Telecom Log Service Specification

July 2003
Version 1.1.2 - editorial update
formal/03-07-xx

An Adopted Specification of the Object Management Group, Inc.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

This specification represents the hard work and contribution of many individuals and companies. We would like to acknowledge the following for their contributions, both large and small:

- Alcatel Corporate Research Center
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Expersoft Corporation
- Hewlett Packard Company

- Nortel Technology
- Telefónica Investigación y Desarrollo S.A. Unipersonal

Telecom Log Service 1

Note – Editorial changes are in this color.

Contents

This chapter contains the following sections.

1.1 Introduction

This chapter presents a specification of the Telecom Log Service that can be used in a pure CORBA environment as well as by TMN systems via a gateway function. This specification not only supports the functionality defined in the ITU-T X.735 specification, but also extends it to allow logging of any type of event and querying of LogRecords based on constraint languages. It also provides capabilities to form log networks for storing and forwarding events.

1.2 The Log Interfaces

Log objects are consumers of events. Log objects store events that fulfill some constraint expression (a filter).

Each log object has a log repository that can consist of one or more files or databases. A log object stores events as entries in its associated log repository. Log entries are accessible by querying the log. Events stored in a log can be of any type, as long as the underlying log repository supports their storage.

Log objects are suppliers of events. A log object generates events when its capacity threshold is reached, one of its log attributes change, its log state changes, or the log is deleted.

Log objects behave as event suppliers when they are connected with other logs to form a “log and forward” network.

In summary, a log object is both an event consumer and event supplier. It interacts with CORBA events in three distinct ways:

1. ***Writing to a log***: Events supplied to a log are stored as log records. These are incoming events.
2. ***Forwarding from a log***: Events supplied to a log are also forwarded to other logs or to any application that wishes to receive them. These are outgoing events.
3. ***Log generated events***: Logs will generate their own events when something of their state changes, for instance when a threshold is being crossed or a log attribute is changed. These are also outgoing events.

An event channel interacts with events in two ways (incoming and outgoing). Log writing and forwarding naturally map to an event channel. The log generated events are emitted through the log factory (see Section 1.3, “The Log Factory Interfaces,” on page 1-25) and are different from those that are forwarded through the log.

To take advantage of the natural mapping described above, logs are modeled as event/notification channels. Log inherits from the typed or untyped event/notification channel. Events are forwarded to a log object by connecting to the event channel or notification channel of the log. By leveraging features of event/notification channel, logs can:

- Support both the push and pull models of event communication.
- Support either the typed or untyped models of event communication.
- Support multiple suppliers, and allow them to disconnect cleanly. The one to one supplier to proxy consumer relationship of the Event Service is preserved.
- Support multiple consumers, and allow them to disconnect cleanly. The one to one consumer to proxy supplier relationship of the Event Service is preserved.
- Form a log network of various topologies by simply connecting to other logs as event channels. A log network supports “log-and-forward” scenarios.

A log that inherits from notification channel can apply filtering to the events it logs. Filters are set and modified using the **CosNotifyFilter::Filter** object associated with a log and applies to events before they are logged. There can also be filters on the notification channel to filter incoming or outgoing events.

Logs can leverage from the notification service QoS framework to support the QoS requirements on the notification aspect of logs. Figure 1-1 gives a graphical representation of the log.

