```
//
// Consumer Type
//

enum ConsumerType {
        PULLCONSUMER,
        PUSHCONSUMER
} ;

typedef union EventConsumer switch(ConsumerType) {
        case PULLCONSUMER:
                CosEventComm::PullConsumer pullc;
        case PUSHCONSUMER:
                CosEventComm::PushConsumer pushc;
} consumer_t;

typedef struct consumer_list_s_t {
        sequence<consumer_t> consumer;
} consumer_list_t;

//
// Supplier Type
//

enum SupplierType {
        PULLSUPPLIER,
        PUSHSUPPLIER
} ;

typedef union EventSupplier switch(SupplierType) {
        case PULLSUPPLIER:
                CosEventComm::PullSupplier pulls;
        case PUSHSUPPLIER:
                CosEventComm::PushSupplier pushs;
} supplier_t;

typedef struct supplier_list_s_t {
        sequence<supplier_t> supplier;
} supplier_list_t;

//
// Event Origin Type
//

typedef struct ems_origin_s_t {
        supplier_t              supplier;
        string_t                descname;
                                        // SysAdminLifeCycle::Location id;
        ems_ulong_int           pid;    // supplementary field
        ems_ulong_int           uid;    // supplementary field
        ems_ulong_int           gid;    // supplementary field
} origin_t;

//
// Event Severity Type
//

typedef enum severity_e_t {
        sev_info,
        sev_fatal,
        sev_error,
        sev_warning,
        sev_notice,
        sev_notice_verbose,
        sev_debug
} severity_t;

//
// Event Priority Type
//

typedef ems_ulong_int    priority_t;

//
// Event Header
//

typedef struct hdr_s_t {
        eventid_t               eventid;
        origin_t                origin;
        severity_t              severity;
        utc_t                   received;
        utc_t                   delivered;
        priority_t              priority;
} hdr_t;

//
// Event Type
//

typedef struct event_s_t {
        hdr_t                   header;
        sequence<attribute_t> item;
} event_t;

typedef struct event_list_s_t {
        sequence<event_t>       event;
} event_list_t;

//
// Event Schema
//

typedef struct event_schema_s_t {
        event_type_t            type;
        string_t                name;
        sequence<attribute_t> attr;
} event_schema_t;

//
// Event Type List
//

typedef struct event_type_list_s_t {
        sequence<event_schema_t> schema;
} event_type_list_t;

//
```

```
// Attribute Operators
//

typedef enum attr_op_e_t {
            c_attr_op_eq,
            c_attr_op_gt,
            c_attr_op_lt,
            c_attr_op_ge,
            c_attr_op_le,
            c_attr_op_ne,
            c_attr_op_bitand,
            c_attr_op_substr
} attr_op_t;

typedef struct attrlist_s_t {
            sequence<attribute_t> attr;
} attrlist_t;

//
// Event Filter Grammar
//

typedef enum filter_grammar_e_t {
            c_fg_default,
            c_fg_OQL,
            c_fg_other
} filter_grammar_t;

//
// Default Event Filter Grammar
//

typedef struct default_fg_s_t {
            string_t                attr_name;
            attr_op_t               attr_operator;
            attribute_t             attr_value;
} default_fg_t;

//
// Event Filter Expression
//

typedef struct filter_exp_s_t {
            union tagged switch(filter_grammar_t) {
                        case c_fg_default:
                                    default_fg_t def_filter;
                        case c_fg_OQL:
                                    string_t oql_filter;
                        case c_fg_other:
                                    string_t other;
            } filter;
} filter_exp_t;

//
// Event Filter Expression List
//

typedef struct filter_exp_list_s_t {
            sequence<filter_exp_t> filter_exp;
} filter_exp_list_t;

//
// Event Filter Type
//

typedef struct filter_s_t {
            string_t                filter_name;
            event_type_t            type;
            filter_exp_list_t   filter_exp_list;
} filter_t;

typedef struct filter_list_s_t {
            sequence<filter_t>      filter;
} filter_list_t;

typedef struct filtername_list_s_t {
            sequence<string_t>      filter_names;
} filtername_list_t;

//
// Exceptions
//

exception ExAlreadyRegistered {};
exception ExConsumerAlreadyStarted {};
exception ExConsumerNotStarted {};
exception ExEmptyFilterDB {};
exception ExEventTypeExists {};
exception ExEventTypeNotFound {};
exception ExFilterExists {};
exception ExFilterInUse {};
exception ExFilterNotFound {};
exception ExForwardingEventServiceNotThere {};
exception ExForwardingEventLoop {};
exception ExInsufficientPermission {};
exception ExInvalidEventType {};
exception ExInvalidFilter {};
exception ExInvalidHandle {};
exception ExInvalidName {};
exception ExNoConsumers {};
exception ExNoEvent {};
exception ExNoEvents {};
exception ExNoMemory {};
exception ExNoSuppliers {};
exception ExNoTypeList {};
exception ExUnknownConsumer {};
exception ExUnsupportedNameService {};
