Open Software Foundation
Request For Comments: 67.0
January 1996

R. Cohen (IBM)
G. Wilson (IBM)

DCE EVENT MANAGEMENT SERVICE

1. INTRODUCTION

In both traditional (SNMP and CMIP) and object system management architectures, communications between the managing and managed systems is bidirectional. One or more managing systems can send requests to query and control various aspects of resources being managed. In addition, the managed resource must be capable of sending asynchronous notifications or events to the managing systems. An event marks a change in state of the managed resource that causes a notification to be sent to interested parties. The routing of these events is usually done via an agent on the managed system.

In order to support remote management of DCE resources, enhancements must be made to DCE to support asynchronous event management for use by system management applications. This document describes the DCE Event Management Service (EMS) which provides asynchronous event support for DCE based applications.

EMS uses the concepts of `event suppliers` and `event consumers`, and sets up an `event channel` between them to support asynchronous communication. In the context of DCE, event suppliers are any DCE core service or DCE-based application (client or server), and event consumers can be any application with an interest in receiving asynchronous events from one or more DCE processes. The transmission of events between suppliers and consumers is uncoupled by routing events via EMS which is the implementation of an event channel. EMS also provides a filtering mechanism to allow administrators and consumers control over which events EMS will send.

The initial release of EMS provides integration for DCE clients and servers using the DCE Serviceability (SVC) or Audit interfaces. DCE applications can use the APIs offered in SVC and Audit to become event suppliers. Future releases of EMS will provide full support for application-defined event types.

2. TARGET

The DCE EMS is targeted for use by system administration applications that enable local or remote monitoring of asynchronous events generated by DCE core and application services. In the initial release, EMS will communicate server events via the SVC or Audit layer to the DCE SNMP subagents.

3. GOALS

The goals of the DCE EMS are as follows:

(a) The effort involved in writing a DCE event consumer application should be minimized.

(b) The EMS service should have good performance, and should minimize network and system load.

(c) EMS should be designed to be a centralized service, even though the first release will not be centralized.

(d) Event consumer applications should be able to locate and register with one or more EMSs on multiple DCE hosts.

(e) Events should be sent by DCE applications once and only once, via the SVC or Audit Interface, to EMS.

(f) Event consumer applications should be able to control both which DCE host is sending events as well as which event types are sent.

(g) EMS can transmit events from multiple DCE event supplier applications to one or more DCE event consumer applications based on a defined set of event filters within a DCE cell in a secure way.

(h) DCE event supplier applications should not be aware of what DCE consumer applications have registered to receive events.

(i) EMS should provide a remote management API.

(j) EMS should provide reliable delivery of events to consumers and be tolerant of network and machine failures.

(k) EMS should provide the ability to define and extend events and event contents.

(l) The EMS architecture should be based on the OMG Event Service Specification as documented in the Joint Object Services Submission as defined in COSS (see [OMG]).

(m) The first release of EMS should support SVC and Audit events.

3.1. Non-Goals

(a) EMS should not be used for monitoring DCE core services start-up failures. EMS uses CDS and Security, and requires them to be running in the cell to start up.

(b) The first release of EMS will not support general message passing, only SVC and Audit events.

4. TERMINOLOGY

(a) **attribute name**

A string uniquely identifying an attribute of a given event type.

(b) **attribute operator**

The operator in a filter expression used in the comparison between the named attribute in the event and the attribute value.

(c) **attribute type**

The data type of an event attribute. This defines the format of the data in this attribute.

(d) **attribute value**

The value in a filter expression which is compared, using the attribute operator, against the event value for the specified attribute name.

(e) **event**

The data to be transmitted between an event supplier and EMS, and EMS and one or more event consumers. An event consists of an event header, and a list of event attributes which contain the event type specific data.

(f) **event channel**

A service that decouples the communications between event suppliers and event consumers. An event channel is both a supplier and consumer of events.

(g) **event consumer**

A DCE server application which registers for, defines filters for, receives, and processes event data.

(h) **event filter**

A collection of one or more filter expressions which are logically ANDed together. An event filter has an event filter name.

(i) **event filter expression.**

A 3-tuple consisting of an attribute name, attribute operator, and an attribute value, which defines a compare operation.

(j) **event filter group**

A collection of one or more event filters which are logically ORed together. As many event filter names as desired can be added to a consumer's event filter group to control what events EMS will send to that consumer. Event filter groups can contain event filters which specify heterogeneous event types.

An event must pass one set of the event filters in the event filter group before it will be forwarded to an event consumer.

(k) **event log**

A collection of events to be sent to consumers. The event log stores events in case EMS cannot forward events to all appropriate event consumers. An event is deleted from the event log after that event is forwarded to all consumers.

(l) **event supplier**

A DCE based application (including DCE core servers) which emits event data. In the first release, a supplier can be any DCE application that uses SVC or Audit.

(m) **event type**

A class of events that have the same event format. An event type format is described via an event type schema. An event type is defined by a unique uuid.

(n) **event type schema**

A description of an event type. It consists of a list of attribute name/type pairs which specify the data format of an event. The event type schema can be used by consumers or EMS administrators to construct event filters and event content.

5. REQUIREMENTS

(a) EMS should not be required for use of SVC or Audit.

(b) The DCE application developer should be able to send events using the SVC or Audit subsystem without instrumenting directly to EMS.

(c) EMS should be capable of event persistence and event retransmission to the appropriate DCE event consumers in the case of network or system failure.

(d) The initial version of EMS should support a minimally acceptable subset of the total function to assure its timely availability as part of the DCE core product.

(e) All DCE coding requirements should be followed. All code should be internationalized, and should be portable across all DCE platforms.

6. FUNCTIONAL DEFINITION

The DCE EMS manages event services in a DCE cell. EMS consists of two parts: the `emsd` server, and the API to access event services.

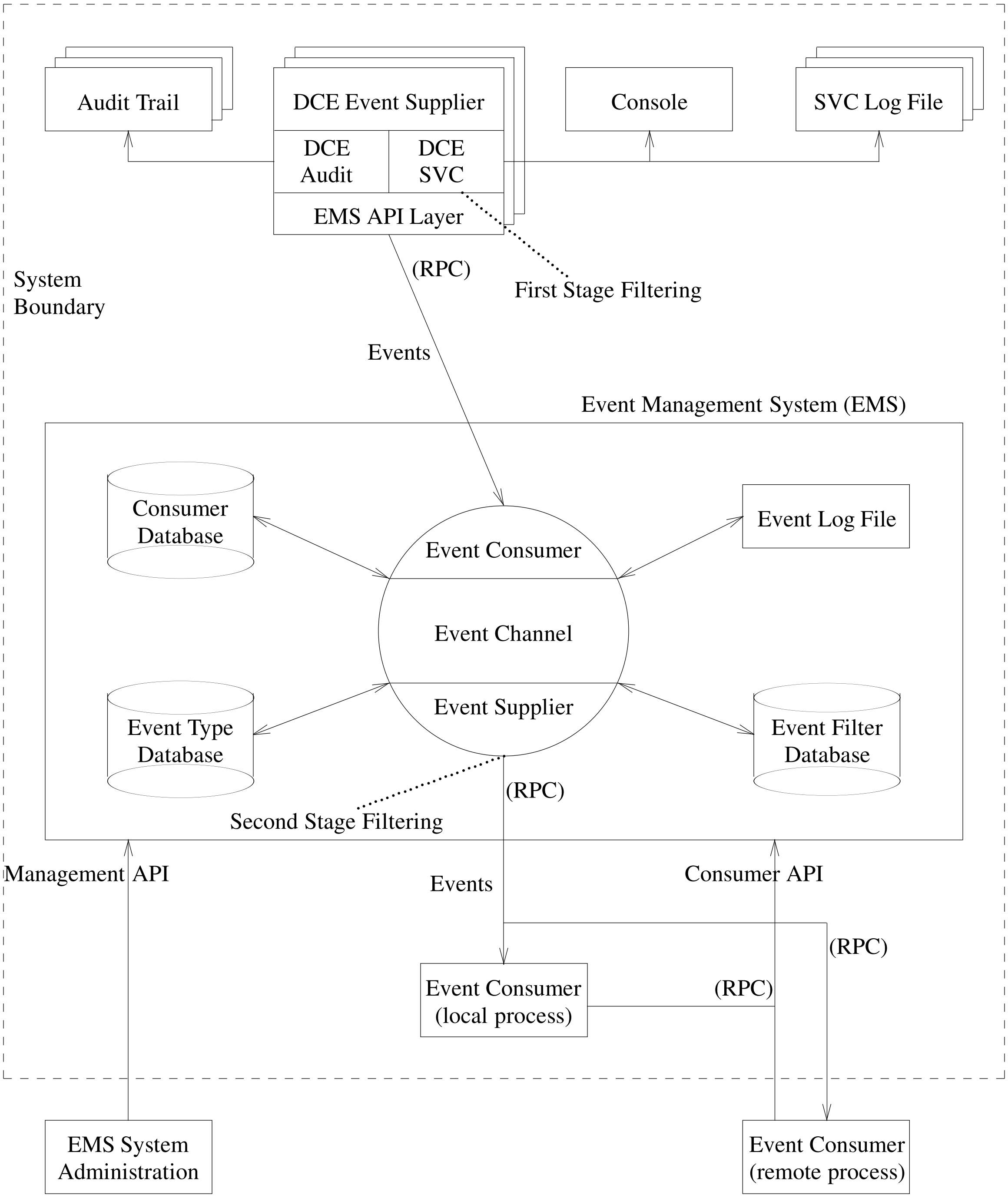
`emsd` is a DCE server which, in the first release, will be decentralized, i.e., will reside on every DCE host in the cell from which consumers will request events (future EMS releases will be centralized). Three interfaces are exported by `emsd`, to support EMS suppliers, consumers, and event services administration. The EMS API provides an interface to the same three interfaces for use by EMS clients.

The following sections describe how EMS works, along with a description of how to define event types, the event flow through EMS, how event filters work, how events get logged, and the security model used to protect both event data, and event filter definitions.

6.1. Event Flow Description

EMS sets up an event channel to decouple the communications between the supplier and consumer (see Figure).

In order to start receiving events, an event consumer must first register with EMS, then set up an event filter group to tell EMS which events to forward to that consumer. Before a supplier can send an event to EMS, the event must pass through the first stage event filter.



In the first release of EMS, first stage filtering is achieved at the supplier level by using SVC and Audit facilities. See Section 6.3, ''Event Filters'' for more details.

Once the event passes through the first stage filter, it is sent to EMS. EMS writes the event to the EMS event log in order to save the event in case the event cannot be immediately delivered.

Once the event reaches EMS, then it must pass through the second stage filter before being forwarded to interested consumers. EMS goes through the list of registered consumers and uses the event type schema from the event type Database and the consumer's event filter group from the consumer Database, and the event filters from the Event Filter Database to determine if this event passes through to be forwarded on for each event consumer. After all appropriate consumers receive the event, then the event is removed from the event log.

6.2. Event Type Definition

The format of EMS event types are defined by event type schemas, and are kept in the EMS event Type Database. The event type schemas consist of a list of attribute names along with the attribute type which specifies the data format of the data associated with that attribute. Events consist of a fixed header part, and a variable length data part. The variable length data part consists of N self-defining data items which consist of an attribute type, then the data itself. See Section 7.1.3, ‘‘Event Attribute’’. The data elements in the variable length part correspond in order, one for one with the list of attribute name/type pairs defined in the event type schema. The variable length part can have additional unnamed data items at the end of the named data items (i.e., the data items that have attribute names).

The event type schemas are used in several different ways.

A consumer can request a list of supported event types, and pick which events types it wants to receive by using the event type schemas to construct event filters, and to map event data according to attribute names (e.g., an event consumer can reconstruct an SVC message for example by using the attribute names to find the correct data items).

Suppliers use event type schemas to define new event types that they intend to produce.

EMS uses the event type schemas to apply event filters to events.

6.2.1. Generic event types

EMS supports events with type `Generic`. Such generic events do not have an event type schema. The only way to define filters for generic events is to use filter expressions with event header attributes (See Table 5, ‘‘Event Header Attributes’’).

6.2.2. Default event types

Following are the definitions of the SVC and Audit event attribute lists which are contained in their event type schemas. See Section 7.3.1, ‘‘Event Type Schema’’, for a definition of the event type schema data structure:

```
#define CNT_SVC_ATTRS (sizeof(svc)/sizeof(ems_attribute_t))

static ems_attribute_t svc[] = {
    {(unsigned char *)"version",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"t",
        {ems_c_attr_utc,0}},
    {(unsigned char *)"argtypes",
        {ems_c_attr_char_string,0}},
    {(unsigned char *)"table_index",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"attributes",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"message_index",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"format",
        {ems_c_attr_char_string,0}},
    {(unsigned char *)"file",
        {ems_c_attr_char_string,0}},
    {(unsigned char *)"progname",
        {ems_c_attr_char_string,0}},
    {(unsigned char *)"line",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"threadid",
        {ems_c_attr_ulong_int,0}},
    {(unsigned char *)"component_name",
        {ems_c_attr_char_string,0}},
    {(unsigned char *)"sc_name",
```

```
            {ems_c_attr_char_string,0}},
        {(unsigned char *)"attribute.debug",
            {ems_c_attr_ushort_int,0}},
        {(unsigned char *)"attribute.severity",
            {ems_c_attr_ushort_int,0}},
        {(unsigned char *)"attribute.actroute",
            {ems_c_attr_ulong_int,0}}
    };

    #define CNT_AUDIT_ATTRS (sizeof(audit)/sizeof(ems_attribute_t))

    ems_attribute_t audit[] = {
        {(unsigned char*)"format",
            {ems_c_attr_ushort_int,0}},
        {(unsigned char*)"server",
            {ems_c_attr_uuid,0}},
        {(unsigned char*)"event",
            {ems_c_attr_ulong_int,0}},
        {(unsigned char*)"outcome",
            {ems_c_attr_ushort_int,0}},
        {(unsigned char*)"authz_st",
            {ems_c_attr_ushort_int,0}},
        {(unsigned char*)"time",
            {ems_c_attr_utc,0}},
        {(unsigned char*)"addr",
            {ems_c_attr_char_string,0} }
    };
```

Several constants have also been defined to use to match against the `attribute.severity` attribute. They are:

(a) `SVC_C_SEV_FATAL`

(b) `SVC_C_SEV_ERROR`

(c) `SVC_C_SEV_WARNING`

(d) `SVC_C_SEV_NOTICE`

(e) `SVC_C_SEV_NOTICE_VERBOSE`

6.3. Event Filters

EMS supports the concept of *two stage filtering*. First stage filtering is applied on the machine that the supplier is running on, to filter out events before they are even sent to EMS; this can also be called supplier-side filtering. Second stage filtering is applied by EMS to events received from suppliers. This stage controls which events get forwarded on to consumers.

The following sections describe the two stages of filtering EMS supports, as well as how to define and organize the event filters.

6.3.1. First and second stage filtering

EMS supports two levels of filtering. The first stage filter is applied at the event supplier, and only events that pass this filter get transmitted to EMS. First stage filtering is also referred to as supplier side filtering.

In the first release of EMS, only SVC and Audit event types are supported. The first stage filter for SVC events will be the SVC routing destination See Section 13.0, ‘‘Seamless Integration With Audit and SVC’’ for more details on how to route SVC messages to EMS. Only SVC messages that are routed to EMS will be sent through the EMS event channel. For Audit, only audit records that pass through the audit filter mechanism will be sent through the EMS event channel, See the OSF DCE Administration Guide — Core Components, ‘‘DCE Audit Service’’, for more information on how to use Audit filters.

The second stage filter is defined per EMS consumer. This stage associates an event filter group with each event consumer, and that consumer will only receive events that pass through one of the entries in the consumer's event filter group.

6.3.2. Defining event filters

A typical scenario of how an event consumer would start interfacing with EMS would be for the consumer to first query EMS about what event types are supported. From that list of event types, the consumer could then query the event type schema, to construct any event filters for that event type. The event type schemas contain the list of attribute names and attribute types which the consumer would use to construct an event filter. The consumer would then tell EMS to apply the constructed event filter to incoming events by adding it to the consumers event filter group.

EMS filters are constructed at several different levels. The lowest level is the filter expression, which is a 3-tuple consisting of an attribute name, attribute operator, and an attribute value which defines a compare operation. All compare operations evaluate to a boolean value. Only certain attribute operators are allowed on certain attribute types. See Table 8 ‘‘Filter Expression Operator Table’’, in Appendix B ‘‘Valid Filter Expressions’’.

Following are some examples of a filter expression:

```
ems_filter_exp_t xmp_SVC, xmp_Audit;

xmp_SVC.attr_name
    = "file";
xmp_SVC.attr_operator
    = ems_c_attr_op_eq;
xmp_SVC.attr_comp_value.type
    = ems_c_attr_char_string;
xmp_SVC.attr_comp_value.tagged_union.char_string
    = "file.c";

xmp_Audit.attr_name
    = "outcome";
xmp_Audit.attr_operator
    = ems_c_attr_op_eq;
xmp_Audit.attr_comp_value.type
    = ems_c_attr_ushort_int;
xmp_Audit.attr_comp_value.tagged_union.ushort_int
    = aud_c_esl_cond_denial;
```

The first example will evaluate to TRUE if the attribute name `"file"` of an SVC event type is equal to the string `"file.c"`. In the second example, the filter expression `xmp_Audit` will evaluate to TRUE if the attribute name `"outcome"` of an Audit event type is equal to the attribute value `aud_c_esl_cond_denial`, which means that an Audit event is recording that access was denied for some action (See the OSF DCE Application Development Reference, ‘‘DCE Security Service’’, for more info on the outcome attribute).

An event filter is made up of a list of filter expressions that are ANDed together. An event filter has both a name and an event type. The event filter name is used to refer to the event filter in event filter create, delete and update operations. The event filter name is also used by consumers and EMS administrators to add event filters to an event filter group.

A consumer's event filter group is a collection of one or more event filter names. For an event to pass through a consumer's filter group, the event filter associated with each event filter name specified in the event filter group is applied to the event. If all the event filter expressions in the named event filter evaluate to TRUE, then the event is forwarded by EMS to that consumer. If any of the event filter expressions evaluate to FALSE, then the event filter specified by the next event filter name in the event filter group is applied to the event. If none of the event filters in the event filter group evaluate to TRUE, then the event is not forwarded to that consumer.