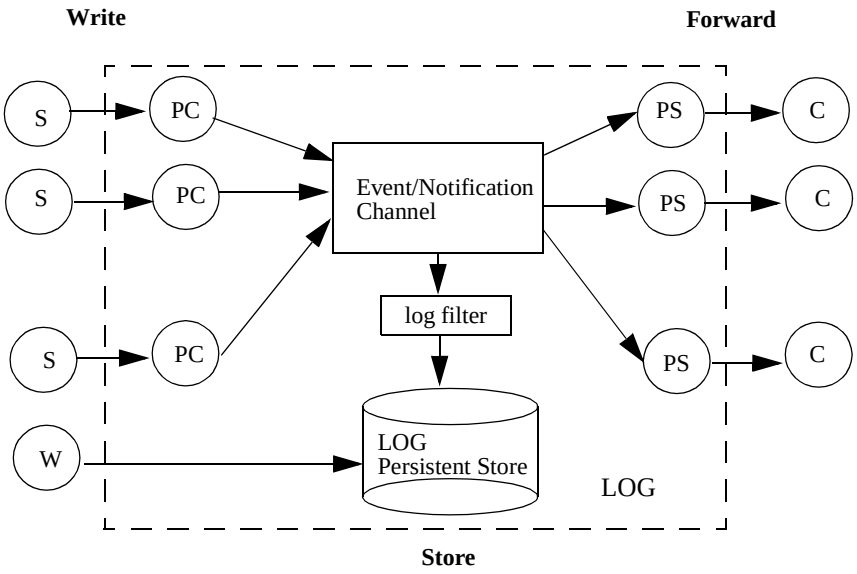

Figure 1-1 The Log

Figure Key:

- S: event/notification supplier
- W: event/notification unaware writer
- PC: proxy consumer
- PS: proxy supplier
- C: event/notification consumer

1.2.1 Log Inheritance

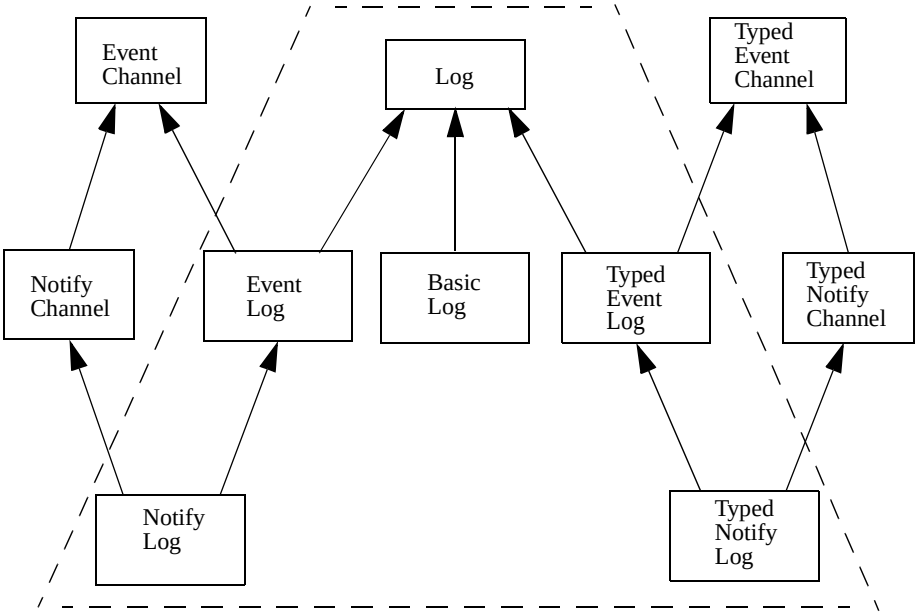

Figure 1-2 Log Inheritance

There are six types of logs, each type of log has its own corresponding log factory (see Section 1.3, “The Log Factory Interfaces,” on page 1-25 for more details):

- The **Log** interface does not inherit from any event service interfaces. **Log** serves as an abstract interface to which all other log interfaces can inherit from. **Log** defines attributes and operations common to all other log interfaces.
- The **BasicLog** interface inherits from the **Log** interface. **BasicLog** allows “event unaware” clients to access a log directly without any knowledge of events or notifications. It does not support event forwarding, nor does it emit log events (see Section 1.4, “Log Generated Events,” on page 1-29 for more details).
- The **EventLog** interface inherits from the **Log** and **CosEventChannelAdmin::EventChannel** interfaces. **EventLog** receives and forwards untyped events via a **CosEventChannelAdmin::EventChannel**. **EventLog** is created by and emits log generated events via the **EventLogFactory** interface (see Section 1.3, “The Log Factory Interfaces,” on page 1-25 for more details).
- The **NotifyLog** interface inherits from the **EventLog** and **CosNotifyChannelAdmin::EventChannel** interfaces. **NotifyLog** receives and forwards untyped events via a **CosNotifyChannelAdmin::EventChannel**. **NotifyLog** supports filtering on incoming, logged, and outgoing events. **NotifyLog** is created by and emits log generated events via the **NotifyLogFactory** interface (see Section 1.3, “The Log Factory Interfaces,” on page 1-25 for more details).