exception ExNotRegistered {};
exception ExNoFilters {};

//
// Registration Interface
//
// Note: This interface is not required. The stubs have a bind
//          method used for connecting to a specific host. In addition,
//          each bind is interface specific.
//
```

```
//
// Event Type Interface
//
//

interface EventType {
          void Add(in event_schema_t schema)
                    raises(ExEventTypeExists, ExInsufficientPermission);

          void Delete(in string_t type_name,
                                 in event_type_t type)
                    raises(ExEventTypeNotFound, ExInvalidName,
                                ExInsufficientPermission);

          void Get(in string_t type_name,
                               in event_type_t type,
                               out event_schema_t schema)
                    raises(ExEventTypeNotFound, ExInvalidName,
                                ExInsufficientPermission);

          void GetList(out event_type_list_t type_list)
                    raises(ExNoTypeList, ExInsufficientPermission);

} ;

//
// Event Filter Interface
//

interface EventFilter {
          void Add(in string_t filter_name,
                               in event_type_t type,
                               in filter_exp_list_t exp_list)
                    raises(ExInsufficientPermission, ExFilterExists,
                                ExInvalidFilter, ExEventTypeNotFound, ExInvalidName);

          void Append(in string_t filter_name,
                                      in filter_exp_list_t exp_list)
                    raises(ExInsufficientPermission, ExInvalidFilter, ExFilterNotFound,
                              ExInvalidName);

          void Get(in string_t filter_name,
                               in event_type_t type,
                               out filter_exp_list_t filter_exprs)
                    raises(ExInsufficientPermission, ExFilterNotFound, ExInvalidName);

          void Delete(in string_t filter_name)
                    raises(ExInsufficientPermission, ExFilterNotFound, ExFilterInUse,
                              ExInvalidName);

          void GetNameList(out filtername_list_t name_list)
                    raises(ExInsufficientPermission, ExEmptyFilterDB);

          void GetList(out filter_list_t filter_list)
                    raises(ExInsufficientPermission, ExEmptyFilterDB);

} ;

//
```

```
        // Consumer Interface
        //

        interface Consumer {
                void PushConsumerRegister(in CosEventComm::PushConsumer consumer,
                                          in filtername_list_t filter_group)
                        raises(ExAlreadyRegistered, ExFilterNotFound,
                            ExInsufficientPermissions, ExNoMemory);

                void PullConsumerRegister(in CosEventComm::PullConsumer consumer,
                                          in filtername_list_t filter_group)
                        raises(ExAlreadyRegistered, ExFilterNotFound,
                            ExInsufficientPermissions, ExNoMemory);

                void Unregister(in consumer_t consumer)
                        raises(ExNotRegistered);

                void AddFilterToGroup(in consumer_t consumer,
                                          in filtername_list_t event_filters)
                        raises(ExInsufficientPermission, ExNotRegistered, ExFilterNotFound);

                void DeleteFilterFromGroup(in consumer_t consumer,
                                          in filtername_list_t filter_name)
                        raises(ExInsufficientPermission, ExNotRegistered, ExFilterNotFound);

                void GetFilterGroup(in consumer_t consumer,
                                          out filtername_list_t filter_group)
                        raises(ExNotRegistered, ExNoFilters);

                void GetRegistration(out consumer_list_t push_consumers,
                                          out consumer_list_t pull_consumers)
                        raises(ExNoConsumers);

                void Receive(inout event_t event)
                        raises(ExNotRegistered);

        } ;

    interface ConsumerAdmin: Consumer {
                void ListConsumers(out consumer_list_t consumers)
                        raises(ExNoConsumers);

                void DeleteConsumer(in string_t consumer,
                                          in uuid_t uuid)
                        raises(ExInvalidName, ExInsufficientPermission);

                void AdminDeleteFilterFromGroup(in string_t consumer,
                                          in uuid_t uuid,
                                          in filtername_list_t filter_name)
                        raises(ExInvalidName, ExInvalidFilter, ExInsufficientPermission);

                void AdminAddFilterToGroup(in string_t consumer,
                                          in uuid_t uuid,