Using the consumer API, a consumer can define a new event filter, and add it to an event filter group. A consumer can also get a list of existing event filter names, and add one of those event filter names to an event filter group. See Section 9.3, ''Event Filter Construction Interface'' for more information on the APIs to retrieve existing event FIlters as well as build new event filters.

Event filter names can be added or deleted from event filter groups by the consumer, as well as by the administrative interface.

6.3.3. Filtering on header information or event type

A predefined set of event header attribute name/type pairs have been defined to allow filtering on the information in the event header. See Table 5, ''Event Header Attributes'', for more details. Filter expressions with event header attributes can be part of filters with event type specific header attributes. Filters with type `Generic` can only have filter expressions with header attributes.

6.4. EMS Event Log

The EMS event log is used to store events in case of EMS failures. EMS writes all events to the event log, and delete the event record once the event has been transmitted to all consumers that are supposed to get the event.

The event log is kept in a file on the machine where `emsd` is running. Events are stored in a directory specified by the environment variable `EMS_EVENTLOG_DIR`. See Section 11.6.1, ''EMS Environment Variables'' for more information.

An API is provided to examine local event logs. See Section 11.4, ''EMS Log Information Management Interface''.

6.5. EMS Security Model

EMS provides for secure manipulation of data in the EMS databases. This includes the Event Filter database, the Event Type database, as well as the list of consumers in the Consumer database. EMS also provides for supplier and consumer authentication and authorization as well as secure transmission of event data.

All ACLs will be associated with names in the DCE namespace, and the `emsd` server will manage the namespace past the junction:

/.:/hosts/*hostname*/ems-server/

The ACL associated with this object will control access to the `emsd` server registered in this namespace. The permissions associated with `ems-server` are:

Perm bit	Name	Description
`c`	control	Modify the ACLs on the server
`r`	read	Read the attributes for this server
`s`	stop	Stop the EMS server
`w`	write	Modify the attributes on this server

Table 1: `emsd` Server Permission Bits

Three security objects will be maintained under the `ems-server` junction. The directories and the databases they represent are:

(a) `event-types` — Event type database.

(b) `filters` — Filter database.

(c) `consumers` — Consumer database.

Each of these databases will have an ACL associated with it, as discussed next.

6.5.1. Event type security management

The Event Type database will be represented by the following name in the DCE name space:

`/.:/hosts/`*`hostname`*`/ems-server/event-types`

The ACL associated with this object will control access to this database. The permissions associated with `event-types` are:

Perm bit	Name	Description
`c`	control	Modify the ACLs on the event type
`d`	delete	Delete an event type schema
`i`	insert	Add an event type schema
`r`	read	Read the contents of event type schemas

Table 2: Event Type Database Permission Bits
(Only `c`, `r` supported in release 1.)

EMS event data access can be granted per event type. Authority on event data of a given event type can be granted by modifying the ACL on:

`/.:/hosts/`*`hostname`*`/ems-server/event-types/`*`event_type_name`*

where *`event_type_name`* is the event type name which appears in the event type schema (see data structures section which describes the event type schema). The names will be recognized for SVC and Audit events are:

`/.:/hosts/`*`hostname`*`/ems-server/events/SVC`
`/.:/hosts/`*`hostname`*`/ems-server/events/Audit`

The permissions associated with *`event_type_name`* are:

Perm bit	Name	Description
`c`	control	Modify the ACLs on the event type
`d`	delete	Delete this event type
`r`	read	Read (consume) an event of this type
`w`	write	Write (supply) an event of this type

Table 3: Event Type Permission Bits
(Only `c`, `r`, `w` supported in release 1.)

A supplier's rights will be verified on the first event send to EMS, and the consumer's rights will be verified before forwarding events to that consumer. Authenticated RPC will be used to access the EMS supplier and consumer Remote API.

6.5.2. Event Filter Security Management

The Filter database will be represented by the following name in the DCE name space:

`/.:/hosts/`*`hostname`*`/ems-server/filters`

The ACL associated with this object will control access to this database. The permissions associated with filters are:

Perm bit	Name	Description
`c`	control	Modify the ACLs on filters
`d`	delete	Delete an event filter
`i`	insert	Add an event filter
`r`	read	Get list, or contents, of event filters

Table 4: Filter Database Permission Bits

Event filter access control can be granted per event filter. Authority on filter access for a given event filter can be granted by modifying the ACL on:

`/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*

where *`filter_name`* is the event filter name given the event filter on the call to `ems_filter_add()` (See Section 9.3.1, ‘‘Add an Event Filter’’).

The permissions associated with event filters are:

Perm bit	Name	Description
`c`	control	Modify the ACL on the event filter
`d`	delete	Delete the event filter
`w`	write	Modify the contents of an event filter

Table 5: Event Filter Permission Bits

When a consumer creates an event filter, that consumer principal automatically gets `cdw` permissions on the created event filter.

6.5.3. Consumer security management

The Consumer database will be represented by the following name in the DCE name space:

`/.:/hosts/`*`hostname`*`/ems-server/consumers`

The ACL associated with this object will control access to this database. The permissions associated with `consumers` are:

Perm bit	Name	Description
`c`	control	Modify the ACLs on `consumers`
`d`	delete	Delete a consumer
`i`	insert	Add (register) a consumer
`r`	read	List consumer information
`w`	write	Modify a consumer, including filter group

Table 6: Consumer Database Permission Bits

6.5.4. EMS security initialization

When EMS is configured, several security groups will be created by default. The groups are `ems-admin`, `ems-consumer`, and `ems-supplier`. The default permissions will be:

(a) `/.:/hosts/`*`hostname`*`/ems-server`

(i) Object ACL:

```
ems-admin:crws
hosts/hostname/self:rws
any_other:r
```

(b) `/.:/hosts/`*hostname*`/ems-server/event-types`

(i) Object ACL:

```
ems-admin:cri
ems-consumer:r
ems-supplier:ri
any_other:r
```

(ii) Initial object ACL (`/.:/hosts/`*hostname*`/ems-server/event-types/`*event_type_name*):

```
ems-admin:cdw
ems-consumer:r
ems-supplier:w
```

(c) `/.:/hosts/`*hostname*`/ems-server/filters`

(i) Object ACL:

```
ems-admin:crdi
ems-consumer:ir
any_other:r
```

(ii) Initial object ACL (`/.:/hosts/`*hostname*`/ems-server/filters/`*filter_name*):

```
ems-admin:cw
```

(d) `/.:/hosts/`*hostname*`/ems-server/consumers`

(i) Object ACL:

```
ems-admin:cdrw
ems-consumer:irw
any_other:r
```

By setting these permissions for the `ems_admin` group, each new event filter and event type created will automatically inherit the same permissions.

Administrators can add principals to each of these groups to give them access to all `emsd`'s running in a cell. If tighter security is desired, the group can be removed from the respective ACL, and principals can be added.

When transmitting events of type Audit, EMS will always use authenticated RPC set to DCE Security Protection Level strong integrity (`rpc_c_protect_level_pkt_integrity`).

6.6. Writing Consumers

Consumers are not simple clients. They have to be implemented as servers. This means that EMS consumers have certain requirements.

Consumers must:

(a) Call `consumer_start()`. This creates the consumer UUID, and then registers the EMS Transmit to Consumer interface (See Section 10.4, ‘‘EMS Transmit To Consumer Interface’’).

(b) Call `ems_consumer_handler_register()`. This sets up an incoming consumer queue, and initializes the event handler function to call the handle function specified.

(c) Register with the various `emsd`'s that it wants to receive events from. One event consumer can register to receive events from multiple `emsds` on different hosts.

(d) Set up event filters. Consumers must set up event filters before any events will be sent.

(e) Call `rpc_server_listen()`. This sets the consumer up to wait for incoming events.

6.7. EMS Recovery After Restart

EMS stores all state data in the following repositories:

(a) Consumer Database — Contains list of all suppliers, and all consumers as well as the list of filters in the consumer filter group.

(b) Event Type Database EM Contains the schemas for all the known event types.

(c) Event Filter Database — Contains all the current filter definitions.

When an `emsd` is restarted, the following steps are performed:

(a) All databases get restored.

(b) All consumers are told to reregister or reconnect.

(c) The event queue is restored from the event log files.

(d) Event transmission proceeds.

6.8. Cross-Cell Access

EMS provides full access across DCE cells for consumers only. If a consumer wishes to register with an EMS in another cell, a valid hostname (See Section 7.2.3, ‘‘Network Name’’) must be passed to the `ems_consumer_register()` routine (See Section 9.4.4, ‘‘Consumer Register’’).

6.9. Iternationalization Support in EMS

EMS supports byte data streams when passing data in an event as an event attribute. This allows I18N data to be passed from supplier to consumer. EMS restricts names to the Portable Character Set (PCS). This includes:

(a) Consumer names.

(b) Filter Names.

(c) Event Type Names.

(d) Event Attribute Names.

When a name that contains characters that are not in the PCS is passed into one of the EMS routines as an argument, that routine returns a status code of `ems_s_invalid_name()`.

7. DATA STRUCTURES

7.1. EMS Event Attributes

7.1.1. 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 unsigned16 ems_attr_type_t;
```

An event attribute type can be one of the following:

Attribute Type	Data Type	Size
ems_c_attr_small_int	idl_small_int	1 byte
ems_c_attr_short_int	idl_short_int	2 bytes
ems_c_attr_long_int	idl_long_int	4 bytes
ems_c_attr_hyper_int	idl_hyper_int	8 bytes
ems_c_attr_usmall_int	idl_usmall_int	1 bytes
ems_c_attr_ushort_int	idl_ushort_int	2 bytes
ems_c_attr_ulong_int	idl_ulong_int	4 bytes
ems_c_attr_uhyper_int	idl_uhyper_int	8 bytes
ems_c_attr_short_float	idl_short_float	4 bytes
ems_c_attr_long_float	idl_long_float	8 bytes
ems_c_attr_boolean	idl_boolean	1 byte
ems_c_attr_uuid	uuid_t	16 bytes
ems_c_attr_utc	utc_t	16 bytes
ems_c_attr_severity	ems_severity_t	2 bytes
ems_c_attr_acl	sec_acl_t *	variable
ems_c_attr_byte_string	idl_byte *	variable
ems_c_attr_char_string	idl_char *	variable
ems_c_attr_bytes	(see structure)	variable
ems_c_attr_octet_array	ems_netaddr_t *	variable

Table 7: Event Attribute Type Specifiers

Byte strings and character strings are terminated with a 0 (zero) byte. The pickling service of the IDL compiler can be used to encode complex data types into byte strings that are to be included in an EMS event.

7.1.2. Event attribute values

The event attribute value union is a self-defining data structure which has an attribute type specifier (type) which tells what type of data is in the union, and then appropriate union members to hold the value of the data specified:

```
typedef struct ems_bytes_s_t {
    unsigned32          size;    /* size of byte data */
    [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;
        case ems_c_attr_octet_array:
            ems_netaddr_t * octet_array;
        default:
            /*empty*/ ;
    } ems_attr_value_t;
```

7.1.3. Event attribute

Event attributes contain an event attribute name/type pair (`attr_name`, `attr_type`) which define an event attribute. Event attributes can be used in defining event types in event type schemas, and in defining event filters in event filter expressions. The `attr_name` specifies the attribute's `name`, and the `attr_type` specifies the format of the attribute's `value`:

```
    typedef struct ems_attribute_s_t {
        ems_string         name;    /* event attribute name */
        ems_attr_value_t   value;   /* event attribute type */
    } ems_attribute_t;
```

7.2. EMS Event Structure

7.2.1. Event type

An event type specifies the unique ID for a given event type:

```
    typedef uuid_t       ems_event_type_t;
```

Event Type	Event Type Name
`ems_c_generic_type`	Generic
`ems_c_svc_type`	SVC
`ems_c_aud_type`	Audit

Table 8: 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 5: ‘‘Event Header Attributes’’).

7.2.2. 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;    /* event type */
    uuid_t              id;      /* unique event identifier */
} ems_eventid_t;
```

7.2.3. 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_nameservice_s_t {
    ems_ns_other = 0,     /* name service other than listed */
    ems_ns_dns,           /* DNS name service */
    ems_ns_dce,           /* DCE CDS name Service */
    ems_ns_x500,          /* X500 */
    ems_ns_nis,           /* NIS */
    ems_ns_sna            /* SNA network */
} ems_nameservice_t;
```

The `ems_netaddr_t` structure specifies the actual network name. It can be interpreted according to the name service specified:

```
typedef char ems_octet_t;    /* used for NLS support */

typedef struct ems_netaddr_s_t {
    unsigned short    len;
    [size_is(len)]
    ems_octet_t       name[];  /* name in appropriate format */
} ems_netaddr_t;

typedef struct ems_netname_s_t {
    ems_nameservice_t      service;  /* name service used
                                        for netname */
    [ptr] ems_netaddr_t *  netaddr;  /* network name/address */
} 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);
```

Note that the `netname.netaddr->name` is used as a character string (`char *`) in this example, and that the NULL terminating byte is included in the length (`netname.netaddr->len`).

7.2.4. Event origin

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

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

7.2.5. Event severity

The event severity specifies the severity of the event. The names have a one-to-one correspondence to DCE SVC severity attribute values. (See OSF DCE Application Development Guide — Core Components:

```
typedef enum ems_severity_e_t {
    ems_sev_info=0,            /* 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
```

7.2.6. Event header

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

```
typedef struct ems_hdr_s_t {
    ems_eventid_t   eventid;    /* event identifier */
    ems_origin_t    origin;     /* event origin */
    ems_severity_t  severity;   /* event Severity */
    utc_t           received;   /* event received timestamp */
    utc_t           delivered;  /* event delivered timestamp */
} 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.mon	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 9: Event Header Attributes

The header attribute names correspond to elements of the event header. Additional support is provided for the received time value to allow for filtering on different parts of the received time. All received time attribute values are evaluted in GMT, and it is the responsibility of the filter creator to convert values to GMT. Following is a list of the received time attributes and how to specify them:

(a) `received.tod` — Time in format HH[[MM]SS] where HH is the Hour (00–23), MM is the Minute (00–59), and SS specifies seconds (00–59). If HH and/or SS is not specified, then the value 00 is assumed.

(b) `received.mday` — Day of month (1–31).

(c) `received.mon` — Month of year (1–12).

(d) `received.year` — Year.

(e) `received.wday` — Day of week (Sunday = 0).

(f) `received.yday` — Day of year (0–365).

7.2.7. 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 date. Count specifies how many data items are in the 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;
```

7.3. EMS Event Types

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

7.3.1. 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_type_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_type_schema_t;
```

7.3.2. Event type list

The event type list contains a list of event type schemas:

```
typedef [ptr] ems_event_schema_t *    ems_schema_ptr_t;

typedef struct ems_event_type_list_s_t {
    long                   size;       /* number of event type
                                          schemas */
    [size_is(size)]
        ems_schema_ptr_t    schema[];   /* ptr to event type
                                          schemas */
} ems_event_type_list_t;
```

7.4. EMS Event Filters

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

7.4.1. Attribute operators

Attribute Operator	Description of Attribute Operator
ems_c_attr_op_eq	TRUE if attr_name equal to (==) attr_value
ems_c_attr_op_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 less 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 10: 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 unsigned16 ems_attr_op_t;
```

7.4.2. Event filter expression

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

```
typedef struct ems_filter_exp_s_t {
    [string] char *    attr_name;       /* attribute name */
    ems_attr_op_t      attr_operator;   /* attribute operator */
    ems_attr_value_t   attr_value;      /* attribute value */
} ems_filter_exp_t;
```

7.4.3. 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 {
    long                  size;           /* number of filter
                                             expressions */
    [size_is(size)]
        ems_filter_exp_t  filter_exps[];  /* filter expressions
                                             in list */
} ems_filter_exp_list_t;
```

7.4.4. 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 which will be entered in the CDS name space, and a list of filter expressions:

```
typedef struct ems_filter_s_t {
    ems_string_t            filter_name;      /* event filter
                                                 name */
    ems_event_type_t        type,             /* type of event
                                                 filter */
    ems_filter_exp_list_t   filter_exp_list;  /* list of filter
                                                 expressions */
} ems_filter_t;
```

Filters with event type of `Generic` can only have filter expressions with header attribute names in them (See Table 5: ‘‘Event Header Attributes’’).

7.4.5. Event filter name list

An event filter list contains a list of event filter names;

```
typedef [string] char *                 ems_string_t;

typedef struct ems_filtername_list_s_t {
    long                size;             /* number of event
                                             filter names */
    [size_is(size)]
        ems_string_t    filter_names[];   /* event filter names
                                             in group  */
} ems_filtername_list_t;
```

7.4.6. Event filter list

The event filter list structure contains a list of size filters:

```
typedef struct ems_filter_list_s_t {
    long                  size;         /* number of event
                                           filters */
    [size_is(size)]
        ems_filter_t *    filters[];    /* event filter
                                           list */
} ems_filter_list_t;
```

7.5. EMS Consumer Data Structures

7.5.1. Consumer

The consumer data structure defines an EMS consumer. Each consumer has a name which is entered in CDS, a hostname where the consumer is running, and a UUID unique to that consumer:

```
typedef struct ems_consumer_s_t {
    [string] char *   name;      /* DCE name of consumer */
    ems_netname_t *   hostname;  /* DCE hostname of consumer */
    uuid_t            uuid;      /* consumer's UUID */
} ems_consumer_t;
```

7.5.2. Consumer list

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

```
typedef struct ems_consumer_list_s_t {
    long                  size;          /* number of consumer
                                            entries */
    [size_is(size)]
        ems_consumer_t    consumer[];    /* consumer info */
} ems_consumer_list_t;
```

7.6. EMS Server Data Structure

7.6.1. Attribute list

The attribute list data structure defines a list of server attributes. Each attribute is a value maintained by an `emsd` server, and the attribute list can be used to query and set those values:

```
typedef struct ems_attrlist_s_t {
    long                  size;    /* number of attributes */
    [size_is(size)]
        ems_attribute_t   attr[];  /* event type attributes */
} ems_attrlist_t;
```

7.7. EMS Event Service Handle

7.7.1. Event service handle

`ems_handle_t` is a pointer to an opaque data structure which contains information used to allow users of EMS to connect to an Event Service on a specific DCE host. Once connected, the EMS handle is used on all subsequent Event Service operations:

```
typedef struct ems_handle_priv_s_t *    ems_handle_t;
```

8. USER INTERFACE CONSIDERATIONS

The design of EMS facilitates developing an Event Management user interface to consumer applications. A consumer could use the `ems_get_event_types()` call to get a list of the event type schemas for all the event types that suppliers will send to EMS, and first present the event types to register for, then the event type attributes to use to build event filters to provide finer control of what events the consumer could receive.