- The **TypedEventLog** interface inherits from the **Log** and **CosTypedEventChannelAdmin::TypedEventChannel** interfaces. **TypedEventLog** receives and forwards typed events via a **CosTypedEventChannelAdmin::TypedEventChannel**. **TypedEventLog** is created by and emits log generated events via the **TypedEventLogFactory** interface (see Section 1.3, “The Log Factory Interfaces,” on page 1-25 for more details).
- The **TypedNotifyLog** interface inherits from the **TypedEventLog** and **CosTypedNotifyChannelAdmin::TypedEventChannel** interfaces. **TypedNotifyLog** receives and forwards events via a **CosTypedNotifyChannelAdmin::TypedEventChannel**. In addition, it supports filtering on incoming, logging and outgoing events. It emits events via the **TypedNotifyLogFactory** (see Section 1.3, “The Log Factory Interfaces,” on page 1-25 for more details).

Note – The **TypedEventLog** and **TypedNotifyLog** interfaces are optional for an implementation of the Telecom Log Service.

This inheritance allows clients to access the log at any level it wishes:

- directly
- event service
- notification service
- typed event service
- typed notification service

A client can widen a derived log interface to any of its base interfaces. For instance a **NotifyLog** object can be widened to either an **EventLog** for event aware applications, or a **Log** for “event unaware” applications.

The following table lists the features supported by various types of logs.

Table 1-1 Features of logs

	write	**store**	**forward**	**emit events**	**QoS**
BasicLog	write operation	no filter	no	no	log QoS
EventLog	untyped push/pull supplier; no filter	no filter	untyped push/pull consumer; no filter	emit via EventLogFactory; no filter	log QoS
NotifyLog	untyped/structured push/pull supplier; notification filter applied to incoming events	log filter applied in addition to notification filter; only filtered events are logged	untyped/structured push/pull consumer; notification filter applied to outgoing events	emit via NotifyLogFactory filter applied to outgoing events	notification QoS + log QoS
TypedEventLog	typed push/pull supplier; no filter	no filter	typed push/pull consumer; no filter	emit via TypedEventLog Factory no filter	log QoS
TypedNotify Log	typed push/pull supplier; notification filter applied to incoming events	log filter applied in addition to notification filter; only filtered events are logged	typed push/pull consumer; notification filter applied to outgoing events	emit via TypedNotifyLog Factory filters applied to outgoing events	notification QoS + log QoS

1.2.2 LogRecord and TypedLogRecord

Log records are created as a result of the receipt of events or notifications, either via the event or notification channel, or by explicitly invoking one of the write operations. A **LogRecord** contains the following:

```
struct LogRecord {
    RecordId      id;
    TimeT         time;
    NVList        attr_list;
    any           info;
};
```

- **id** - Unique number assigned to the record by the log.
- **time** - Timestamp indicating the time an event is logged.
- **attr_list** - User defined name/value pairs that are not a part of the event received by the log. These attributes contain log record related information and can be queried or modified.

- **info** - The event data stored in a CORBA any. Note that a structured or typed event can be wrapped in a CORBA any as specified by the Notification Service specification.