                                          in filtername_list_t filter_name)
                        raises(ExInvalidName, ExInvalidFilter, ExInsufficientPermission);

                void AdminGetFilterGroup(in string_t consumer,
                                          in uuid_t uuid,
                                          out filtername_list_t filter_name)
                        raises(ExInvalidName, ExInsufficientPermission, ExNoFilters);

        } ;

        //
        // Supplier Interface
        //

        interface Supplier {
                void PushSupplierRegister(in CosEventComm::PushSupplier supplier)
                        raises(ExAlreadyRegistered, ExInsufficientPermission);

                void PullSupplierRegister(in CosEventComm::PullSupplier supplier)
                        raises(ExAlreadyRegistered, ExInsufficientPermission);

                void UnRegister(in supplier_t supplier)
                        raises(ExNotRegistered, ExInsufficientPermission);

                void Send(in event_t event)
                        raises(ExNotRegistered, ExInsufficientPermission);

                void GetRegistration(out supplier_list_t push_suppliers,
                                        out supplier_list_t pull_supplier)
                        raises(ExNoSuppliers);

        } ;

interface SupplierAdmin: Supplier {
                void ListSuppliers(out supplier_list_t suppliers)
                        raises(ExInsufficientPermission, ExNoSuppliers);

                void DeleteSupplier(in string_t supplier,
                                        in uuid_t uuid)
                        raises(ExInvalidName, ExInsufficientPermission);

        } ;

        //
        // Event Iterator Interface
        //

        interface EventIterator {
                ems_boolean NextOne(out event_t event);

                ems_boolean NextN(in ems_ulong_int how_many,
                                        out event_list_t events);

                void Destroy( );

        } ;

        //
        // Registry Interface
        //

interface Registry {
                Supplier ForSupplier( );

                Consumer ForConsumer( );
```

```
} ;

interface RegistryAdmin: Registry {
        void ListAttributes(out attrlist_t attributes);

        void GetUndeliveredEvents(in string filter_name,
                                  in unsigned long how_many,
                                  out event_list_t events,
                                  out EventIterator ei)
                raises(ExNoEvents, ExFilterNotFound, ExInsufficientPermission);

        void DeleteUndeliveredEventsByFilter(in string filter_name)
                raises(ExNoEvent, ExFilterNotFound, ExInsufficientPermission);

        void DeleteUndeliveredEvent(in eventid_t event)
                raises(ExNoEvent, ExInsufficientPermission);

        void Forward(in filtername_list_t filter_group,
                                   in string_t host,
                                   out string_t name,
                                   out uuid_t uuid)
                raises(ExForwardingEventServiceNotThere, ExForwardingEventLoop,
                        ExInsufficientPermission);

} ;

//
// Security Interface
//

interface Security {
        void Edit(in secobj_t secobj,
                                  in secsubj_t subject,
                                  in secperm_t newperm,
                                  out secperm_t oldperm)
                raises(ExInvalidName, ExInsufficientPermission);

        void Read(in secobj_t secobj,
                                  in secsubj_t subject,
                                  out secperm_t oldperm)
                raises(ExInvalidName, ExInsufficientPermission);

        void SubjAdd(in secsubj_t subject,
                                  in secprin_t principal)
                raises(ExInvalidName, ExInsufficientPermission);

        void SubjDelete(in secsubj_t subject)
                raises(ExInvalidName, ExInsufficientPermission);

        void SubjGet(in secprin_t principal,
                                   out secsubj_t subject)
                raises(ExInvalidName, ExInsufficientPermission);

} ;

//
// Administration Interface
//
```

```
        interface Management;

        typedef sequence<Management> EventManagementSystems;

        interface Management {
                readonly attribute EventManagementSystems Systems;

                Registry ObtainRegistry( )
                        raises(ExInsufficientPermission);

                Security ObtainSecurityManagement( )
                        raises(ExInsufficientPermission);

                EventType ObtainTypeRepository( )
                        raises(ExInsufficientPermission);

                EventFilter ObtainFilterRepository( )
                        raises(ExInsufficientPermission);

        } ;
} ;
```