9. EMS APPLICATION PROGRAMMING INTERFACES

The following list of routines is a summary of the EMS API:

(a) `ems_register()`

Obtain an EMS handle for future calls to EMS routines.

(b) `ems_unregister()`

Frees the resources obtained by an `ems_register()` call.

(c) `ems_event_type_add()`

Add a new event type schema to the Event Type Database.

(d) `ems_event_type_delete()`

Delete an event type schema from the Event Type Database.

(e) `ems_event_type_get()`

Get an event type schema from the Event Type Database.

(f) `ems_event_type_get_list()`

Get a list of event type schemas from the Event Type Database.

(g) `ems_event_type_free_list()`

Free the list of event type schemas.

(h) `ems_svc_supplier_send()`

Send an event to EMS (only used by SVC).

(i) `ems_aud_supplier_send()`

Send an event to EMS (only used by Audit).

(j) `ems_supplier_send()`

Send an event to EMS.

(k) `ems_filter_add()`

Add a filter to the Event Filter Database.

(l) `ems_filter_append()`

Append filter expressions to the Event Filter Database.

(m) `ems_filter_get()`

Get the contents of an event filter.

(n) `ems_filter_free()`

Free storage for an event filter.

(o) `ems_filter_delete()`

Delete a filter from the Event Filter Database.

(p) `ems_filter_get_namelist()`

Get a list of the names of all filters in the Event Filter Database.

(q) `ems_filter_free_namelist()`

Free the storage for a filter namelist.

(r) `ems_filter_get_list()`

Get a list of all the filters in the Event Filter Database.

(s) `ems_filter_free_list()`

Free the list of filters.

(t) `ems_consumer_start()`

Called to start an event consumer.

(u) `ems_consumer_stop()`

Called to stop an event consumer.

(v) `ems_consumer_handler_register()`

Register a consumer event handler.

(w) `ems_consumer_register()`

Register a consumer with EMS.

(x) `ems_consumer_unregister()`

Unregister a consumer with EMS.

(y) `ems_add_filter_to_group()`

Add a filter name to a consumer’s event filter group.

(z) `ems_delete_filter_from_group()`

Delete a filter name from a consumer’s event filter group.

(aa) `ems_get_filter_group()`

Get the list of filter names that comprise a consumer's event filter group.

9.0.1. Register with EMS

SYNOPSIS:

```
#include <dce/ems.h>

void ems_register(
    ems_netname_t *     hostname,    /* DCE host name */
    ems_handle_t *      handle,      /* EMS handle */
    error_status_t *    status);     /* mgmt request status */
```

DESCRIPTION:

This routine registers with EMS, and obtains an EMS binding handle. This routine can be used by a management application that will be using the EMS Management API (See Section 11.3, ''EMS Remote Management Interface'') or by event suppliers that wish to add new event types (See Section 9.1, ''Event Type Interface'').

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`hostname` — The name of the DCE host machine where `emsd` 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — An EMS handle cannot be allocated.

`ems_s_unsupported_nameservice` — Hostname contains an unsupported name service.

9.0.2. Unregister with EMS

SYNOPSIS:

```
#include <dce/ems.h>

void ems_unregister(
    ems_handle_t *      handle,      /* EMS handle */
    error_status_t *    status);     /* mgmt request status */
```

DESCRIPTION:

This routine unregisters and frees up the resources used by an EMS handle. This routine should be called with a handle obtained by the `ems_register()` routine.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`handle` — An EMS handle obtained from the `ems_register()` routine.

Output:

`handle` — Sets the value of handle to NULL.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — An invalid handle was passed.

9.1. Event Type Interface

9.1.1. Add an event type

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

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.

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

PARAMETERS:

Input:

`handle` — A handle returned from a call to ems_register call.

Output:

`schema` — Is an EMS event type schema which describes the format of an event type.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_event_type_exists` — The event type already exists.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_invalid_event_type` — The event schema is not valid.

`ems_s_not_supported` — Not supported in this release.

This routine is not supported in release 1.

9.1.2. Delete an event type

SYNOPSIS:

```
#include <dce/ems.h>

void ems_event_type_delete(
    ems_handle_t          handle,       /* EMS handle */
    char *                type_name,    /* event type name */
    ems_event_type_t *    type,         /* event type ID */
    error_status_t *      status);      /* request status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*hostname*`/ems-server/event-types`, or

(d) on `/.:/hosts/`*hostname*`/ems-server/event-types/`*type_name*

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 type to delete.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_event_type_not_found` — The specified event type was not found.

`ems_s_invalid_name` — The event type name specified an name that is not valid.

`ems_s_not_supported` — Not supported in this release

The caller of this routine must have appropriate authority on `/.:/subsys/ems/events`. See Section 6.5.1, ‘‘Event Type Security Management’’.

This routine is not supported in release 1.

9.1.3. Get an event type

SYNOPSIS:

```
#include <dce/ems.h>

void ems_event_type_get(
   ems_handle_t            handle,      /* EMS handle */
   char *                  type_name,   /* event type name */
   ems_event_type_t *      type,        /* event type ID */
   ems_event_schema_t **   schema,      /* event type schema */
   error_status_t *        status);     /* request status */
```

DESCRIPTION:

This routine is used to retrieve event type schemas from the event type database.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

PARAMETERS:

Input:

`handle` — Should be the handle returned from a call to `ems_consumer_register call()`.

`type_name` — The event type name to retreive from the event type database.

`type` — The event type ID to retreive from the event type database.

Output:

`schema` — Returns the requested event type schema.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_invalid_name` — The event type name specified a name that is not valid.

`ems_s_event_type_not_found` — The requested event type was not found.

This routine is not supported in release 1.

9.1.4. Get event types list

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

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.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

PARAMETERS:

Input:

`handle` — Should be the handle returned from a call to `ems_consumer_register()` call.

Output:

`type_list` — Returns the list of available event types.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_no_type_list` — There is no event type list available.

9.1.5. Free event types list

SYNOPSIS:

```
#include <dce/ems.h>

void ems_event_type_free_list(
    ems_event_type_list_t **   type_list, /* list of event
                                             types */
    error_status_t *           status);   /* request status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

None.

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

9.2. Event Supplier Interface

9.2.1. Supplier send

SYNOPSIS:

```
void ems_svc_supplier_send(
    ems_handle_t        handle,     /* EMS handle */
    dce_svc_prolog_t    prolog,     /* svc prolog */
    error_status_t *    status,     /* send status */
    va_list             args);      /* svc arguments */

void ems_aud_supplier_send(
    ems_handle_t             handle, /* EMS handle */
    dce_aud_hdr_t            header, /* Audit record header */
    aud_log_ev_info_list_t * tailp,  /* Audit record data */
    error_status_t *         status);/* send status */

#include <dce/ems.h>

void ems_supplier_send(
    ems_handle_t            handle,  /* EMS handle */
    ems_event_t *           event,   /* event data */
    error_status_t *        status); /* send status */
```

DESCRIPTION:

This routine is called by event suppliers to send events to EMS. The two additional routines, `ems_aud_supplier_send` and `ems_svc_supplier_send`, were designed to be called by the DCE Audit and SVC subsystem respectively, and are not designed to be called by the general EMS supplier.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/event-types/`*`type_name`*

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — A handle that has not been initialized or that is invalid was used.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_no_memory` — Ems server received an error allocating memory.

`ems_s_queue_full` — The EMS Input Queue is full, event was not transmitted.

9.3. Event Filter Construction Interface

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

9.3.1. Add an event filter

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_add(
    ems_handle_t            handle,      /* EMS handle */
    ems_string_t            filter_name, /* event filter name
                                            */
    ems_event_type_t        type,        /* filter event type
                                            */
    ems_filter_exp_list_t * exp_list;    /* filter expression
                                            list */
    error_status_t *        status);     /* Filter construction
                                            status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

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 applier 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_exists` — The given filter name already exists.

`ems_s_invalid_name` — The name specified is not a valid EMS name..

`ems_s_invalid_filter` — The input parameters specifies an invalid filter.

9.3.2. Append to an event filter

SYNOPSIS:

```
#include <dce/ems.h>

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,     /* filter exprs to
                                             add */
    error_status_t *        status);      /* filter
                                             construction
                                             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.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_invalid_name` — The name specified is not a valid EMS name.

`ems_s_filter_not_found` — Given filter name not found in filter database.

`ems_s_invalid_filter` — The input parameters specifies an invalid filter.

9.3.3. Get an event filter

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_get(
    ems_handle_t             handle,       /* EMS handle */
    ems_string_t             filter_name,  /* event filter name
                                              */
    ems_event_type_t *       type,         /* filter event type
                                              */
    ems_filter_exp_list_t ** exp_list,     /* returned filter
                                              exp list */
    error_status_t *         status);      /* Filter
                                              construction
                                              status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

`(r)` on `/.:/hosts/`*`hostname`*`/ems-server/filters`

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` — Specifies the event type of the filter.

`exp_list` — The list of filter expressions which are part of event filter `filter_name`. `filter_name` can be freed by using the `ems_filter_free()` routine.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_invalid_name` — The name specified is not a valid EMS name.

`ems_s_filter_not_found` — Given filter name not found in filter database.

9.3.4. Free an event filter

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_free(
    ems_filter_exp_list_t ** exp_list,     /* filter exp list to
                                              free */
    error_status_t *         status);      /* Filter
                                              construction
                                              status */
```

DESCRIPTION:

This routine is used to free an event filter expression list obtained by a call to `ems_filter_get()`.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`exp_list` — The list of filter expressions to free. This list should have been obtained by a call to `ems_fitler_get()`.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

9.3.5. Delete an event filter

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_delete(
    ems_handle_t        handle,       /* EMS handle */
    ems_string_t        filter_name, /* event filter name */
    error_status_t *    status);      /* filter construction
                                         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.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*, or (d) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_invalid_name` — The name specified is not a valid EMS name.

`ems_s_filter_not_found` — Given filter name not found in filter database

`ems_s_filter_in_use` — `filter_name` appears in a consumer's event filter group

9.3.6. List event filter names

SYNOPSIS:

```
#include <dce/ems.h>

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
                                                  construction
                                                  status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(`r`) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

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. Free `name_list` using the `ems_filter_free_namelist()` routine.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_empty_filter_db` — No filters exist in the database.

9.3.7. Free an event filter name list

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_free_namelist(
    ems_filtername_list_t **  name_list,    /* event filter name
                                               list */
    error_status_t *          status);      /* free status */
```

DESCRIPTION:

This routine is used to free a filter namelist returned by various routines. The routines that return a filter namelist are `ems_filter_get_namelist()`, `ems_get_filter_group()`, and `ems_mgmt_get_filter_group()`.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`name_list` — the filter namelist to free. `name_list` can be obtained by any of the three routines mentioned above.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

9.3.8. Get event filter list

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_get_list(
    ems_handle_t          handle,        /* EMS handle */
    ems_filter_list_t **  filter_list,   /* list of filters */
    error_status_t *      status);       /* filter construction
                                            status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_invalid_handle — Handle parameter is not valid.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

ems_s_empty_filter_db — No filters exist in the database.

9.3.9. Free event filter list

SYNOPSIS:

```
#include <dce/ems.h>

void ems_filter_free_list(
    ems_filter_list_t **  filter_list,  /* event filter list */
    error_status_t *      status);      /* request 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_db_t) structure.

PERMISSION REQUIRED:

None.

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.

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

9.4. Event Consumer Interface

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

All event consumers have to make calls to the EMS event consumer setup routines before receiving EMS events. These routines set up the consumers with the DCE RPC mechanism, and set up the consumers event handler routines.

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

EMS supports the ability to queue events in a buffer per event consumer to handle cases of event overload from multiple event suppliers and conditions where the event consumer is not responding to transmit requests. EMS ensures that no event is sent to the same consumer more than once.

If an event consumer's event queue becomes too large, implying some significant problem at the event consumer, EMS will unregister the consumer, and flush the queue. EMS stores information on consumers and suppliers registered, as well as filters defined in backing store. This data is used to restart EMS.

9.4.1. Consumer start

SYNOPSIS:

```
#include <dce/ems.h>

void ems_consumer_start(
    char *                consumer,  /* consumer name */
    unsigned32             flags,    /* consumer start flags */
    error_status_t *      status);   /* start status */
```

DESCRIPTION:

This routine should be called at the beginning of each event consumer before making any register calls. It will create an object uuid to uniquely identify this event consumer and register its endpoint so that EMS can send this consumer event data. This routine does not make any RPC calls to EMS.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`consumer` — specifies the consumer name. This name must be unique, and will be registered in the CDS namespace under `/.:/hosts/`*`hostname`*`/ems/consumers`. The name will be used by the administrative interface to refer to this consumer.

`flags` — reserved for future use.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory

`ems_s_consumer_already_started` — a call has already been made to this routine.

`ems_s_invalid_name` — The name specified is not a valid EMS name.

9.4.2. Consumer stop

SYNOPSIS:

```
#include <dce/ems.h>

void ems_consumer_stop(
    error_status_t        *status);   /* stop status */
```

DESCRIPTION:

This routine should be called at the end of each event consumer. It will unregister the endpoint of this event consumer and kill the thread that was created by the consumer's event handler interface to receive all events from EMS. This routine does not make any RPC calls to EMS.

PERMISSION REQUIRED:

None.

PARAMETERS:

Output

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_consumer_not_started` — The routine `ems_consumer_start()` has not been called

9.4.3. Consumer event handler register

SYNOPSIS:

```
#include <dce/ems.h>

void ems_consumer_handler_register(
    ems_handler_t       hfunc,     /* event handler function */
    error_status_t *    status);   /* stop status */
```

DESCRIPTION:

This routine declares the event consumer's event handler. The event consumer developer is responsible for providing the handler to process events. This routine does not make any RPC calls to EMS.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

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

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

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

`ems_s_mutex_init` — Error initializing event queue.

`ems_s_cond_variable_init` — Error initializing event queue.

`ems_s_pthread_create` — Error initializing event queue.

`ems_s_consumer_not_started` — The routine `ems_consumer_start()` has not been called.

9.4.4. Consumer register

SYNOPSIS:

```
#include <dce/ems.h>

void ems_consumer_register(
    ems_netname_t *          hostname,      /* emsd hostname */
    ems_filtername_list_t *  filter_group,  /* event filter
                                               group */
    ems_handle_t *           handle,        /* EMS handle */
    error_status_t *         status);       /* register
                                               status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`hostname` — The name of the DCE host machine where `emsd` is running. If the `hostname` is NULL, then the local host is assumed.

`filter_group` — A list of event filter names which will define this consumers initial event filter group. If `filter_group` is empty, no filter group is specified, and EMS will not forward any events to this consumer until the consumer makes a call to `ems_add_event_to_group()`.

Output:

`handle` — Returns an EMS handle which can be used on subsequent calls to EMS routines.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_no_memory` — Error allocating memory.

`ems_s_already_registered` — Consumer with this name already registered.

`ems_s_mutex_init` — Error initializing event queue.

`ems_s_cond_variable_init` — Error initializing event queue.

`ems_s_pthread_create` — Error initializing event queue.

`ems_s_consumer_not_started` — The routine `ems_consumer_start()` has not been called.

`ems_s_invalid_name` — The name specified is not a valid EMS name.

9.4.5. Consumer unregister

SYNOPSIS:

```
#include <dce/ems.h>

void ems_consumer_unregister(
    ems_handle_t *          handle,     /* EMS binding handle */
    error_status_t *        status);    /* unregister status */
```

DESCRIPTION:

This routine is used by EMS event consumers to unregister with EMS. This routine should be called once for each call to `ems_consumer_register()`. The event consumer should call this routine before calling the `ems_consumer_stop()` routine.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`.

PARAMETERS:

Input:

`handle` — a handle returned from a call to `ems_consumer_register()` call. 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_unknown_consumer` — Tried to unregister a consumer that was not registered.