Typed events can also be stored in a typed log as **TypedLogRecord**, which contains the following:

```
struct TypedLogRecord {
    DsLogAdmin::RecordId        id;
    DsLogAdmin::TimeT           time;
    DsLogAdmin::NVList          attr_list;
    CORBA::RepositoryId         interface_id;
    CORBA::Identifier           operation_name;
    ArgumentList                arg_list;
};
```

- **id** - Unique number assigned to the record by the log.
- **time** - Timestamp indicating the time the event was logged.
- **attr_list** - User defined name/value pairs that are not a part of the event received by the log. These attributes contain log record related information and can be queried or modified.
- **interface_id** - Repository id of the interface that sent the typed event.
- **operation_name** - Name of the operation that emitted the typed event.
- **arg_list** - Argument list that contains the event data.

LogRecords and **TypedLogRecords** can be queried, retrieved, and deleted based on record id, log time, attributes, or event contents (see Section 1.2.5, “Log Record Manipulation,” on page 1-18 for more details).

Note – The **LogRecord** format is supported by all log interfaces. **TypedLogRecord** format is only supported by **TypedEventLog** and **TypedNotifyLog**. The same typed event can be represented as a **LogRecord** or a **TypedLogRecord**, depending on the operations client invokes to retrieve it (see Section 1.2.5, “Log Record Manipulation,” on page 1-18 for more details).

1.2.3 Logging Scenarios

Figure 1-3 on page 1-8 shows the basic steps involved in logging.

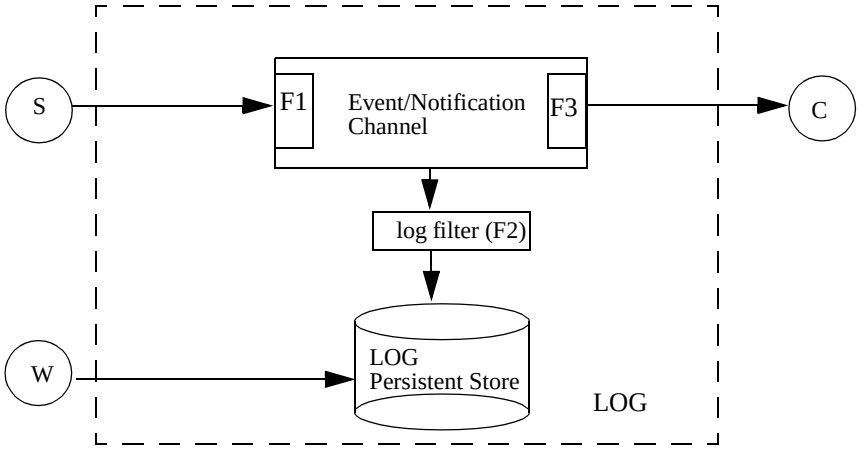

Figure 1-3 Event Logging and Forwarding

Figure key:

- S: even/notification supplier
- W: event/notification unaware writer
- F1, F3: notification filters
- F2: log filter
- C: event/notification consumer

The basic steps involved in logging are:

1. The client (push or pull style event/notification supplier) connects to the log object via the event or notification channel interface it supports.
2. **NotifyLog** or **TypedNotifyLog** clients can specify filters (F1) on the supplier side of notification channel.
3. **NotifyLog** or **TypedNotifyLog** clients can specify filter (F2) on the log object.
4. The client sends events to the log object. Event aware clients invoke one of the push or pull operations while event unaware clients invoke the write operation.
5. Notification filters (F1) are applied to incoming events.
6. The log object receives events from the client.
7. The log filters (F2) are applied to each event.
8. For each event that passes both filters F1 and F2 a log record is created that consists of the log record id, the log time, and the event data. The **LogRecord** data is stored in the log repository and can be retrieved via the query operations supported by the log.

As shown in Figure 1-3 on page 1-8, the basic steps involved in log event forwarding are:

- The client (push or pull style event/notification consumer) connects to the log object via the event or notification channel interface it supports.
- **NotifyLog** or **TypedNotifyLog** clients can specify filters (F3) on the consumer side of notification channel.
- The client sends events to the log object. Event aware client uses push or pull model; event unaware client invokes the write operation.
- Notification filters (F1) are applied to incoming events.
- Notification filters (F3) are applied to outgoing events.
- Events that pass both filters F1 and F3 are forwarded to the client (event/notification consumer object).