Preliminary Specification

Part 3:

Event Structures for the Basic Event Set

The Open Group

Chapter 14

Event Objects

14.1 CMIP Event Objects

CMIP defines an event report which is used to report an event to a peer *CMISE-service-user*. Its arguments are given in the structures on the following page.

```
EventReply ::= SEQUENCE {
    eventType                EventTypeId,
    eventReplyInfo           [8] ANY DEFINED by eventType OPTIONAL }

EventReportArgument ::= SEQUENCE {
    managedObjectClass       ObjectClass,
    managedObjectInstance    ObjectInstance,
    eventTime                [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType                EventTypeId,
    eventInfo                [8] ANY DEFINED BY eventType OPTIONAL }

EventReportResult ::= SEQUENCE {
    managedObjectClass       mapping ObjectClass OPTIONAL,
    managedObjectInstance    ObjectInstance OPTIONAL,
    currentTime              [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventReply               EventReply OPTIONAL }

EventTypeId ::= CHOICE {
    globalFor                [6] IMPLICIT OBJECT IDENTIFIER,
    localFor                 [7] IMPLICIT INTEGER }

ObjectClass ::= CHOICE {
    globalFor                [6] IMPLICIT OBJECT IDENTIFIER,
    localFor                 [7] IMPLICIT INTEGER }

ObjectInstance ::= CHOICE {
    distinguishedName        [2] IMPLICIT DistinguishedName,
    nonSpecificFor           [3] IMPLICIT OCTET STRING,
    localDistinguishedName   [4] IMPLICIT RDNSequence }

AttributeValueAssertion ::= SEQUENCE  {
    attributeType            AttributeType ,
    attributeValue           AttributeValue    }

Attribute ::= SEQUENCE    {
    type                     AttributeType ,
    values                   SET OF AttributeValue   }

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

DistinguishedName ::= RDNSequence

Name ::= CHOICE  {
    rDNSequence              RDNSequence   }
```

These arguments map down to the set of basic attribute types listed in Table 14-1.

Attribute Name	**Attribute Format**	**Description**
ANY	`es_c_attr_bytes`	Supports the capability to pass arbitrary ASN.1 syntax
GeneralizedTime	`ems_c_attr_utc`	Time event occurred
OBJECT IDENTIFIER	`es_c_attr_ulong_int*`	Sequence of integers to uniquely identify anything
INTEGER	`es_c_attr_ulong_int`	Integer value
OCTET STRING	`es_c_attr_char_string`	Sequence of bytes

Table 14-1 Mapping for CMIS Event Structure Data Types

14.2 DCE SVC Event Objects

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 apping from an SVC message to an XEMS Event.

The data declaration of the SVC data structure is as follows:

```
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;
```

Index	Attribute Name	Attribute Format	Description
0	*version*	`es_c_attr_ulong_int`	Version number of interface that generated message
1	*t*	`es_c_attr_utc`	Time message was written
2	*argtypes*	`es_c_attr_char_string`	The format-specifier string for the message
3	*table_index*	`es_c_attr_ulong_int`	Subcomponent table index
4	*attributes*	`es_c_attr_ulong_int`	Message attributes, OR'd together
5	*essage_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*	`es_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*	`es_c_attr_ulong_int`	Line number where message came from
10	*threadid*	`es_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*	`es_c_attr_ushort_int`	Debug attribute of message
14	*attribute.severity*	`es_c_attr_ushort_int`	SVC Severity of message
15	*attribute.actroute*	`es_c_attr_ulong_int`	Action / Routing attribute of message
16			SVC Argument 1
...			
15+N			SVC Argument N

Table 14-2 Mapping for a DCE SVC Message

14.3 EMS Event Objects

The following set of events are supplied by the Event Service itself. These events are supplied to notify when defined event service events occur.

14.3.1 Event Service Action Notification Event

This event type is supplied when the event service performs an action such as register a consumer or supplier, or add or delete an event type or filter.