9.4.6. Add event filter to group

SYNOPSIS:

```
#include <dce/ems.h>

void ems_add_filter_to_group(
    ems_handle_t              handle,          /* EMS handle */
    ems_filtername_list_t *   event_filters,   /* event filter
                                                  names to add */
    error_status_t *          status);         /* filter request
                                                  status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_consumer_register()`.

`event_filters` — Contains a list of one or more event filter names to add to this consumer's 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_list_empty` — `event_filters` parameter contains no event filter names.

`ems_s_no_memory` — Error allocating memory.

`ems_s_filtername_exists` — An event filter in the filter list already exists in the consumer's event filter group.

`ems_s_invalid_name` — A name specified is not a valid EMS name.

9.4.7. Delete event filter from group

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_list_empty` — `event_filters` parameter contains no event filter names.

`ems_s_filtername_not_there` — Specified filter name to delete not in consumer's filter group.

`ems_s_invalid_name` — A name specified is not a valid EMS name.

`ems_s_consumer_not_found` — The specified consumer is not registered.

9.4.8. Get filter group

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_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` by using the `ems_filter_free_namelist()` routine.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — The specified consumer is not registered.

`ems_s_no_filter_group_defined` — Consumer does not have a filter group defined.

10. REMOTE INTERFACES

(a) `revent_type_add()`

Add an event type to the Event Type Database.

(b) `revent_type_delete()`

Delete an event type from the Event Type Database.

(c) `revent_type_get_list()`

Get the list of event types from the Event Type Database.

(d) `rsupplier_transmit()`

Send an event from a supplier to EMS.

(e) `rconsumer_register()`

Register a consumer with EMS.

(f) `rconsumer_unregister()`

Unregister a consumer with EMS.

(g) `rconsumer_reregister()`

Reregister a consumer with EMS after new `emsd` start-up.

(h) `rconsumer_add_filter_to_group()`

Add a filter to a consumer filter group.

(i) `rconsumer_del_filter_from_group()`

Delete a filter from a consumers filter group.

(j) `rconsumer_get_filter_group()`

Get the list of filters in a consumers filter group.

(k) `ems_transmit()`

Transmit an event from `emsd` to a consumer.

(l) `ems_reregister()`

Tell a consumer to reregister with EMS after new `emsd` start-up.

(m) `rfilter_add()`

Add a filter to the Event Filter Database.

(n) `rfilter_delete()`

Delete a filter from the Event Filter Database.

(o) `rfilter_get()`

Get the contents of a filter from the Event Filter Database.

(p) `rfilter_append()`

Append filter expressions to a filter in the Event Filter Database.

(q) `rfilter_get_namelist()`

Get a list of all the filter names in the Event Filter Database.

(r) `rfilter_get_list()`

Get a list of all the filters in the Event Filter Database.

10.1. EMS Event Type Interface

10.1.1. Remote event type add

SYNOPSIS:

```
void revent_type_add(
    [in]        handle_t              handle,
    [in]        ems_event_schema_t *  event_type,
    [out]       error_status_t *      status);
```

DESCRIPTION:

Add an event type schema to the EMS event type Database.

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`event_type` — Event type schema.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_not_supported` — Not supported in this release.

10.1.2. Remote event type delete

SYNOPSIS:

```
void revent_type_delete(
    [in]        handle_t              handle,
    [in]        char *                type_name,
    [out]       error_status_t *      status);
```

DESCRIPTION:

Delete an event type schema from the EMS Event Type Database.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*`hostname`*`/ems-server/event-types` or (d) on `/.:/hosts/`*`hostname`*`/ems-server/event-types/`*`type_name`*

PARAMETERS:

Input:

`handle` — EMS binding handle.

`type_name` — name of an event type schema.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_not_supported` — Not supported in this release.

10.1.3. Remote event type get event types list

SYNOPSIS:

```
void revent_type_get_list(
    [in]        handle_t                  handle,
    [out]       ems_event_type_list_t **  type_list,
    [out]       error_status_t *          status);
```

DESCRIPTION:

Delete an event type schema to the EMS Event Type Database.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

PARAMETERS:

Input:

`handle` — EMS binding handle.

Output:

`type_list` — 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_type_list` — There is no event type list available.

10.2. EMS Supplier Interface

10.2.1. Remote supplier transmit

SYNOPSIS:

```
void rsupplier_transmit(
    [in]        handle_t              handle,
    [in]        ems_event_t *         event,
    [out]       error_status_t *      status);
```

DESCRIPTION:

Transmit an event to EMS.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/event-types/`*`type_name`*

PARAMETERS:

Input:

`handle` — EMS binding handle.

`event` — Transmitted event.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — EMS server received an error allocating memory.

`ems_s_queue_full` — The EMS Input Queue is full, event was not queued.

10.3. EMS Consumer Interface

A consumer context handle has the form:

```
typedef [context_handle] void * cons_context_t;
```

The consumer context handle is used to maintain context for registered consumers. If EMS loses contact with a consumer, the context rundown routine will unregister the consumer.

10.3.1. Remote consumer register

SYNOPSIS:

```
void rconsumer_register(
    [in]          handle_t                 handle,
    [in]          uuid_t                   uuid,
    [in,string]   char *                   consumer,
    [in]          ems_filtername_list_t *  filter_group,
    [out]         cons_context_t *         ch,
    [out]         error_status_t *         status);
```

DESCRIPTION:

Register consumer with EMS.

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*hostname*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`uuid` — Consumer unique identifier.

`consumer` — Consumer name.

`filter_group` — Initial consumer event filter group.

Output:

`ch` — consumer context handle, to use in future consumer operations.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` The caller does not have permission to perform this operation.

`ems_s_no_memory` — Error allocating memory.

`ems_s_already_registered` — Consumer with this name already registered.

`ems_s_mutex_init` — Error initializing event queue.

`ems_s_cond_variable_init` — Error initializing event queue.

`ems_s_pthread_create` — Error initializing event queue.

10.3.2. Remote consumer unregister

SYNOPSIS:

```
void rconsumer_unregister(
    [in]        handle_t              handle,
    [in,out]    cons_context_t *      ch,
    [in]        uuid_t                uuid,
    [out]       error_status_t *      status);
```

DESCRIPTION:

Unregister the specified consumer with EMS. The consumer is identified by its UUID.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*hostname*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`ch` — Consumer context handle returned by rconsumer_register.

`uuid` — Consumer unique identifier.

Output:

`ch` — consumer context handle returned by `rconsumer_register()`.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_unknown_consumer` — Tried to unregister a consumer that was not registered.

10.3.3. Remote consumer reregister

SYNOPSIS:

```
void rconsumer_reregister(
    [in]            handle_t            handle,
    [in,out]        cons_context_t *    ch,
    [in]            uuid_t              uuid,
    [in,string]     char *              consumer,
    [out]           error_status_t *    status);
```

DESCRIPTION:

Reregister the specified consumer with EMS. The consumer is identified by its UUID.

PERMISSION REQUIRED:

(i) on `/.:/hosts/hostname/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`ch` — Consumer context handle returned by `rconsumer_register()`.

`uuid` — Consumer unique identifier.

`consumer` — Consumer name.

Output:

`ch` — Consumer context handle.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — Tried to unregister a consumer that was not registered.

10.3.4. Remote consumer add event filters to a group

SYNOPSIS:

```
void rconsumer_add_filter_to_group(
    [in]        handle_t                    handle,
    [in]        cons_context_t              ch,
    [in]        uuid_t                      uuid,
    [in]        ems_filtername_list_t *     filter_list,
    [out]       error_status_t *            status);
```

DESCRIPTION:

Add the event filters names in `filter_list` to a consumers event filter group. The consumer is identified by its UUID.

PERMISSION REQUIRED:

(w) on `/.:/hosts/hostname/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`ch` — Consumer context handle returned by rconsumer_register.

`uuid` — Consumer unique identifier.

`filter_list` — List of event filter names.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

ems_s_consumer_not_found — Could not find specified consumer.

ems_s_filter_list_empty — Filter list parameter contains no event filter names.

ems_s_no_memory — Error allocating memory.

ems_s_filtername_exists — An event filter in the filter list already exists in the consumer's event filter group.

10.3.5. Remote consumer delete event filters from a group

SYNOPSIS:

```
void rconsumer_del_filter_from_group(
    [in]      handle_t                   handle,
    [in]      cons_context_t             ch,
    [in]      uuid_t                     uuid,
    [in]      ems_filtername_list_t *    filter_list,
    [out]     error_status_t *           status);
```

DESCRIPTION:

Delete event filters named in filter_list from a consumer's event filter group. The consumer is identified by its UUID.

PERMISSION REQUIRED:

(w) on /.:/hosts/*hostname*/ems-server/consumers

PARAMETERS:

Input:

handle — EMS binding handle.

ch — Context handle of consumer to delete event filters from, returned by rconsumer_register().

uuid — Consumer unique identifier.

filter_list — List of event filter names.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

ems_s_consumer_not_found — Could not find specified consumer.

ems_s_filter_list_empty — Filter list parameter contains no event filter names.

ems_s_no_memory — Error allocating memory.

ems_s_filtername_not_there — An event filter in the filter list already exists in the consumer's event filter group.

10.3.6. Remote consumer get filter group

SYNOPSIS:

```
void rconsumer_get_filter_group(
    [in]        handle_t                    handle,
    [in]        cons_context_t              ch,
    [in]        uuid_t                      uuid,
    [out]       ems_filtername_list_t **    filter_list,
    [out]       error_status_t *            status);
```

DESCRIPTION:

Get the event Filter names in a Consumers event filter group. The list goes into `filter_list`. The consumer is identified by `ch`, its context handle.

PERMISSION REQUIRED:

(`r`) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`ch` — Context handle of consumer returned by rconsumer_register.

`uuid` — Consumer unique identifier.

Output:

`filter_list` — List of event filter names

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — Could not find specified consumer.

`ems_s_no_filter_group_defined` — Consumer does not have a filter group defined.

`ems_s_no_memory` — Error allocating memory.

10.4. EMS Transmit To Consumer Interface

The EMS Transmit to Consumer interface is used by consumers when they receive events from `emsd`. When a consumer calls the `ems_consumer_start()` routine, the consumer gets set up as a DCE server to receive events from `emsd`. This interface is then exported. See Section 6.6, ‘‘Writing Consumers’’ for more information about setting up consumers.

10.4.1. EMS transmit

SYNOPSIS:

```
void ems_transmit(
    [in]        handle_t                handle,
    [in]        ems_event_t *           event,
    [out]       error_status_t *        status);
```

DESCRIPTION:

Transmit an event from EMS to a consumer.

PARAMETERS:

Input:

`handle` — Consumer binding handle.

`event` — Event.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

10.4.2. EMS reregister

SYNOPSIS:

```
void ems_reregister(
    [in]        handle_t                handle,
    [out]       error_status_t *        status);
```

DESCRIPTION:

This routine is used by EMS for recovery purposes. EMS will tell a consumer to reregister if EMS has gone down, and then comes back up.

PARAMETERS:

Input:

`handle` — Consumer binding handle.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_pthread_create` — Error creating thread.

`ems_s_no_memory` — Error allocating memory.

10.5. EMS Filter Database Interface

10.5.1. Remote filter add an event filter

SYNOPSIS:

```
void rfilter_add(
    [in]        handle_t                handle,
    [in]        ems_filter_t *          filter,
    [out]       error_status_t *        status);
```

DESCRIPTION:

Add event filter filter to the EMS Event Filter Database.

PERMISSION REQUIRED:

(i) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`filter` — EMS event filter.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_exists` — The given filter name already exists.

10.5.2. Remote filter delete an event filter

SYNOPSIS:

```
void rfilter_delete(
    [in]      handle_t                handle,
    [in]      ems_string_t            filter_name,
    [out]     error_status_t *        status);
```

DESCRIPTION:

Delete the event filter named by filter_name from the Event Filter Database.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*`hostname`*`/ems-server/filters` or (d) on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*

PARAMETERS:

Input:

`handle` — EMS binding handle.

`filter_name` — Filter name to delete.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_not_found` — Given filter name not found in filter database.

`ems_s_filter_in_use` — `filter_name` appears in a consumer's event filter group.

10.5.3. Remote filter get an event filter

SYNOPSIS:

```
void rfilter_get(
    [in]        handle_t                handle,
    [in]        ems_string_t            filter_name,
    [out]       ems_filter_t **         filter,
    [out]       error_status_t *        status);
```

DESCRIPTION:

Get the event filter called filter_name from the Event Filter Database and put it into filter.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`filter_name` — Name of an event filter.

Output:

`filter` — Returned event filter.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_not_found` — Given filter name not found in filter database.

10.5.4. Remote filter update an event filter

SYNOPSIS:

```
void rfilter_append(
    [in]       handle_t                 handle,
    [in]       ems_string_t             filter_name,
    [in]       ems_filter_exp_list_t *  exp_list,
    [out]      error_status_t *         status);
```

DESCRIPTION:

Update the contents of the event filter specified by filter in the Event Filter Database. The name of the event filter is taken from the name in filter. The filter expressions in the filter are appended to the end of the existing event filter in the Event Filter Database.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filter_name`*

PARAMETERS:

Input:

`handle` — EMS binding handle.

`filter_name` — Name of filter to append to.

`exp_list` — List of filter expressions to append.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_filter_not_found` — Given filter name not found in filter database.

10.5.5. Remote filter list event filter names

SYNOPSIS:

```
void rfilter_get_namelist(
    [in]     handle_t                    handle,
    [out]    ems_filtername_list_t **    name_list,
    [out]    error_status_t *            status);
```

DESCRIPTION:

List the names of the event filters in the Event Filter Database. Put the list in list.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/filters`

PARAMETERS:

Input:

`handle` — EMS binding handle.

Output:

`name_list` — EMS event filter name list.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_empty_filter_db` — No filters exist in the database.

10.5.6. Remote filter get event filters

SYNOPSIS:

```
void rfilter_get_list(
    [in]     handle_t                handle,
    [out]    ems_filter_list_t **    filter_list,
    [out]    error_status_t *        status);
```

DESCRIPTION:

Get the list of all the event filters in the Event Filter Database. Put the list in `filter_list`.

PERMISSION REQUIRED:

(r) on `/.:/hosts/hostname/ems-server/filters`

PARAMETERS:

Input:

`handle` — EMS binding handle.

Output:

`filter_list` — EMS event filter name list.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_empty_filter_db` — No filters exist in the database.

11. EMS MANAGEMENT INTERFACE

The EMS Management interface provides a means to manage various aspects of EMS. Using this interface, applications can manage event consumers, event filters, and the EMS event log. System Administrators can also use `dcecp` to manage the same set of resources.

(a) `ems_mgmt_list_ems()`

List all hosts running `emsd`'s.

(b) `ems_mgmt_list_ems()`

Free the list all hosts running `emsd`'s.

(c) `ems_mgmt_list_attributes()`

Lists attributes for a specific `emsd`.

(d) `ems_mgmt_free_attributes()`

Free a list of EMS attributes.

(e) `ems_mgmt_set_attributes()`

Sets the attributes for a specified `emsd`.

(f) `ems_mgmt_list_consumers()`

List consumers registered with EMS.

(g) `ems_mgmt_free_consumers()`

Free a consumer list.

(h) `ems_mgmt_delete_consumer()`

Delete a consumer from the Consumer Database.

(i) `ems_mgmt_delete_filter_from_group()`

Delete a filter name from a consumer's filter group.

(j) `ems_mgmt_add_filter_to_group()`

Add a filter name to a consumer's filter group.

(k) `ems_mgmt_get_filter_group()`

Get the list of names in a consumer's filter group.

11.1. EMS Server Management Interface

11.1.1. List EMS hosts

SYNOPSIS:

```
#include <dce/ems.h>

void ems_mgmt_list_ems(
    char **           host_list, /* output list of hosts
                                    running ems */
    error_status_t *  status);   /* mgmt request status */
```

DESCRIPTION:

List hosts running `emsd`.

PERMISSION REQUIRED:

None.

PARAMETERS:

Output:

`host_list` — contains the list of hosts running `emsds`. Use free to free memory used by `host_list`.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

`ems_s_bad_arg` — Bad argument passed in.

11.1.2. List EMS emsd attributes

SYNOPSIS:

```
#include <dce/ems.h>

void ems_mgmt_list_attributes(
    ems_handle_t        handle,      /* EMS handle */
    ems_attrlist_t **   list,        /* returned attribute
                                        list */
    error_status_t *    status);     /* mgmt request status */
```

DESCRIPTION:

List `emsd` server attributes.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_register()` routine.

Output:

`list` — Contains the list of `emsd` attributes. Free this list using the `ems_mgmt_free_attributes()` routine.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_no_memory` — Error allocating memory.

11.1.3. Free a list of EMS attributes

SYNOPSIS:

```
#include <dce/ems.h>

void ems_mgmt_free_attributes(
        ems_attrlist_t **   list,       /* returned attribute
                                           list */
        error_status_t *    status);    /* mgmt request status */
```

DESCRIPTION:

Free a list of `emsd` server attributes obtained by the `ems_mgmt_list_attributes()` routine.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`list` — The list of `emsd` attributes, obained by a call to `ems_mgmt_list_attributes()`, to free.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_bad_arg` — Bad argument passed in.

11.1.4. Set EMS emsd attributes

SYNOPSIS:

```
#include <dce/ems.h>

void ems_mgmt_set_attributes(
    ems_handle_t        handle,     /* EMS handle */
    ems_attrlist_t *    list,       /* returned attribute
                                       list */
    error_status_t *    status);    /* mgmt request status */
```

DESCRIPTION:

Set attributes on an EMS server.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_register()` routine.

`list` — Contains the list of `emsd` attributes to set.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_bad_arg` — Bad argument passed in.

`ems_s_not_supported` — Not supported in this release.

Not supported in this release.

11.2. EMS Consumer Management Interface

11.2.1. List consumers

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

DESCRIPTION:

List consumers registered with EMS.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from ems_register routine.

Output:

`list` — Contains the list of consumers. This list can be freed by calling `ems_mgmt_free_consumers()`.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_bad_arg` — Bad argument passed in.

`ems_s_no_memory` — Error allocating memory.

`ems_s_no_consumers` — No consumers registered.

11.2.2. Free a list of consumers

SYNOPSIS:

```
#include <dce/ems.h>

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

DESCRIPTION:

Free a List of consumers obtained from `ems_mgmt_list_consumers()`.

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

`list` — Contains the list of consumers to free. This list was obtained from the `ems_mgmt_list_consumers()` routine.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_bad_arg` — Bad argument passed in.

11.2.3. Delete consumer

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

DESCRIPTION:

Clear all information stored in EMS about the specified consumer. This means clearing the consumers filters, then unregistering the consumer. The consumer receives notification that it is being deleted.

PERMISSION REQUIRED:

(d) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — Must contain a valid consumer handle obtained from `ems_register()` routine.

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

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

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_invalid_name` — A name specified is not a valid EMS name.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_bad_arg` — Bad argument passed in.

`ems_s_consumer_not_found` — Could not find specified consumer.