See Section 1.3, “The Log Factory Interfaces,” on page 1-25 and Section 1.4, “Log Generated Events,” on page1-29 for details on receiving log generated events from log factories.

1.2.4 Log Control and Management

The **Log** interface supports the following log control and management functions.

Monitoring and setting of:

- administrative state
- availability status
- maximum log size
- log full action
- log duration
- log scheduling
- log capacity thresholds
- expiration time (lifetime) for log records
- quality of service properties

Monitoring only of:

- operational state
- current log size (in octets and number of records)

EventLog and **TypedEventLog** also support monitoring and setting of log forwarding state.

NotifyLog and **TypedNotifyLog** also support event filtering on incoming, logged, and outgoing events.

1.2.4.1 Log Id and Factory

LogId id();
LogMgr my_factory();

The **id()** operation returns the **id** of a log. Log objects are created by a log factory (described in the next section) and each log is assigned an **id** that uniquely identifies the log object within the factory.

The **my_factory()** operation returns the log factory object that created the log. The return type **LogMgr** is the abstract interface of all log factories. A client should narrow **LogMgr** to the actual derived log factory interface (see Section 1.3.1, "Log Factory Inheritance," on page 1-26 for more details on log factory inheritance). Clients need to narrow the **LogMgr** to a specific log factory interface if they wish to create new logs or receive log generated events.

Note that the **CosNotifyChannelAdmin::EventChannel** interface defines a **MyFactory** attribute that returns a **CosNotifyChannelAdmin::NotifyFactory** object. It is up to the implementation of the Telecom Log Service to use either the **NotifyLogFactory** or **CosNotifyChannelAdmin::NotifyFactory** to create the notification channel. Therefore, for a **NotifyLog** object, the **my_factory()** operation will return a **NotifyLogFactory** object, while the **MyFactory** attribute (inherited from **CosNotifyChannelAdmin::EventChannel** interface) may return a **CosNotifyChannelAdmin::NotifyFactory** object or raise a CORBA **NO_IMPLEMENT** exception (since it cannot return the **NotifyLogFactory**).

1.2.4.2 Operational State

enum OperationalState {disabled, enabled};

OperationalState get_operational_state();

The **get_operational_state()** operation returns the operational capability of the log to perform its function. The following operational states are defined:

- **enabled**: The log is operational and ready for use.
- **disabled**: The log is not available for use due to some run time problem. New records cannot be created.

A **StateChange** event is generated whenever the operational state of a log changes.

1.2.4.3 Administrative State

enum AdministrativeState {locked, unlocked}; // logging on/off

AdministrativeState get_administrative_state();

void set_administrative_state(in AdministrativeState state);

- The **get_administrative_state()** operation returns the administrative state of the log.
- The **set_administrative_state()** operation sets the administrative state of the log.

These operations allow clients to control the administrative capability of the log to perform its function. The following administrative states are defined:

- **Unlocked**: Use of the log has been permitted by a managing system. Information from subordinate records may be retrieved and, conditional on the values of other state and status attributes, new records may be created.
- **Locked**: Use of the log has been prohibited by a managing system. Log records may be retrieved but new records will not be created. Records may be deleted.

By default, the administrative state is set to “unlocked” when a log object is created.

Administrative state can also be thought of as the “logging state” and is used to turn logging on and off. Administrative state does not affect log’s ability to forward events. If the administrative state of the log is locked, events will pass through the event/notification channel as long as the forwarding state is on.

A **StateChange** event is generated whenever the administrative state of a log is set.