Type: e433700c-b3cd-11cf-9550-10005a4f3556

Name: EventServiceActionNotification

Size: 33

Index	Attribute Name	Attribute Format	Description
0	*EStype*	`es_c_attr_ushort_int`	Event Service event type
1	*name*	`ems_c_attr_char_string`	Name depending on EStype
2	*uuid*	`es_c_attr_uuid`	UUID depending on EStype

Table 14-3 Event Service Action Notification Event

Possible values for **EStype** are:

ESconsumerRegister
notifies when a consumer has registered.

name	Consumer name
uuid	Consumer uuid

ESsupplierRegister
notifies when a supplier has registered.

name	Supplier name
uuid	Supplier uuid

ESconsumerConnect
notifies when a consumer has connected.

name	Consumer name
uuid	Consumer uuid

ESconsumerUnregister
notifies when a consumer has unregistered.

name	Consumer name
uuid	Consumer uuid

ESeventTypeAdd
notifies when an even type has been added.

name	Event Type name
uuid	Event Type uuid

ESeventTypeDelete
notifies when an event type has been deleted.

name	Event Type name
uuid	Event Type uuid

ESfilterAdd
notifies when a filter was added.

name	Filter name
uuid	Filter uuid

ESfilterDelete
notifies when a filter has been deleted.

name	Filter name

uuid	Filter uuid

ESfilterModify
notifies when a filter has been modified.

name	Filter name
uuid	Filter uuid

14.3.2 Event Service Queue Full Event

This event type is supplied when the event service event queue is full. If the consumer name is specified, then that consumer's event queue is full, otherwise, the event service event queue is full.

Type: e67902ca-bfce-11cf-a767-10005a4f3556

Name: EventServiceQueueFull

Size: 33.

Index	Attribute Name	Attribute Format	Description
0	*QueueSize*	`es_c_attr_ulong_int`	Event queue size (that is, number of events in Queue)
1	*name*	`ems_c_attr_char_string`	Consumer name if this is a consumer event queue
2	*uuid*	`es_c_attr_uuid`	Consumer UUID

Table 14-4 Event Service Queue Full Event

14.3.3 Event Service Error Event

This event type is supplied when an event service error occurs. The error message is a *printf* style of error message with the arguments being in attributes 1 thru N.

Type: ff3c46fe-bfcf-11cf-b01c-10005a4f3556

Name: EventServiceError

Size: 1.

Index	Attribute Name	Attribute Format	Description
0	*ErrorMsg*	`es_c_attr_char_string`	Event Service error message
1			Event Service error message Argument 1
. . .			
1+N			Event Service error message Argument N

Table 14-5 Event Service Error Event

14.3.4 Event Service Undelivered Event Notification Event

This event type is supplied when the event service fails to deliver an event to a consumer.

Type: ff3c46ff-bfcf-11cf-b01c-10005a4f3556

Name: UndeliveredEvent Notification

Size:

Index	Attribute Name	Attribute Format	Description
0	*Consumer*	`ems_c_attr_bytes`	The consumer structure, ems_consumer_t, associated with the targeted consumer
1	*Supplier*	`es_c_attr_bytes`	The supplier structure, `es_supplier_t`, associated with the supplier of the undeliverable event
2	*EventHeader*	`es_c_attr_bytes`	The event header structure, es_hdr_t, associated with the undeliverable event

Table 14-6 Event Service Undelivered Event Notification Event

14.3.5 Event Service Consumer Filter Group Changes

This event type is supplied when the event service changes the number of filters associated with an active consumer filter group.

Type: ff3c4701-bfcf-11cf-b01c-10005a4f3556

Name: ConsumerFilterGroup

Size:

Index	Attribute Name	Attribute Format	Description
0	*type*	`es_c_attr_ushort_int`	Action type
1	*name*	`ems_c_attr_char_string`	Consumer name
2	*uuid*	`es_c_attr_uuid`	Consumer uuid
3	*index*	`es_c_attr_ulong_int`	Event handler function index for a given consumer environment

Possible values for **action** types are:

ESaddFilterToGroup
notifies when an event filter has been added to a group.

ESdeleteFilterFromGroup
notifies when an event filter has been deleted from a group.

14.3.6 Event Service Consumer Interest

This event type is supplied when the event service changes the number of consumers interested in an event type.

Type: ff3c4700-bfcf-11cf-b01c-10005a4f3556

Name: ConsumerEventInterest

Size:

Index	Attribute Name	Attribute Format	Description
0	*name*	`ems_c_attr_char_string`	Event type name
1	*uuid*	`es_c_attr_uuid`	Event type uuid
2	*count*	`es_c_attr_ulong_int`	Number of interested consumers

14.4 SNMP Event Objects

SNMP defines an event report which is used to report an event from a SNMP proxy manager.