11.3. EMS Event Consumer Filter Group Management Interface

11.3.1. Management delete event filter from group

SYNOPSIS:

```
#include <dce/ems.h>

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, /* name of filters to
                                             delete */
    error_status_t *         status);     /* mgmt request
                                             status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

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 this parameter is NULL, then only one consumer can exist with the 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_invalid_name` — A name specified is not a valid EMS name.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — Could not find specified consumer.

`ems_s_filtername_not_there` — Specified filter name to delete not in consumer's filter group.

`ems_s_filter_list_empty` — Filter list parameter is empty.

11.3.2. Management add event filter to group

SYNOPSIS:

```
#include <dce/ems.h>

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, /* list of filter
                                             names to add */
    error_status_t *         status);     /* mgmt request
                                             status */
```

DESCRIPTION:

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

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

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 this parameter is NULL, then only one consumer can exist with the 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.

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_handle` — Handle parameter is not valid.

`ems_s_invalid_name` — A name specified is not a valid EMS name.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_bad_arg` — Bad argument passed in.

`ems_s_consumer_not_found` — Could not find specified consumer.

`ems_s_filter_list_empty` — Filter list parameter is empty.

`ems_s_filtername_exists` — Specified filtername already exists.

11.3.3. Management get filter group

SYNOPSIS:

```
#include <dce/ems.h>

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 */
```

DESCRIPTION:

This routine returns a list of event filter names in a consumer's event filter group.

PERMISSION REQUIRED:

(r) on /.:/hosts/*hostname*/ems-server/consumers

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 the name consumer.

Output:

filter_group — contains the list of event filter names in the specified consumers event filter group. Free filter_group using the ems_filter_free_namelist() routine.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_invalid_handle — Handle parameter is not valid.

ems_s_invalid_name — A name specified is not a valid EMS name.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

ems_s_bad_arg — Bad argument passed in.

ems_s_consumer_not_found — Could not find specified consumer.

ems_s_no_filter_group_defined — Conaumser does not have a filter group defined.

11.4. EMS Remote Management Interface

(a) rmgmt_list_ems_attributes()

List server attributes.

(b) rmgmt_set_ems_attributes()

Set server attributes.

(c) `rmgmt_list_consumers()`

List consumer registered with EMS.

(d) `rmgmt_delete_consumer()`

Delete a consumer registered with EMS.

(e) `rmgmt_delete_filter_from_group()`

Delete a filter name from a consumers filter group.

(f) `rmgmt_add_filter_to_group()`

Add a filter name to a consumers filter group.

(g) `rmgmt_get_filter_group()`

Get the list of names in a consumers filter group.

11.4.1. Remote management list server attributes

SYNOPSIS:

```
void rmgmt_list_ems_attributes(
    [in]      handle_t               handle,
    [out]     ems_attrlist_t **      list,
    [out]     error_status_t *       status);
```

DESCRIPTION:

Return a list of all the server attributes in list.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server`

PARAMETERS:

Input:

`handle` — EMS binding handle.

Output:

`list` — List of server attributes.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

11.4.2. Remote management set server attributes

SYNOPSIS:

```
void rmgmt_set_ems_attributes(
    [in]      handle_t               handle,
    [in]      ems_attrlist_t *       list,
    [out]     error_status_t *       status);
```

DESCRIPTION:

Sets the server attributes in list.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`list` — List of server attributes to set.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

Not supported in Release 1.

11.4.3. Remote management list consumers

SYNOPSIS:

```
void rmgmt_list_consumers(
    [in]      handle_t                handle,
    [out]     ems_consumer_list_t **  consumer_list,
    [out]     error_status_t *        status);
```

DESCRIPTION:

Return a list of all the registered consumers in `consumer_list`.

PERMISSION REQUIRED:

(r) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

Output:

`consumer_list` — List of registered consumers.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_memory` — Error allocating memory.

`ems_s_no_consumers` — No consumers registered.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

11.4.4. Remote management clear a consumer

SYNOPSIS:

```
void rmgmt_delete_consumer(
    [in]        handle_t                handle,
    [in,string] char *                  consumer_name,
    [in]        uuid_t                  consumer_uuid,
    [out]       error_status_t *        status);
```

DESCRIPTION:

Clear all information stored in EMS about the consumer specified by consumer_uuid and consumer_name. If consumer_uuid is specified, then it is used to uniquely identify the consumer. This means clearing the consumers filters, then unregistering the consumer.

PERMISSION REQUIRED:

(d) on /.:/hosts/*hostname*/ems-server/consumers

PARAMETERS:

Input:

handle — EMS binding handle.

consumer_name — Consumer name.

consumer_uuid — Consumer unique ID.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_insufficient_permission — The caller does not have permission to perform this operation.

ems_s_consumer_not_found — The specified consumer is not registered.

11.4.5. Remote management delete filter from filter group

SYNOPSIS:

```
void rmgmt_delete_filter_from_group(
    [in]        handle_t                  handle,
    [in,string] char *                    consumer_name,
    [in]        uuid_t                    consumer_uuid,
    [in]        ems_filtername_list_t *   filter_list,
    [out]       error_status_t *          status);
```

DESCRIPTION:

Delete event filters named in filter_list from a consumers event filter group. If the consumer_uuid is specified, then it is used to identify the consumer, otherwise the consumer_name is used.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` EMS binding handle.

`consumer_name` Consumer name.

`consumer_uuid` Consumer unique ID.

`filter_list` List of event filter names.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — The specified consumer is not registered.

`ems_s_filter_list_empty` — `event_filters` parameter contains no event filter names.

`ems_s_filtername_not_there` — Specified filter name to delete not in consumer's filter group.

`ems_s_no_memory` — EMS server received an error allocating memory.

11.4.6. Remote management add filter from to group

SYNOPSIS:

```
void rmgmt_add_filter_to_group(
    [in]         handle_t                 handle,
    [in,string]  char *                   consumer_name,
    [in]         uuid_t                   consumer_uuid,
    [in]         ems_filtername_list_t *  filter_list,
    [out]        error_status_t *         status);
```

DESCRIPTION:

Adds event filters named in `filter_list` to a consumers event filter group. If the `consumer_uuid` is specified, then it is used to identify the consumer, otherwise the `consumer_name` is used.

PERMISSION REQUIRED:

(w) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`consumer_name` — Consumer name.

`consumer_uuid` — Consumer unique ID.

`filter_list` — List of event filter names.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — The specified consumer is not registered.

`ems_s_filter_list_empty` — `event_filters` parameter contains no event filter names.

`ems_s_filtername_exists` — Specified filtername already exists.

`ems_s_no_memory` — EMS server received an error allocating memory.

11.4.7. Remote management get a filter group

SYNOPSIS:

```
void rmgmt_get_filter_group(
    [in]          handle_t                    handle,
    [in,string]   char *                      consumer_name,
    [in]          uuid_t                      consumer_uuid,
    [out]         ems_filtername_list_t **    filter_list,
    [out]         error_status_t *            status);
```

DESCRIPTION:

Returns the list of event filter names in a consumer's event filter group in `filter_list`. If the `consumer_uuid` is specified, then it is used to identify the consumer, otherwise the `consumer_name` is used.

PERMISSION REQUIRED:

(`r`) on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

PARAMETERS:

Input:

`handle` — EMS binding handle.

`consumer_name` — Consumer name.

`consumer_uuid` — Consumer unique ID.

Output:

`filter_list` — List of event filter names

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_insufficient_permission` — The caller does not have permission to perform this operation.

`ems_s_consumer_not_found` — The specified consumer is not registered.

`ems_s_no_filter_group_defined` — Conaumser does not have a filter group defined.

11.5. EMS Log Information Management Interface

EMS offers an interface in the EMS event log. This interface allows management applications to manipulate event logs. The log interface is a local interface only, and can only be executed on the machine running the `emsd` server.

(a) `ems_log_open()`

Open an EMS event log.

(b) `ems_log_read()`

Read events from an EMS event log.

(c) `ems_log_close()`

Close an EMS event log.

(d) `ems_log_rewind()`

Rewind an EMS event log.

(e) `ems_event_free()`

Free storage used by an EMS event.

11.5.1. Open event log

SYNOPSIS:

```
#include <dce/ems.h>

void ems_log_open(
    ems_log_file_t *     log_file,     /* log file handle */
    char *               log_dir,      /* directory where log
                                          file located */
    error_status_t *     status);      /* log function status */
```

DESCRIPTION:

Open an EMS event log. This routine locks the event log database until the `ems_log_close()` call is called.

PERMISSION REQUIRED:

On AIX, have to be root.

PARAMETERS:

Input:

`log_dir` Directory where log directory located. If NULL, then the environment variable `EMS_EVENTLOG_DIR` is checked, if that is not set, then the default directory is used.

Output:

`log_file` — Log file handle to use in other `ems_log_*()` routines.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_no_event_log` — Event log not found.

`ems_s_no_log_entries` — No event log entries.

`ems_s_no_memory` — Error encountered allocating memory.

11.5.2. Read event log

SYNOPSIS:

```
#include <dce/ems.h>

void ems_log_read(
    ems_log_file_t     handle,     /* log file handle to read */
    ems_event_t **     event,      /* return event value */
    error_status_t *   status);    /* log function status */
```

DESCRIPTION:

Read an event from the EMS event log.

PERMISSION REQUIRED:

On AIX, have to be root.

PARAMETERS:

Input:

`handle` — specifies the open event log to read from.

Output:

`event` — returns the next event from the event log. Call `ems_event_free()` to free storage used by event.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

`ems_s_invalid_log_handle` — Invalid log file handle passed in.

`ems_s_no_more_events` — No more events to read in log file.

`ems_s_no_memory` — Error encountered allocating memory.

11.5.3. Close event log

SYNOPSIS:

```
#include <dce/ems.h>

void ems_log_close(
    ems_log_file_t *  handle,     /* log file handle to close */
    error_status_t *  status);    /* log function status */
```

DESCRIPTION:

Close an event log file.

PERMISSION REQUIRED:

On AIX, have to be root.

PARAMETERS:

Input:

`handle` — specifies the event log file to close.

Output:

`handle` — Memory freed, and set to NULL.

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_invalid_log_handle — Invalid log file handle passed in.

11.5.4. Rewind event log

SYNOPSIS:

```
#include <dce/ems.h>

void ems_log_rewind(
    ems_log_file_t    handle,   /* log file handle to rewind */
    error_status_t *  status);  /* log function status */
```

DESCRIPTION:

Rewind an event log. This allows the event log to be rewound to the beginning, to be reread. This function is equivalent to calling ems_log_close(), then ems_log_open() again.

PERMISSION REQUIRED:

On AIX, have to be root.

PARAMETERS:

Input:

handle — Specifies the event log file to rewind.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

error_status_ok — Success.

ems_s_invalid_log_handle — Invalid log file handle passed in.

ems_s_no_log_entries — No event log entries.

11.5.5. Free an EMS event

SYNOPSIS:

```
#include <dce/ems.h>

void ems_event_free(
    ems_event_t **      event,     /* event to free */
    error_status_t *    status);   /* log function status */
```

DESCRIPTION:

Free an EMS event obtained from ems_log_read().

PERMISSION REQUIRED:

None.

PARAMETERS:

Input:

event — event obtained from ems_log_read() to free.

Output:

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

POSSIBLE STATUS CODES AND THEIR MEANING:

`error_status_ok` — Success.

11.6. EMS Management Command Line (dcecp) Interface

The EMS `dcecp` interface allows system administrators to access and modify various aspects of EMS. Here is a summary of the EMS `dcecp` objects:

(a) `ems` — A `dcecp` object that manages the EMS daemon on a DCE host.

(b) `emsconsumer` — A `dcecp` object that manages EMS consumers and their event filter groups.

(c) `emsevent` — A `dcecp` object that displays EMS Event types and event type schemas.

(d) `emsfilter` — A `dcecp` object that manages EMS event filters on a DCE host.

(e) `emslog` — A `dcecp` object that manages the EMS log file on the current host.

11.6.1. ems commands

PURPOSE:

Manage the EMS daemon on a DCE host.

FORMAT:

```
ems catalog
ems help [operation | -verbose]
ems operations
ems show [-host hostname]
```

ARGUMENT:

`operation` — The name of one specific `ems` operation (subcommand) about which you want to see help information.

ATTRIBUTES:

`eventlog_dir` — Specifies the directory name used where the EMS daemon puts the event log.

`queue_size` — Specifies the queue size for the event queues.

USAGE:

The `ems` object represents the EMS daemon (called `emsd`) on a host.

This command operates on the EMS daemon on the local host, unless the `-host` option is specified. The format of the host name accepted is either an entire DCE name (e.g., `/.:/hosts/jurassic.austin.ibm.com`) or a hostname with domain name (e.g., `jurassic.austin.ibm.com`).

RELATED INFORMATION:

Commands: `dcecp`, `ems catalog`, `ems help`, `ems operations`, `ems show`, `emsconsumer` commands, `emsevent` commands

11.6.2. ems catalog

PURPOSE:

Returns the list of all hosts that the EMS daemon is running on in the current cell.

FORMAT:

```
ems catalog
```

USAGE:

The `ems catalog` command returns the list of all hosts that the EMS daemon is running on in the current cell.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> ems catalog
/.:/hosts/eagle.austin.ibm.com
/.:/hosts/umesh.austin.ibm.com
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `ems help`, `ems operations`, `ems show`, `emsconsumer` commands, `emsevent` commands

11.6.3. ems help

PURPOSE:

Returns help information on the object.

FORMAT:

```
ems help [operation | -verbose]
```

ARGUMENT:

`operation` — The name of one specific EMS operation (subcommand) about which you want to see help information.

OPTION

`-verbose` — Displays detailed information about the ems commands object.

USAGE:

The `ems help` command returns help information on the object. The help operation takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> ems help
catalog    Returns a list of all hosts that the EMS daemon \
           is running on.
help       Prints a summary of command-line options.
operations Returns the valid operations for command.
show       Returns the attributes for the EMS daemon.
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `ems catalog`, `ems operations`, `ems show`, `emsconsumer` commands, `emsevent` commands

11.6.4. ems operations

PURPOSE:

Returns a list of operations supported by the object.

FORMAT:

```
ems operations
```

USAGE:

The `ems operations` command returns a list of operations supported by the object. It takes no arguments, and always returns a TCL list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that help and operations are listed last.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> ems operations
catalog show help operations
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `ems catalog`, `ems help`, `ems show`, `emsconsumer` commands, `emsevent` commands

11.6.5. ems show

PURPOSE:

Returns the attribute list for the EMS daemon.

FORMAT:

```
ems show [-host hostname]
```

OPTIONS:

`-host` *hostname* — Specifies the host where the EMS daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

ATTRIBUTES:

`eventlog_dir` — Specifies the directory name used where the EMS daemon puts the event log.

`queue_size` — Specifies the queue size for the event queues.

USAGE:

The `ems show` command returns the attribute list for the EMS daemon.

PERMISSION REQUIRED:

None.

You must have read (r) permission on `/.:/hosts/`*hostname*`/ems-server`

EXAMPLES:

```
dcecp> ems show
{eventlog_dir /opt/dcelocal/dce/var/ems}
{queue_size 5000}
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `ems catalog`, `ems help`, `ems operations`, `emsconsumer` commands, `emsevent` commands

11.6.6. emsconsumer commands

PURPOSE:

Manage EMS consumers and their event filter groups.

FORMAT:

```
emsconsumer catalog [-host hostname]
emsconsumer delete consumer {-uuid uuid} \
            [-host hostname]
emsconsumer help [operation | -verbose]
emsconsumer modify consumer {-uuid uuid} \
            {-add|-delete} {filter filtername} \
            [-host hostname]
emsconsumer operations
emsconsumer show consumer {-uuid uuid} \
            [-host hostname]
```

ARGUMENTS:

`consumer` — A consumer name.

`operation` — The name of one specific `emsconsumer` operation (subcommand) about which you want to see help information.

USAGE:

The `emsconsumer` object represents an EMS consumer. An EMS consumer registers with EMS to receive event data. It defines event filters to identify what events should be forwarded to it. This command operates on the EMS daemon on the local host, unless the `-host` option is specified. The format of the host name accepted is either an entire DCE name (`/.:/hosts/jurassic.autin.ibm.com`) or a hostname with domain name (`jurassic.austin.ibm.com`).

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer catalog`, `emsconsumer delete`, `emsconsumer help`, `emsconsumer modify`, `emsconsumer operations`, `emsconsumer show`, `emsfilter` commands

11.6.7. emsconsumer catalog

PURPOSE:

Returns the list of registered consumers with EMS on a host.

FORMAT:

```
emsconsumer catalog [-host hostname]
```

OPTIONS:

`-host` *`hostname`* Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

USAGE:

The `emsconsumer catalog` command returns the list of registered consumers with EMS on a host. The consumer names returned are in an arbitrary order.

PERMISSION REQUIRED:

None.

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

EXAMPLES:

```
dcecp> emsconsumer catalog
consumer1 7e383761-f41f-11ce-9051-08005acd43c6 \
          /.:/hosts/eagle.austin.ibm.com
consumer1 a4c7ff26-f449-11ce-a863-10005a4f3556 \
          /.:/hosts/eagle.austin.ibm.com
consumer2 283cc40c-f447-11ce-9dd3-10005a4f3556 \
          /.:/hosts/umesh.austin.ibm.com
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer delete`, `emsconsumer help`, `emsconsumer modify`, `emsconsumer operations`, `emsconsumer show`, `emsfilter` commands

11.6.8. emsconsumer delete

PURPOSE:

Deletes a registered consumer from EMS on a host.

FORMAT:

```
emsconsumer delete consumer {-uuid uuid} \
            [-host hostname]
```

ARGUMENTS:

`consumer` — A consumer name.

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

`-uuid` *`uuid`* — Specifies the unique universal identifier (UUID) that is assigned to the consumer. host-name with a domain name.

USAGE:

The `emsconsumer delete` command deletes a registered consumer from EMS on a host. The argument is the name of the consumer to be deleted or its assigned UUID. The command returns an empty string on success.

PERMISSION REQUIRED:

You must have delete (d) permission on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

EXAMPLES:

```
dcecp> emsconsumer delete consumer2
dcecp> emsconsumer delete \
       -uuid 7e383761-f41f-11ce-9051-08005acd43c6
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer catalog`, `emsconsumer help`, `emsconsumer modify`, `emsconsumer operations`, `emsconsumer show`, `emsfilter` commands

11.6.9. emsconsumer help

PURPOSE:

Returns help information on the object.

FORMAT:

```
emsconsumer help [operation | -verbose]
```

ARGUMENTS:

`operation` — The name of one specific `emsconsumer` operation (subcommand) about which you want to see help information.

OPTIONS:

`-verbose` — Returns detailed information about the emsconsumer commands object.

USAGE:

The `emsconsumer help` command returns help information on the object. The `help` operation takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsconsumer help
catalog    Returns the list of registered consumers with EMS \
           on a host.
delete     Deletes a registered consumer from EMS on a host.
modify     Modifies the event filter group associated with a \
           consumer.
show       Shows information about a consumer including the \
           consumer's filter group.
help       Prints a summary of command-line options.
operations Returns the valid operations for command.
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer catalog`, `emsconsumer delete`, `emsconsumer modify`, `emsconsumer operations`, `emsconsumer show`, `emsfilter` commands

11.6.10. emsconsumer modify

PURPOSE:

Modifies the event filter group associated with the given consumer.

FORMAT:

```
emsconsumer modify consumer {-uuid uuid} \
            {-add|-delete} {filter filtername} \
            [-host hostname]
```

ARGUMENTS:

`consumer` — A consumer name.

OPTIONS:

`-add`|`-delete` — Adds or deletes filternames from the consumer filter group.

`filter` *`filtername`* — Specifies the name of the consumer filter group.

`-host` *`hostname`* — Specifies the host where the EMS daemon is running. The format of the hostname is either an entire DCE name or a hostname with a domain name.

`-uuid` *`uuid`* — Specifies the unique universal identifier (UUID) that is assigned to the consumer.

USAGE:

The `emsconsumer modify` command modifies the event filter group associated with the given consumer. Filters can be added or deleted from a consumer event filter group. Added filters are always placed at the end of the consumer event filter group. The command returns an empty string on success.

PERMISSION REQUIRED:

You must have write (w) permission on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

EXAMPLES:

```
dcecp> emsconsumer modify consumer2 -add {filter foo}
dcecp>
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer catalog`, `emsconsumer delete`, `emsconsumer help`, `emsconsumer operations`, `emsconsumer show`, `emsfilter` commands

11.6.11. emsconsumer operations

Returns a list of operations supported by the object.

FORMAT:

```
emsconsumer operations
```

USAGE:

The `emsconsumer operations` command returns a list of operations supported by the object. It takes no arguments, and always returns a TCL list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that help and operations are listed last.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsconsumer operations
catalog delete modify show help operations
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer catalog`, `emsconsumer delete`, `emsconsumer help`, `emsconsumer modify`, `emsconsumer show`, `emsfilter` commands

11.6.12. emsconsumer show

PURPOSE:

Returns the list of filter names in a consumer filter group.

FORMAT:

```
emsconsumer show consumer {-uuid uuid} \
            [-host hostname]
```

ARGUMENTS:

`consumer` — A consumer name.

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

`-uuid` *`uuid`* — Specifies the unique universal identifier (UUID) that is assigned to the consumer.

USAGE:

The `emsconsumer showcommand` returns the list of filter names in a consumer's filter group. This command takes the consumer name as an argument.

PERMISSION REQUIRED:

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/consumers`

EXAMPLES:

```
dcecp> emsconsumer show consumer2
{foo2 foo3 foo4 foo5}
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsconsumer catalog`, `emsconsumer delete`, `emsconsumer help`, `emsconsumer modify`, `emsconsumer operations`, `emsfilter` commands

11.6.13. emsevent commands

PURPOSE:

Display EMS event types and event type schemas.

FORMAT:

```
emsevent catalog [-host hostname]
emsevent help [operation | -verbose]
emsevent operations
emsevent show event_type [-host hostname]
```

ARGUMENTS:

`event_type` — Name of the event type.

`operation` — The name of one specific `emsevent operation` (subcommand) about which you want to see help information.

USAGE:

The `emsevent object` represents the EMS event type which is a class of events with the same format. This format of the event types are defined by event type schemas. A event type schema consists of a list of attribute name/type pairs which specify the data format of an event. This command allows for the list of available event types to be displayed and the event type schema for a particular event type. It operates on the EMS daemon on the local host, unless the `-host` option is specified. The format of the host name accepted is either an entire DCE name (`/.:/hosts/jurassic.austin.ibm.com`) or a hostname with domain name (`jurassic.austin.ibm.com`).

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent catalog`, `emsevent help`, `emsevent operations`, `emsevent show`

11.6.14. emsevent catalog

PURPOSE:

Returns the list of available event types.

FORMAT:

`emsevent catalog [-host` *`hostname`*`]`

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS Daemon is running. The format of the hostname is either an entire DCE name or a hostname with a domain name.

USAGE:

The `emsevent catalog` command displays a list of the available event types and the event type schema for a particular event type.

PERMISSION REQUIRED:

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/event-types`

EXAMPLES:

```
dcecp> emsevent catalog
{SVC}
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emsevent help`, `emsevent operations`, `emsevent show`

11.6.15. emsevent help

PURPOSE:

Returns help information on the object.

FORMAT:

`emsevent help [`*`operation`*` | -verbose]`

ARGUMENTS:

`operation` — The name of one specific `emsevent operation` (subcommand) about which you want to see help information.

OPTIONS:

`-verbose` — Returns detailed information about the `emsevent` commands object.

USAGE:

The `emsevent help` command returns help information on the object. The `help` operation takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsevent help
catalog    Returns the list of available event types.
help       Prints a summary of command-line options.
operations Returns the valid operations for command.
show       Returns the event type schema for a event type.
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emsevent catalog`, `emsevent operations`, `emsevent show`

11.6.16. emsevent operations

PURPOSE:

Returns a list of operations supported by the object.

FORMAT:

```
emsevent operations
```

USAGE:

The `emsevent` operations command returns a list of operations supported by the object. It takes no arguments, and always returns a TCL list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that help and operations are listed last.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsevent operations
catalog show help operations
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emsevent catalog`, `emsevent help`, `emsevent show`

11.6.17. emsevent show

PURPOSE:

Returns the event type schema for a event type.

FORMAT:

`emsevent show event_type [-host` *`hostname`*`]`

ARGUMENTS:

`event_type` — Name of the event type.

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS daemon is running.

USAGE:

The `emsevent show` command returns the event type schema for a event type. A list of attribute name/type pairs is displayed.

PERMISSION REQUIRED:

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/event-types/`*`event_type`*

EXAMPLES:

```
dcecp> emsevent show SVC
{version ems_c_attr_ulong_int}
{t ems_c_attr_utc}
{argtypes ems_c_attr_char_string}
{table index ems_c_attr_ulong_int}
{attributes ems_c_attr_ulong_int}
{message index ems_c_attr_ulong_int}
{format ems_c_attr_char_string}
{file ems_c_attr_char_string}
{progname ems_c_attr_char_string}
{line ems_c_attr_ulong_int}
{threadid ems_c_attr_ulong_int}
{component name ems_c_attr_char_string}
{sc_name ems_c_attr_char_string}
{attribute.debug ems_c_attr_ushort_int}
{attribute.severity ems_c_attr_ushort_int}
{attribute.actroute ems_c_attr_ulong_int}
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emsevent catalog`, `emsevent help`, `emsevent` operations

11.6.18. emsfilter commands

PURPOSE:

Manage EMS event filters on a DCE host.

FORMAT:

```
emsfilter catalog [-host hostname]
emsfilter delete filtername [-host hostname]
emsfilter help [operation | -verbose]
emsfilter operations
emsfilter show filtername [-host hostname]
```

ARGUMENTS:

`filtername` — A filter name.

`operation` — The name of one specific `emsfilter operation` (subcommand) about which you want to see help information.

USAGE:

The `emsfilter` object represents EMS event filters which are kept by the EMS daemon. The EMS event filters are applied by EMS to events received from suppliers to determine if the events will get forwarded on to the consumers. An EMS event filter is a collection of one or more filter expressions. Each filter expression consists of an attribute name, an attribute operator, and an attribute value. This command operates on the EMS daemon on the local host, unless the `-host` option is specified. The format of the host name accepted is either an entire DCE name (`/.:/hosts/jurassic.austin.ibm.com`) or a hostname with domain name (`jurassic.austin.ibm.com`).

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter catalog`, `emsfilter delete`, `emsfilter help`, `emsfilter operations`, `emsfilter show`

11.6.19. emsfilter catalog

PURPOSE:

Returns a list of names of all filters from EMS on a host.

FORMAT:

`emsfilter catalog [-host hostname]`

OPTIONS:

`-host hostname` Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

USAGE:

The `emsfilter catalog` command returns a list of names of all filters from EMS on a host. The filter names returned are in alphabetical order and not in the order received by EMS.

PERMISSION REQUIRED:

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/filters`

EXAMPLES:

In the following example, there are two filters kept by the EMS daemon:

```
dcecp> emsfilter catalog
{Filter1 Filter2}
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter` commands, `emsfilter delete`, `emsfilter help`, `emsfilter operations`, `emsfilter show`

11.6.20. emsfilter delete

PURPOSE:

Deletes a filter and its associated filter expressions.

FORMAT:

`emsfilter delete filtername [-host hostname]`

ARGUMENTS:

`filtername` — A filter name.

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

USAGE:

The `emsfilter delete` command deletes a filter and its associated filter expressions. The argument is an event filtername to be deleted. If the filter to be deleted is currently being used by at least one consumer, it cannot be deleted and an error message will be displayed. The command returns an empty string on success.

PERMISSION REQUIRED:

You must have delete (d) permission on `/.:/hosts/`*`hostname`*`/ems-server/filters/`*`filtername`*.

EXAMPLE

```
dcecp> emsfilter delete Filter1
dcecp>
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter` commands, `emsfilter catalog`, `emsfilter help`, `emsfilter operations`, `emsfilter show`

11.6.21. emsfilter help

PURPOSE:

Returns help information on the object.

FORMAT:

`emsfilter help [`*`operation`*` | -verbose]`

ARGUMENTS:

`operation` — The name of one specific `emsfilter` operation (subcommand) about which you want to see help information.

OPTIONS:

`-verbose` — Returns detailed information about the emsfilter commands object .

USAGE:

The `emsfilter help` command returns help information on the object. The help operation takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsfilter help
catalog     Returns a list of names of all filters from EMS on \
            a host.
delete      Deletes a filter and its associated filter \
            expressions.
help        Prints a summary of command-line options.
operations  Returns the valid operations for command.
show        Returns a list of filter expressions in a \
            specified filter.
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter` commands, `emsfilter catalog`, `emsfilter delete`, `emsfilter operations`, `emsfilter show`

11.6.22. emsfilter operations

PURPOSE:

Returns a list of operations supported by the object.

FORMAT:

`emsfilter operations`

USAGE:

The `emsfilter` operations command returns a list of operations supported by the object. It takes no arguments, and always returns a TCL list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that help and operations are listed last.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emsfilter operations
catalog delete show help operations
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter` commands, `emsfilter catalog`, `emsfilter delete`, `emsfilter help`, `emsfilter show`

11.6.23. emsfilter show

PURPOSE:

Returns a list of filter expressions in a specified filter.

FORMAT:

`emsfilter show filtername [-host` *`hostname`*`]`

ARGUMENTS:

`filtername` — A filter name.

OPTIONS:

`-host` *`hostname`* — Specifies the host where the EMS Daemon is running. The format of the host name is either an entire DCE name or a host name with a domain name.

USAGE:

The `emsfilter show` command returns a list of filter expressions in a specified filter. The argument is a filter name to be shown.

PERMISSION REQUIRED:

You must have read (r) permission on `/.:/hosts/`*`hostname`*`/ems-server/filters`

EXAMPLES:

```
dcecp> emsfilter show Filter2
{event_type == SVC}
{file == file.c}
```

RELATED INFORMATION:

Commands: `dcecp`, `emsconsumer` commands, `emsfilter` commands, `emsfilter catalog`, `emsfilter delete`, `emsfilter help`, `emsfilter show`

11.6.24. emslog commands

PURPOSE:

Manage EMS log files on the current host.

FORMAT:

```
emslog help [operation | -verbose
emslog operations
emslog show [-dir directory]  [-to file]
```

ARGUMENTS:

`operation` — The name of one specific `emslog` operation (subcommand) about which you want to see help information.

USAGE:

The `emslog` object represents the EMS event log which is used to store events in case of failures of the EMS daemon. The EMS daemon writes all events to the event log, and deletes the event record once the event has been transmitted to all the consumers that were supposed to get the event. The event log is kept in a file on the machine where EMS daemon is running. This command operates on the EMS daemon on the local host.

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emslog help`, `emslog operations`, `emslog show`

11.6.25. emslog help

PURPOSE:

Returns help information on the object.

FORMAT:

`emslog help [`*`operation`*` | -verbose]`

ARGUMENTS:

`operation` — The name of one specific `emslog` operation (subcommand) about which you want to see help information.

OPTIONS:

`-verbose` — Displays information about the `emslog` commands object.

USAGE:

The `emslog help` command returns help information on the object. The help operation takes an argument which may be an operation supported by the object or the `-verbose` switch to return more information.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emslog help
help       Prints a summary of command-line options.
operations Returns the valid operations for command.
show       Returns a list of events in the event log file.
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emslog` commands, `emslog operations`, `emslog show`

11.6.26. emslog operations

PURPOSE:

Returns a list of operations supported by the object.

FORMAT:

```
emslog operations
```

USAGE:

The `emslog` operations command returns a list of operations supported by the object. It takes no arguments, and always returns a TCL list suitable for use in a `foreach` statement. The order of the elements is alphabetical with the exception that help and operations are listed last.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emslog operations
show help operations
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emslog` commands, `emslog help`, `emslog show`

11.6.27. emslog show

PURPOSE:

Returns a list of events in the event log file.

FORMAT:

`emslog show [-dir` *directory*`] [-to` *file*`]`

OPTIONS:

`-dir` *directory* — Specifies the directory where the log file is stored.

`-to` *file* — Specifies the filename into which the output is captured.

USAGE:

The `emslog show` command returns a list of events in the event log file.

PERMISSION REQUIRED:

None.

EXAMPLES:

```
dcecp> emslog show
--- Start of an EMS event record ---
Type: SVC:Event Id: 8d1b0b00-e9e7-11ce-8af3-10005a890435
Name Service: DCE /.../eagle_dce/hosts.hidalgod.austin.ibm.com
Description Name: EMS_Test_Producer
PID: 565 UID: 0 GID: 0
Severity: NOTICE
Arrival Time: 1995-09-08-14:06:32.970+00:00I-----
Printing 16 items
Item 1: [version] = ulong init 1
Item 2: [t] = 1995-09-08-14:06:32.970+00:00I-----
Item 3: [argtypes] = char string
Item 4: [table_index] = ulong int 0
Item 5: [attributes] = ulong int 64
Item 6: [message_index] = ulong int 389738500
Item 7: [format] = char string Test Supplier starting
Item 8: [file] = char string supplier.c
Item 9: [progname] char string EMS_Test_Producer
Item 10: [line] = ulong int 63
```

```
Item 11: [threadid] = ulong int 2
Item 12: [component_name] = char string sup
Item 13: [sc_name] = char string general
Item 14: [attribute.debug] = ushort int 0
Item 15: [attribute.severity] = ushort int 4
Item 16: [attribute.actroute] = ulong int 0
--- End of an EMS event record ---
```

RELATED INFORMATION:

Commands: `dcecp`, `ems` commands, `emsevent` commands, `emslog` commands, `emslog help`, `emslog show`

11.7. EMS Configuration

The following steps are required to configure EMS on a machine:

(a) Create directory `$DCELOCAL/var/ems` for the following databases:
  (i) Consumer database.
  (ii) Filter database.
  (iii) Event Type database.

(b) Create log file directory `$EMS_EVENTLOG_DIR`.

(c) Create junction in CDS:

```
/.:/hosts/host name/ems-server
```

(d) Create the groups `ems-admin`, `ems-consumer`, and `ems-supplier`, and give them the appropriate authority:

```
/.:/hosts/hostname/ems-server/events
    ems-admin       wrc
    ems-consumer    r
    ems-supplier    w

/.:/hosts/hostname/ems-server/filters
    ems-admin       dimrc
    any_other       r
```

(e) Put `emsd` start-up command in `rc.dce`.

11.7.1. EMS Environment Variables

Variable Name	Default Value	Description
`EMS_EVENTLOG_DIR`	`$DCELOCAL/var/ems`	Directory for storing EMS event log files
`EMS_QUEUE_SIZE`	512	Size of EMS event queue

Table 11: EMS Environment Variables

11.8. EMS Binding Information

EMS puts its binding information in `/.:/hosts/hostname/ems-server`.

11.9. EMS Start-up command

`emsd`

PURPOSE:

Starts the DCE Event Management Services Daemon.

FORMAT:

```
emsd [-d debug_level] [-l log_directory] [-q queue_size] \
     [-w svc_route ... -w svc_route]
```

OPTIONS:

`-d` *`debug_level`* — Specifies DCE debug routing instructions. For a complete discussion of the ways you can specify routings for debug messages, see the svcroute reference page.

`-l` *`log_directory`* — Specifies where the log file will reside.

`-q` *`queue_size`* — Specifies the maximum number of events that will be queued by EMS. The default size is 512. This value can also be set by setting the `EMS_QUEUE_SIZE` environment variable. Specifying the `-q` option overrides the environment variable setting.

`-w` *`svc_route`* — Specifies DCE serviceability routing instructions. Fo a complete discussion of the ways in which you can specify routings for serviceability messages, see the `svcroute` reference page in the Application Development Guide — Core Components.

USAGE:

The `emsd` command starts the Event Management Service (EMS) daemon. The EMS daemon must be run on the host before EMS suppliers and consumers can start receiving events. An EMS daemon must be running in the DCE cell before a consumer can receive events or a supplier can supply events. The EMS daemon runs under the local host machine principal identity (`host/`*`hostname`*`/self`). A DCE Host daemon (`dced`) must be running on the local host when `emsd` is started. The `emsd` command also requires a `cdsadvertiser`.

PERMISSION REQUIRED:

None.

EXAMPLE

```
emsd -q 2048 -l /opt/dcelocal/var/ems
emsd -w NOTICE:STDOUT:- -w NOTICE-VERBOSE:STDOUT-:- \
     -d ems:log.5
```

12. RESTRICTIONS AND LIMITATIONS

One goal of EMS is to guarantee event persistence over system or network failures. This goal places a limitation on the size of the event queue dependent on the system resources available to store events. Storing events may require writing data to the file system, which would imply that enough space is available to write the required amount of data.

13. SEAMLESS INTEGRATION WITH AUDIT AND SVC

13.1. Default Definition of SVC and Audit Event Types

Two default event types will be provided to support the DCE Audit and SVC subsystems. This means that when a call is made to `ems_get_event_types()`, Audit and SVC will be the default types if the event type database is not there.

13.2. Integration With the SVC Subsystem

In order to integrate EMS with SVC, a new routing destination will be added to the serviceability routing specification called EMS. When this destination is specified, messages will also go to EMS.

Here are routing specifications from a sample routing file which sends fatal and error messages to EMS, as well as to standard error:

```
ERROR:STDERR:;EMS
FATAL:STDERR:;EMS
```

13.3. Integration With the Audit Subsystem

In order to integrate EMS with the Audit subsystem, a new audit action will be added to both the audit filter guide specification, as well as to the options parameter to `dce_aud_commit()` and the `dce_aud_start*()` routines.

(a) The value of the audit action will be:

```
aud_c_esl_act_ems
```

(b) The value that will appear in the filter guides will be:

```
ems
```

(c) The value that will be used in the options parameter will be:

```
aud_c_evt_always_ems
```

14. OTHER EVENT SERVICES

Today, there are at least two different event services which could provide this function for DCE. One is DME's Event Services (EVS) (see [DME]), which is an emitter-side model providing event controllers (ECs) as a mechanism for event consumers to define filters. However, the feasibility of using EVS is very limited by its cumbersome subscription architecture, its complexity, and lack of integration with other DCE services.