Index	Attribute Name	Attribute Format	Description
0	*version*	`es_c_attr_ulong_int`	SNMP Version number that generated this message
1	*community*	`ems_c_attr_byte_string`	Community name
2	*Trap-PDU*	`es_c_attr_ulong_int`	Indicates this message carries a Trap PDU
3	*enterprise*	`es_c_attr_char_string`	MIB II **sysObjectID** of object generating trap
4	*agent-addr*	`es_c_attr_byte_string`	*NetworkAddress* of object generating this trap
5	*generic-trap*	`es_c_attr_long_int`	A generic SNMP trap type
6	*specific-trap*	`es_c_attr_long_int`	Specific code is present even if generic trap is not enterpriseSpecific
7	*tie-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[7]

Table 14-7 Event Service Report Event from SNMP Proxy Manager

7. An SNMP OBJECT IDENTIFIER type which needs to be applied to ARRAY of Integer. Currently this EMS does not support this construct.

Table 14-8 Mapping for SNMP Event Structure Data Types

SNMP Primitive Types[9]	**XEMS Attribute Format**	**Description**
INTEGER	`Any es_c_attr_xxxx_int`	Variable. The ASN.1 encoding for each variable provides its size information.
OCTET STRING	`es_c_attr_byte_string`	May contain value X'00'.
OBJECT IDENTIFIER	`es_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	`es_c_attr_????`	ANSI NULL. Currently this EMS 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		
IpADDRESS	`es_c_attr_byte_string`	Currently, only the Internet protocol faily is defined for NetworkAddress CHOICE.
Counter	`es_c_attr_ulong_int`	Non-negative integer which monotonically increases and wraps to 0 after reaching maximum value (2ˆ32-1).
Gauge	`es_c_attr_ulong_int`	Non-negative integer which may increase or decrease, but which latches at a maximum value (2ˆ32-1).
TimeTicks	`ems_c_attr_ulong_int`	Non-negative integer which counts the tie in hundredths of a second since some epoch.
Opaque	`es_c_attr_bytes`	Supports the capability to pass arbitrary ASN.1 syntax.
Common Constructs		
RequestId	`es_c_attr_long_int`	Used by SNMP application entities to correlate incoming responses with outstanding requests.

SNMP Primitive Types[9]	**XEMS Attribute Format**	**Description**
ErrorStatus	`es_c_attr_long_int`	A non-zero instance is used to indicate an exception.
ErrorIndex	`es_c_attr_ulong_int`	May provide additional information by indicating which variable in a list caused the exception.
VarBind	`es 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	`es_c_attr_ulong_int`	0
GetNextRequest-PDU	`es_c_attr_ulong_int`	1
GetResponse-PDU	`es_c_attr_ulong_int`	2
SetRequest-PDU	`es_c_attr_ulong_int`	3
Trap-PDU	`es_c_attr_ulong_int`	4
MIB II		
DisplayString	`es_c_attr_char_string`	Printable display string restricted to the NVT ASCII character set defined in RFC 854.
PhysAddress	`es_c_attr_byte_string`	Used to represent edia- or physical-level addresses.

9. 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).

Glossary

consumer
A Consumer processes event data, for example, a server application which registers for, receives, and processes event data.

EMS
Event Management Service.

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, 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. 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, that is, selection by time or type.

ISV
Independent Software Vendor.

manager
A manager is the initiator of an event management interaction.

MAScOTTE
MAnagement Services for Object-oriented disTributed sysTEms. This is a European-based project of partner companies with some funding support from the European Commission. The project goal is to enable management solutions for CORBA-based environments by use not only of CORBA applications, but also through a gateway to external (non-CORBA) applications so that existing management solutions can be extended to management of the CORBA environment. The MAScOTTE project is due to be completed by October 1997. It bases part of its work on The Open Group specifications for CORBA services, and provides feedback on these specifications. Further information on the MAScOTTE project may be obtained from the

MAScOTTE Web server (http://www.esrin.esa.it/htdocs/MAScOTTE).

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

OQL
Object Query Language.

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

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

registration
Registration is a mechanism by which a management application can indicate an interest in receiving notification on specific events.

supplier
A Supplier produces event data.

utc
Universal Coordinated Time.

uuid
universal unique identifier.

Index