Another alternative is the OMG's Event Services architecture which is part of the Joint Object Services Submission (see [OMG]). This architecture presents an object based event service that defines two basic models for communicating event data: the push model which allows event suppliers to initiate the transfer of events to consumers, and the pull model which allows event consumers to request events from event suppliers. An event channel object allows multiple suppliers to communicate with multiple consumers in an asynchronous way. The main drawback to the OMG Event Service model is that there is no currently-available implementations, and it would require DCE to standardize on one particular object model.

In the interest of time to market, cost to DCE system vendors, and pragmatics, this document proposes that an event service be implemented and integrated into the DCE 1.1 product. The implementation architecture should closely follow the OMG Event Service specification from the Joint Object Services Submission. However, its overriding goal is to assure that the event service is integrated into existing DCE services where appropriate.

14.1. DCE Event Management Service vs. OMG Event Service

The DCE Event Management Service is a partial procedural implementation of the OMG event service that is tightly integrated with the existing (as of DCE 1.1) SVC subsystem, will utilize the DCE security service for authentication of suppliers and consumers, and will utilize the DCE directory service to enable consumers to locate the EMS service on various DCE hosts.

(a) OMG event service is a CORBA based service where event data is communicated between event supplier and consumer by issuing a standard CORBA request.

(b) DCE Event Management Service is a DCE based service where event data is communicated between event supplier and consumer by issuing a standard DCE RPC request.

(c) OMG event service will support the push and pull models for communication between supplier and consumer directly and via an event channel.

(d) DCE Event Management Service (EMS) will initially support the push model for communication between supplier and consumer.

(e) OMG Event Channels is a CORBA object that allows multiple suppliers to communicate with multiple consumers in an asynchronous fashion.

(f) DCE Event Management Service (EMS) is a DCE server that allows multiple suppliers to communicate with multiple consumers in an asynchronous fashion.

(g) OMG event service supports the notion of event suppliers and event consumers, where suppliers produce event data and consumers process event data.

(h) DCE Event Management Service supports the notion of event suppliers and event consumers, where suppliers produce event data and consumers process event data.

(i) OMG event service will not, to my knowledge, be integrated into the DCE event producing service (i.e., the DCE 1.1 Serviceability subsystem).

(j) DCE Event Management Service will be integrated with / hidden under the DCE event producing service (DCE 1.1 SVC).

15. Open Issues / Futures

(a) Event log size considerations.

Discussion: All the details on how big the event log gets, and what happens when the event log runs out of space have not been worked out.

(b) Notifying consumers of new event types.

Discussion: How does a consumer know that there are new event types available?

(c) Adding an event filter using the `dcecp` interface.

Discussion: The details for how to define filter expressions, and adding event filters to the Event Filter Database are being deferred to a future release.

(d) EMS Global Names.

Discussion: There are several EMS objects that require unique names. In particular consumers, event types and event filters. It is assumed that all these names will be registered in the CDS namespace. I am not sure if this is the correct place or not.

(e) `emsd` start-up options.

Discussion: All `emsd` start-up options have not been worked out

(f) Supplier-side filtering.

Discussion: How to efficiently keep event suppliers in sync with the current list of registered event types.

(g) Distributed EMS.

Discussion: Supplier-side filtering becomes much more important since RPCs are not guaranteed to be local. EMS management gets more complex. Would require support of replicas and synchronizing of EMS data within the cell or similar concept.

(h) OMG Event Service Full Function.

Discussion: Support/need for event Pull model via EMS (event channel). Support/need for event Push/Pull directly between supplier and consumer.

APPENDIX A. EVENT CONSUMER TEMPLATE

What follows is a sample of the structure of a typical event consumer:

```
/*****************************************************************/
/* SAMPLE CONSUMER: sample_consumer.c                            */
/* COPYRIGHT:                                                    */
/* ---------                                                     */
/* Copyright (C) International Business Machines Corp., 1992,1995. */
/*                                                               */
```

```
/* DISCLAIMER OF WARRANTIES:                                       */
/* -------------------------                                       */
/* The following [enclosed] code is sample code created by IBM     */
/* Corporation.  This sample code is not part of any standard IBM  */
/* product and is provided to you solely for the purpose of        */
/* assisting you in the development of your applications.  The     */
/* code is provided "AS IS", without warranty of any kind.  IBM    */
/* shall not be liable for any damages arising out of your use of  */
/* the sample code, even if they have been advised of the          */
/* possibility of such damages.                                    */
/*                                                                 */
/*******************************************************************/
/*                                                                 */
/*  Sample to demonstrate an Event Management Service consumer.    */
/*                                                                 */
/*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#ifdef IBMOS2
#include <process.h>
#define GETPID _getpid
#else
#include <unistd.h>
#define GETPID getpid
#endif
#include <pthread.h>
#include <time.h>

#include <dce/dce.h>
#include <dce/dce_error.h>
#include <dce/dce_cf.h>
#include <dce/dce_svc.h>
#include <dce/ems.h>
#include <dce/keymgmt.h>
#include <dce/sec_login.h>
#include <dce/secsts.h>
#include "sample_consumer.h"

/* fill in your HOST1 and HOST2 here, must be a DCE qualified name
   where emsd is running */

/*#define _TWOHOSTS                      // comment out for 1 host */

#define HOST1  "/.:/hosts/fixbuild.austin.ibm.com"
#ifdef _TWOHOSTS
#define HOST2  "/.:/hosts/shark.austin.ibm.com"
#endif

#define DEFAULT_CONS_NAME           "sampleConsumer"
#define PRINCIPAL_NAME              "consumer"

/* filter group names ... */
```

```
static ems_string_t  compSup             = (ems_string_t)"CompSup";
static ems_string_t  compSupAndSevNotice =
                                 (ems_string_t)"CompSupAndSevNotice";

/* severities */
static char *sev[] =
     {"INVALID","FATAL","ERROR","WARNING","NOTICE","NOTICE_VERBOSE"};

extern void MaintainLoginContext        (pthread_addr_t);
extern void ManageKey                   (pthread_addr_t);
boolean     does_filter_exist           (ems_handle_t*,ems_string_t);
void        BuildEventText              (ems_event_t*, char*, int);

/*---------------------------------------------------------------------
This is the event handler defined on the
ems_consumer_handler_register API call in the mainline code ...

This is where you might want to do creative things based on the
type of event that has been received.
---------------------------------------------------------------------*/

void log_event(ems_event_t *event,error_status_t *status)
{
   int   i;
   char  text[1025];

/* Build the event text for display
   (ie, SVC event schema "format" attr ... */
   BuildEventText(event,&text[0],1024);

/* determine the severity of the event */
   i = event->item[14].value.tagged_union.ushort_int;
   if ( i >= sizeof(sev)/sizeof(sev[0])) i = 0;

/* event processing ...
   display SVC event schema "format" attribute ... */
   fprintf(stdout,"%s: %s\n",sev[i],text);

/* return good status */
   *status = error_status_ok;
}

/*---------------------------------------------------------------------
This is the mainline code for the sample consumer ...  it must
perform the following functions:

1) call the ems_consumer_start() API to start the event consumer
2) call the ems_consumer_handler_register() API to register an
   event handler (ie, log_event in this case)
3) call the ems_consumer_register() API to register with the EMSD
   (ie, the EMS daemon) that this consumer is interested in
   receiving events from ...
4) setup the event filter expressions and/or event filter groups
5) call the rpc_server_listen() API to start listening for events
---------------------------------------------------------------------*/
```

```
main()
{
   ems_handle_t           emsHandle1, emsHandle2;
   ems_filter_exp_list_t *expList1, *expList2;
   ems_netname_t          netname;
   error_status_t         status;
   ems_filtername_list_t *fnList;
   char                  *server_name;

/* Generate the name of the consumer */
   server_name=(char*)malloc(36);
   if (server_name==NULL)
   {
      printf("*** Insufficient memory ***\n");
      exit(1);
   }

   sprintf( server_name, "%s%d", DEFAULT_CONS_NAME, GETPID());
   printf("    Name of the sample consumer = %s\n",server_name);

/* Establish network credentials and login context. */
   if (!EstablishLoginContextAndKey())
   {
      status=1;
      CHECK_STATUS("*** ALERT: EstablishLoginContextAndKey \
failed!\n",&status);
   }

/* This is called to start the event consumer */
   printf("    Start the sample consumer\n");
   ems_consumer_start
      (server_name,       /* name given to this instance of
                             the consumer */
       0,                 /* start flags - should be zeros */
       &status);          /* address of status - set in start code */
   CHECK_STATUS("ems_consumer_start", &status);  /* checks returned
                                                    status */

   free(server_name);

/* declare the event handler */
   printf("    Enable the sample consumer event handler\n");
   ems_consumer_handler_register
      (log_event,        /* name of the function that will process
                            the event */
       &status);         /* address of status - set in registration
                            code */
   CHECK_STATUS("ems_consumer_handler_register", &status);

/* allocate memory for an event filter expression list */
   expList1=(ems_filter_exp_list_t *)malloc(
                  sizeof(ems_filter_exp_list_t) +
                  1*sizeof(ems_filter_exp_t));
   if (expList1==NULL) {
      printf("*** Insufficient memory ***\n");
```

```
      exit(1);
   }

/* define a filter expression ...  this filter allows all events
   from SVC event component, "sup". */
   expList1->filter_exps[0].attr_name=(unsigned char *)
                                    "component_name";
   expList1->filter_exps[0].attr_operator=ems_c_attr_op_eq;
   expList1->filter_exps[0].attr_value.format=ems_c_attr_char_string;
   expList1->filter_exps[0].attr_value.tagged_union.char_string =
                                            (unsigned char *)"sup";
   expList1->size=1;   /* indicates the number of filter expressions
                          in list */

/* allocate memory for another event filter expression list */
   expList2=(ems_filter_exp_list_t *)malloc(
                    sizeof(ems_filter_exp_list_t) +
                    1*sizeof(ems_filter_exp_t));
   if (expList2==NULL) {
      printf("*** Insufficient memory ***\n");
      exit(1);
   }

/* define more filter expressions ... this filter allows all events
   from SVC event component, "sup" that are NOTICE events */
   expList2->filter_exps[0].attr_name=(unsigned char *)
                                    "component_name";
   expList2->filter_exps[0].attr_operator=ems_c_attr_op_eq;
   expList2->filter_exps[0].attr_value.format =
                                            ems_c_attr_char_string;
   expList2->filter_exps[0].attr_value.tagged_union.char_string =
                                            (unsigned char *)"sup";
   expList2->filter_exps[1].attr_name=
                             (unsigned char *)"attribute.severity";
   expList2->filter_exps[1].attr_operator=ems_c_attr_op_le;
   expList2->filter_exps[1].attr_value.format=ems_c_attr_ushort_int;
   expList2->filter_exps[1].attr_value.tagged_union.ushort_int=
                                          SVC_C_SEV_NOTICE_VERBOSE;
   expList2->size = 2; /* indicates the number of filter expressions
                          in list */

/* allocate memory for the event filter name list */
   fnList=(ems_filtername_list_t *)malloc(
                       sizeof(ems_filtername_list_t) +
                       1*sizeof(ems_string_t));
   if (fnList==NULL) {
      printf("+*** Insufficient memory ***\n");
      exit(1);
   }

   TRY
   {
      netname.service=ems_ns_dce; /* DCE CDS name service ...
                                  indicates which name service
                                  will recognize the network
```

```
                                name being defined here ... */
   /* allocate memory for the network address ...
      see HOST1 definition above.*/
   netname.netaddr=(ems_netaddr_t *)malloc(
                                   sizeof(ems_netaddr_t)+
                                   strlen(HOST1)+1);
   if (netname.netaddr==NULL) {
      printf("*** Insufficient memory ***\n");
      exit(1);
   }

/* complete the definition of the network name structure
   ... len, name ...*/
   netname.netaddr->len=strlen(HOST1)+1;
   strcpy((char *)&netname.netaddr->name[0],HOST1);

/* register network name with EMS (Event Management Service) */
   printf("    Register with %s\n",&netname.netaddr->name[0]);
   ems_consumer_register
      (&netname,      /* address of network name just defined */
       NULL,          /* requires an ems_add_filter_to_group
                         later */
       &emsHandle1,   /* address of handle set by registration
                         code */
       &status);      /* address of status set by registration
                         code */
   CHECK_STATUS("ems_consumer_register", &status);

   if (does_filter_exist(&emsHandle1,compSup)==FALSE)
   {
      ems_filter_add
         (emsHandle1,      /* handle returned by registration */
          compSup,         /* name of filter group for filter
                              expressions */
          ems_c_svc_type,  /* filter event type */
          expList1,        /* list of filter expressions */
          &status);        /* address of status set by
                              registration code */
      CHECK_STATUS("ems_filter_add",&status);
   }

   if (does_filter_exist(&emsHandle1,compSupAndSevNotice)==FALSE)
   {
      ems_filter_add
         (emsHandle1,          /* handle returned by registration
                                  */
          compSupAndSevNotice, /* name filter grp for filter
                                  expressions */
          ems_c_svc_type,      /* filter event type */
          expList2,            /* list of filter expressions */
          &status);            /* addr of status set by
                                  registration code */
      CHECK_STATUS("ems_filter_add", &status);
   }
```

```
   /* note that 2 filter groups were defined but only the latter is
      used */
      fnList->size=1;        /* indicates the number of filter names
                                in list */
      fnList->filter_names[0] = compSupAndSevNotice;

      printf("    Add filter compSupAndSevNotice to filter group\n");
      ems_add_filter_to_group
         (emsHandle1,        /* handle returned by registration */
          fnList,            /* list of filter names */
          &status);          /* address of status set by registration
                                code */
      CHECK_STATUS("ems_add_filter_to_group", &status);

      free(netname.netaddr);

#ifdef _TWOHOSTS
   /* allocate memory for a second network address ... */
      netname.netaddr=(ems_netaddr_t *)malloc(
                                       sizeof(ems_netaddr_t)+
                                       strlen(HOST2)+1);
      if (netname.netaddr==NULL) {
         printf("*** Insufficient memory ***\n");
         exit(1);
      }

   /* complete the definition of the network name structure
      ... len, name ... */
      netname.netaddr->len=strlen(HOST2)+1;
      strcpy( (char *)&netname.netaddr->name[0],HOST2);

   /* register network name with EMS (Event Management Service) */
      printf("    Register with %s\n",&netname.netaddr->name[0]);
      ems_consumer_register
         (&netname,      /* address of network name just defined */
          NULL,          /* requires an ems_add_filter_to_group later
                            */
          &emsHandle2,   /* address of handle set by registration
                            code */
          &status);      /* address of status set by registration
                            code */
      CHECK_STATUS("ems_consumer_register", &status);

      if (does_filter_exist(&emsHandle2,compSup)==FALSE)
      {
         ems_filter_add
            (emsHandle2,      /* handle returned by registration */
             compSup,         /* name of filter group for filter
                                 expressions */
             ems_c_svc_type,  /* filter event type */
             expList1,        /* list of filter expressions */
             &status);        /* address of status set by
                                 registration code */
         CHECK_STATUS("ems_filter_add",&status);
      }
```

```
      if (does_filter_exist(&emsHandle2,compSupAndSevNotice)==FALSE)
      {
         ems_filter_add
            (emsHandle2,           /* handle returned by registration
                                      */
             compSupAndSevNotice,  /* name filter grp for filter
                                      expressions */
             ems_c_svc_type,       /* filter event type */
             expList2,             /* list of filter expressions */
             &status);             /* addr of status set by
                                      registration code*/
         CHECK_STATUS("ems_filter_add", &status);
      }
   /* note that 2 filter groups were defined but only the former is
      used */
      fnList->size=1;        /* indicates the number of filter names
                                in list */
      fnList->filter_names[0] = compSup;

      printf("    Add filter compSupAndSevNotice to filter group\n");
      ems_add_filter_to_group
         (emsHandle2,          /* handle returned by registration */
          fnList,              /* list of filter names */
          &status);            /* address of status set by
                                  registration code */
      CHECK_STATUS("ems_add_filter_to_group", &status);

      free(netname.netaddr);
#endif

   /* start listening for events */
      printf("    Start listening for events\n");
      rpc_server_listen( 1, &status );
   }

   FINALLY
   {
   /* cleanup and exit ... */
      printf("    Start cleanup\n");
      ems_consumer_unregister( &emsHandle1, &status );
#ifdef _TWOHOSTS
      ems_consumer_unregister( &emsHandle2, &status );
#endif
      ems_consumer_stop( &status );
   }

   ENDTRY

} /* end main */

/*********************************************************************
Name                         = EstablishLoginContextAndKey()
Function                     = Gets network credentials and
                               establishes the login context.
Input                        = <none>
```

```
Output                        = <none>
Return                        = TRUE/FALSE
********************************************************************/

int EstablishLoginContextAndKey(void)
{
   sec_login_handle_t           login_context;
   sec_login_auth_src_t         auth_src;
   void                        *server_key;
   error_status_t               status;
   boolean32                    identity_valid;
   boolean32                    reset_passwd;
   int                          rstatus = TRUE;
   pthread_t                    refresh_login_context_thread;
   pthread_t                    manage_key_thread;


   do {
/*---------------------------------------------------------------------
The sec_login_setup_identity routine creates any local context
necessary to perform authenticated network operations.  It does not
establish any local operating system context; that is the
responsibility of the caller.  It is the standard network login
routine.  The network identity set up by this operation cannot be
used until it is validated by means of sec_login_validate_identity.
However, the sec_key_mgmt_get_key routine must be called first to
extract the specified key from the local key storage.

The sec_login_setup_identity and the sec_login_validate_identity
operations are two halves of a single logical operation.  Together
they collect the identity data needed to establish an authenticated
identity.
---------------------------------------------------------------------*/
      sec_login_setup_identity((unsigned_char_p_t)PRINCIPAL_NAME,
                               sec_login_no_flags,
                               &login_context,&status);
      CHECK_STATUS("*** ALERT: sec_login_setup_identity failed!",
                   &status);

      sec_key_mgmt_get_key(rpc_c_authn_dce_secret,NULL,
                           (idl_char *)PRINCIPAL_NAME,
                           0,&server_key,&status);
      CHECK_STATUS("*** ALERT: sec_key_mgmt_get_key failed!",
                   &status);

      identity_valid = sec_login_validate_identity(login_context,
                                       (sec_passwd_rec_t*)server_key,
                                       &reset_passwd,&auth_src,
                                       &status);
      CHECK_STATUS("*** ALERT: sec_login_validate_identity failed!",
                   &status);

/*---------------------------------------------------------------------
The sec_key_mgmt_free_key routine releases any storage allocated for
the indicated key data by sec_key_mgmt_get_key.  The storage for the
```

```
key data returned by sec_key_mgmt_get_key was dynamically allocated.
-------------------------------------------------------------------*/
      sec_key_mgmt_free_key(&server_key,&status);
      CHECK_STATUS("*** ALERT: sec_key_mgmt_free_key failed!",
                   &status);

      if (!(identity_valid && auth_src ==
            sec_login_auth_src_network)) {
         rstatus = FALSE;
         break;
      }

/*-------------------------------------------------------------------
The sec_login_set_context routine sets the network credentials to
those specified by the previously validated login context.

You cannot perform this function if the context is marked as private
(ie, sec_login_credentials_private).  Such a context can only be used
as an explicit context in user space applications.
-------------------------------------------------------------------*/
      sec_login_set_context(login_context,&status);
      CHECK_STATUS("*** ALERT: sec_login_set_context failed!",
                   &status);

/*-------------------------------------------------------------------
Start a thread to maintain the login context
-------------------------------------------------------------------*/
      TRY {
         pthread_create(&refresh_login_context_thread,
                         pthread_attr_default,
                        (pthread_startroutine_t)MaintainLoginContext,
                        (pthread_addr_t)login_context);
         pthread_detach(&refresh_login_context_thread);
      }
      CATCH_ALL {
         status=1;
         CHECK_STATUS("*** ALERT: THREAD CREATE ERROR!",&status);
      }
      ENDTRY


/*-------------------------------------------------------------------
Start a thread to manage the key
-------------------------------------------------------------------*/
      TRY {
         pthread_create(&manage_key_thread,
                         pthread_attr_default,
                        (pthread_startroutine_t)ManageKey,
                        (pthread_addr_t)NULL);
         pthread_detach(&manage_key_thread);
      }
      CATCH_ALL {
         status=1;
         CHECK_STATUS("*** ALERT: THREAD CREATE ERROR!",&status);
      }
```

```
      ENDTRY

   } while(0);

   return rstatus;

} /* end EstablishLoginContextAndKey */


/*********************************************************************
Name                        = MaintainLoginContext()
Function                    = Maintains login context.  This code
                              executes in the thread created in
                              the EstablishLoginContextAndKey()
                              routine.
Input                       = parg - DCE pthread args
Output                      = <none>
Return                      = <none>
*********************************************************************/

extern void MaintainLoginContext(pthread_addr_t parg)
{
   sec_login_handle_t           login_context =
                                (sec_login_handle_t)parg;
   signed32                     expiration;
   signed32                     delay_time;
   struct timespec              delay;
   unsigned32                   used_kvno;
   boolean32                    reset_passwd;
   boolean32                    identity_valid;
   boolean32                    *server_key;
   sec_login_auth_src_t         auth_src;
   error_status_t               status;
   time_t                       cur_time;

   while (1) {
      time(&cur_time);

      printf("\n*** Maintain Login Context ENTRY ***\n");
      printf("      %s\n",asctime(localtime(&cur_time)));

/*--------------------------------------------------------------------
The sec_login_get_expiration routine extracts the lifetime for the
TGT that belongs to the authenticated identity contained in the login
context.  The lifetime value is filled in if available, otherwise it
is set to 0 (zero).  This routine allows an application to tell an
interactive user how long the user's network login (and authenticated
identity) will last before it must be refreshed.

This routine works only on previously validated contexts.
--------------------------------------------------------------------*/
      sec_login_get_expiration(login_context,&expiration,&status);
      if ((status != rpc_s_ok) &&
          (status != sec_login_s_not_certified)) {
         CHECK_STATUS("*** ALERT: sec_login_get_expiration failed!",
```

```
                  &status);
      }

/*--------------------------------------------------------------------
The pthread_delay_np routine causes a thread to delay execution for a
specified period of elapsed wall clock time.  The period of time the
thread waits is at least as long as the number of seconds and
nanoseconds specified in the interval parameter.
--------------------------------------------------------------------*/
      if ((delay_time = expiration-cur_time-(600)) > 0) {
          /* 600 sec = 10 min*/
         delay.tv_sec = delay_time;     /* An integer number of
                                           seconds. */
         delay.tv_nsec = 0;             /* An integer number of
                                           nanoseconds. */
         pthread_delay_np(&delay);
      }

/*--------------------------------------------------------------------*/
The sec_login_refresh_identity routine refreshes a previously
established identity.  It operates on an existing valid context, and
cannot be used to change credentials associated with that identity.
The refreshed identity reflects changes that affect ticket lifetimes,
but not other changes.  For example, the identity reflects a change
to maximum ticket lifetime, but does not reflect the addition of the
identity as a member to a group.  Only a DCE login reflects all
administrative changes that have been made since the last login.

The network identity set up by this operation cannot be used until it
is validated by means of sec_login_validate_identity.  However, the
sec_key_mgmt_get_key routine must be called first to extract the
specified key from the local key storage.

The sec_login_refresh_identity and the sec_login_validate_identity
operations are two halves of a single logical operation.  Together
they collect the identity data needed to establish an authenticated
identity.

It is an error to refresh a locally authenticated context.
--------------------------------------------------------------------*/
      sec_login_refresh_identity(login_context,&status);
      CHECK_STATUS("*** ALERT: sec_login_refresh_identity failed!",
                   &status);

      sec_key_mgmt_get_key(rpc_c_authn_dce_secret,NULL,
                           (idl_char *)PRINCIPAL_NAME,
                           0,(idl_void_p_t *)&server_key,&status);
      CHECK_STATUS("*** ALERT: sec_key_mgmt_get_key failed",&status);

      identity_valid = sec_login_validate_identity(login_context,
                                       (sec_passwd_rec_t*)server_key,
                                       &reset_passwd,&auth_src,
                                       &status);
      CHECK_STATUS("*** ALERT: sec_login_validate_identity failed!",
                   &status);
```

```
/*---------------------------------------------------------------------
The sec_key_mgmt_free_key routine releases any storage allocated for
the indicated key data by sec_key_mgmt_get_key.  The storage for the
key data returned by sec_key_mgmt_get_key was dynamically allocated.
--------------------------------------------------------------------*/
      sec_key_mgmt_free_key(&server_key,&status);
      CHECK_STATUS("*** ALERT: sec_key_mgmt_free_key failed!",
                   &status);

      if (!identity_valid) {
         status=1;
         CHECK_STATUS("*** ALERT: INVALID IDENTITY!",&status);
         pthread_exit((pthread_addr_t)1);
      }

   } /* end while(1) */

} /* end MaintainLoginContext */

/*********************************************************************
Name                          = ManageKey()
Function                      = Manages the key.  This code runs in
                                thread created in
                                EstablishLoginContextAndKey()
                                routine.
Input                         = parg - NULL for this routine ...
Output                        = <none>
Return                        = <none>
********************************************************************/

extern void ManageKey(pthread_addr_t parg)
{
   error_status_t status;

   while (1) {
/*---------------------------------------------------------------------
The sec_key_mgmt_manage_key routine changes the specified principal's
key on a regular basis, as determined by the local cell's policy.  It
runs indefinitely and never returns during normal operation.  It
should be called only from a thread that has been devoted to managing
keys.

This routine queries the DCE Registry to determine the password
expiration policy that applies to the named principal.  It then idles
until a short time before the current key is due to expire and uses
the sec_key_mgmt_gen_rand_key to produce a new random key.  It
updates both the local key storage and the DCE Registry.
--------------------------------------------------------------------*/
      sec_key_mgmt_manage_key(rpc_c_authn_dce_secret,
                              NULL,
                              (idl_char *)PRINCIPAL_NAME,
                              &status);
      CHECK_STATUS("*** ALERT: sec_key_mgmt_manage_key failed!",
                   &status);
   }
```

```
} /* end ManageKey */

/*****************************************************************
Name                          = does_filter_exist()
Function                      = Check to see if a filter already
                                exists.
Input                         = EMSD handle and filter name
Output                        = <none>
Return                        = TRUE if it exists, FALSE is it does
                                not.
*****************************************************************/
boolean does_filter_exist( ems_handle_t *handle,
                           ems_string_t filter_name )
{
   int                    i;
   error_status_t         status;
   ems_filtername_list_t *fnList;

   ems_filter_get_namelist(*handle,&fnList,&status);
   if (status) return FALSE;

   for (i=0;i<fnList->size;i++)
   {
      if (strcmp((char*)fnList->filter_names[i],
                 (char *)filter_name)==0)
      {
         return TRUE;
         break;
      }
   }

   ems_filter_free_namelist(&fnList,&status);
   /* This status is ignored */
   return FALSE;

} /* end does_filter_exist */

/*****************************************************************
Name       = BuildEventText()
Function   = Builds event message text using the SVC format attribute
             which contains the %s, %d, etc.  It is also supposed to
             handle the cases where %1$d, %2$s, %3$6ld, %6ld, etc.
Input      = event  - EMS event
           = pTxt   - Pointer to where the resolved text is placed
           = maxlen - Max length of pTxt area
Output     = pTxt   - Updated ...
Return     = <none>
*****************************************************************/

#define SVC_FORMAT_ATTR    6   /* INDEX into SVC event type schema */
#define SVC_ARGTYPES_ATTR  2

void BuildEventText(ems_event_t *event,char *pText, int maxlen)
{
   int          cntr=16;             /* first 16 items are for SVC */
```

```
char         *pTmp=NULL;
char         *pTxt=pText;
size_t        len;
char const   *pBeg=(const char*)event->
              item[SVC_FORMAT_ATTR].value.tagged_union.char_string;
char const   *pEnd;
char         *pArg=NULL;
int           islong=FALSE;
char          fmt[32];
char          realfmt[32];
size_t        realcnt=0;
char          cnt[2];
size_t        fmtlen=0;
int           done;

if ((pText==NULL)||(maxlen==0)) return;

do
{
/* If no args to be parsed, just copy the message. */
   if (event->count<=16)
   {
      strcpy(pTxt,pBeg);
      break;
   }

   pTmp=(char*)malloc(maxlen+1);
   pTmp[0]=0x00;

   pArg=strstr(pBeg,"%");

/* Parse the message and fill in the args. */
   for (done=FALSE; !done; pArg++)
   {
      fmt[fmtlen++]=*pArg;

      switch (*pArg)
      {
         default:
            if ((*pArg>'0')&&(*pArg<='9'))
            {
               if (*(pArg+1)=='$')
               {
                  cnt[0]=*pArg;
                  cnt[1]=0;
                  cntr=atoi(cnt);
                  cntr+=15;
               }
            }
            continue;
         case '%':
            memset(fmt,0x00,sizeof(fmt));
            fmtlen=1;
            fmt[0]='%';
            memset(realfmt,0x00,sizeof(realfmt));
```

```
            realcnt=1;
            realfmt[0]='%';
            continue;
         case 'l':
            islong=TRUE;
            realfmt[realcnt++]='l';
            continue;
         case 'd': case 'i': case 'o': case 'x': case 'X':
            if (!(pEnd=strstr(pBeg,fmt)))
            {
               strcpy(pTxt,pBeg);
               done=TRUE;
               break;
            }

            realfmt[realcnt++]=*pArg;
            len=pEnd-pBeg;
            memcpy(pTmp,pBeg,len);
            *(pTmp+len)=0;
            pBeg=pEnd+strlen(fmt);
            strcat(pTmp,realfmt);

            if (islong)
               sprintf(pTxt,pTmp,
                 event->item[cntr++].value.tagged_union.long_int);
            else
               sprintf(pTxt,pTmp,
                event->item[cntr++].value.tagged_union.short_int);

            pTxt+=strlen(pTxt);
            done=TRUE;
            break;
         case 'u':
            if (!(pEnd=strstr(pBeg,fmt))) {
               strcpy(pTxt,pBeg);
               done=TRUE;
               break;
            }

            realfmt[realcnt++]=*pArg;
            len=pEnd-pBeg;
            memcpy(pTmp,pBeg,len);
            *(pTmp+len)=0;
            pBeg=pEnd+strlen(fmt);
            strcat(pTmp,realfmt);

            if (islong)
               sprintf(pTxt,pTmp,
                event->item[cntr++].value.tagged_union.ulong_int);
            else
               sprintf(pTxt,pTmp,
               event->item[cntr++].value.tagged_union.ushort_int);

            pTxt += strlen(pTxt);
            done=TRUE;
```

```
      break;
   case ’f’: case ’e’: case ’E’: case ’g’: case ’G’:
      if (!(pEnd=strstr(pBeg,fmt))) {
         strcpy(pTxt,pBeg);
         done=TRUE;
         break;
      }

      realfmt[realcnt++]=*pArg;
      len=pEnd-pBeg;
      memcpy(pTmp,pBeg,len);
      *(pTmp+len)=0;
      pBeg=pEnd+strlen(fmt);
      strcat(pTmp,realfmt);

      sprintf(pTxt,pTmp,
         event->item[cntr++].value.tagged_union.long_float);
      pTxt+=strlen(pTxt);
      done=TRUE;
      break;
   case ’c’:
      if (!(pEnd=strstr(pBeg,fmt))) {
         strcpy(pTxt,pBeg);
         done=TRUE;
         break;
      }

      realfmt[realcnt++]=*pArg;
      len=pEnd-pBeg;
      memcpy(pTmp,pBeg,len);
      *(pTmp+len)=0;
      pBeg=pEnd+strlen(fmt);
      strcat(pTmp,realfmt);

      sprintf(pTxt,pTmp,
         event->item[cntr++].value.tagged_union.ushort_int);
      pTxt+=strlen(pTxt);
      done=TRUE;
      break;
   case ’s’:
      if (!(pEnd=strstr(pBeg,fmt))) {
         strcpy(pTxt,pBeg);
         done=TRUE;
         break;
      }

      realfmt[realcnt++]=*pArg;
      len=pEnd-pBeg;
      memcpy(pTmp,pBeg,len);
      *(pTmp+len)=0;
      pBeg=pEnd+strlen(fmt);
      strcat(pTmp,realfmt);

      sprintf(pTxt,pTmp,
        event->item[cntr++].value.tagged_union.char_string);
```

```
            pTxt+=strlen(pTxt);
            done=TRUE;
            break;
         case 'p':
            if (!(pEnd=strstr(pBeg,fmt))) {
               strcpy(pTxt,pBeg);
               done=TRUE;
               break;
            }

            realfmt[realcnt++]=*pArg;
            len=pEnd-pBeg;
            memcpy(pTmp,pBeg,len);
            *(pTmp+len)=0;
            pBeg=pEnd+strlen(fmt);
            strcat(pTmp,realfmt);

            sprintf(pTxt,pTmp,
              event->item[cntr++].value.tagged_union.byte_string);
            pTxt+=strlen(pTxt);
            done=TRUE;
            break;
         case 'b':
            if (!(pEnd=strstr(pBeg,fmt))) {
               strcpy(pTxt,pBeg);
               done=TRUE;
               break;
            }

            realfmt[realcnt++]=*pArg;
            len=pEnd-pBeg;
            memcpy(pTmp,pBeg,len);
            *(pTmp+len)=0;
            pBeg=pEnd+strlen(fmt);
            strcat(pTmp,realfmt);

            sprintf(pTxt,pTmp,
               event->item[cntr++].value.tagged_union.bytes.data);
            pTxt+=strlen(pTxt);
            done=TRUE;
            break;
      }

      islong=FALSE;

      if (done==TRUE)
      {
         pArg=strstr(pBeg,"%");
         if (pArg)
         {
            pArg--;
            done=FALSE;
         }
      }
```

```
        }

    /* Copy the remaining format string. */
        if (*pBeg) strcpy(pTxt,pBeg);

        free(pTmp);

    } while(0);

} /* end BuildEventText */
```

APPENDIX B. VALID FILTER EXPRESSIONS

The following table describes which filter operators are valid with which attribute types (T = TRUE, F = FALSE). Operations on data type ACL are not supported in release 1.

Data type	eq	gt	lt	ge	le	ne	bitand	substr
`small int`	T	T	T	T	T	T	F	F
`short int`	T	T	T	T	T	T	F	F
`long int`	T	T	T	T	T	T	F	F
`hyper int`	T	T	T	T	T	T	F	F
`usmall int`	T	T	T	T	T	T	T	F
`ushort int`	T	T	T	T	T	T	T	F
`ulong int`	T	T	T	T	T	T	T	F
`uhyper int`	T	T	T	T	T	T	F	F
`short float`	T	T	T	T	T	T	F	F
`long float`	T	T	T	T	T	T	F	F
`boolean`	T	F	F	F	F	T	F	F
`uuid`	T	T	T	T	T	T	F	F
`utc`	T	T	T	T	T	T	F	F
`severity`	T	T	T	T	T	T	F	F
`acl`	F	F	F	F	F	F	F	F
`byte string`	T	F	F	F	F	T	F	F
`char string`	T	T	T	T	T	T	F	T
`bytes`	T	F	F	F	F	T	T	F
`octet array`	T	F	F	F	F	T	T	F

Table 12: Filter Expression Operator Table

REFERENCES

[OMG] ‘‘Joint Object Services Submission — Event Service Specification’’, July 2, 1993 (document TC 93.7.3).

[DME] Lisa Ferrante ‘‘DME Event Services Architecture, Version 2.0’’, April 28, 1993.

[RFC 24.1] R. Salz, ‘‘DCE 1.1 Serviceability Proposal’’, April 28, 1993.

[RFC 21.0] N. Mishkin, ‘‘DCE RPC API Extensions for Modular Servers’’, November 1992.

AUTHORS' ADDRESSES

Rick Cohen
IBM
11400 Burnett Road
Austin TX 78758
USA
Internet email: cohen@austin.ibm.com
Telephone: +1-512-838-8198

Gregg Wilson
IBM
11400 Burnett Road
Austin TX 78758
USA
Internet email: gwilson@austin.ibm.com
Telephone: +1-512-838-2244