1.2.4.4 Log Size

```
// size in bytes
ulong long get_max_size();
void set_max_size(in ulong long size) raises (InvalidParam);

ulong long get_current_size(); // size in bytes
ulong long get_n_records();
```

The **get_max_size()** operation returns the size of the log measured in number of bytes. A log may have an indeterminate size.

The **set_max_size()** operation sets the maximum log size. If a value of zero is supplied, then the log size will be set to have no predefined limit. If the maximum log size specified is less than the current log size, an **InvalidParam** exception will be raised.

The **get_current_size()** operation returns the current size of the log measured in bytes.

The **get_n_records()** operation returns the current number of records contained in the log.

An **AttributeValueChange** event is generated whenever the size of a log is set.

1.2.4.5 Log Full Action

```
typedef unsigned short LogFullActionType;

const LogFullActionType wrap = 0;
const LogFullActionType halt = 1;
LogFullActionType get_log_full_action();
void set_log_full_action(in LogFullActionType action)
                raises (InvalidLogFullAction);
```

The **get_log_full_action()** operation returns the action that will be taken when the maximum size of the log has been reached. Two options are currently defined:

- **wrap**: The oldest records in the log, based on the log time, will be deleted to free resources for the creation of new records.
- **halt**: No more records will be logged and all incoming events are discarded. Records already in the log will be retained.

Note – A specific implementation of the Telecom Log Service can define and support more **LogFullActionTypes** if desired.

The **set_log_full_action()** operation sets which action should be taken when the maximum size of the log has been reached.

When a log is full its availability status will change to indicate the log full condition. If the log's full action is set to "halt," then no new log records will be added and the event will be lost. When log records are deleted the log will leave the log full condition and new log records can be added to the log. If the log's full action is set to "wrap," then new records will be added overwriting the oldest records contained in the log. When log records are deleted the log will leave the log full condition and new log records will no longer overwrite existing log records in the log.

An **AttributeValueChange** event is generated whenever the log full action of a log is set.

1.2.4.6 Log Duration

```
struct TimeInterval {
    TimeT start;
    TimeT stop;
    };
TimeInterval get_interval();
void set_interval(in TimeInterval interval)
                raises (InvalidTime, InvalidTimeInterval);
```

The **get_interval()** operation returns the coarse grained time interval during which an unlocked and enabled **Log** is functional.

The **set_interval()** operation sets the start and stop time during which an unlocked and enabled log is functional.

The **start** field indicates the date and time at which an unlocked and enabled log starts functioning. Specifying 0 for the start field means to start logging immediately.

The **stop** field indicates the date and time at which an unlocked and enabled lock stops functioning. Specifying 0 for the stop field means to keep logging indefinitely (do not stop until the log is destroyed).

A **race** condition could exist when setting the start/stop time. For instance, if a Log start time is specified so close to the time the **set_interval()** operation is invoked, then by the time the Log is activated it may have missed some events that should have been logged. To avoid this race condition, it is recommended that:

- If a client wants to set the start/stop time immediately after a Log is created, then the client should invoke the **set_interval()** operation before connecting the Log to an event supplier.
- If a Log is in operation, then the client should set the administrative state of the log to locked by invoking the **set_administrative_state()** operation before invoking the **set_interval()** operation.

By default, the duration of a log is for its lifetime and logging takes place immediately after the log is created and stops when the log is destroyed.

Log duration does not effect the status of event forwarding (see Section 1.2.4.12, “Forwarding State,” on page 1-17).

An **AttributeValueChange** event is generated whenever the log duration interval of a log is set (either start or stop).

1.2.4.7 Log Scheduling

```
struct Time24 {
    unsigned short hour;  // 0-23
    unsigned short minute;// 0-59
};

struct Time24Interval {
    Time24 start;
    Time24 stop;
};
typedef sequence<Time24Interval> IntervalsOfDay;

const short Sunday      =  1;
const short Monday      =  2;
const short Tuesday     =  4;
const short Wednesday   =  8;
const short Thursday    = 16;
const short Friday      = 32;
const short Saturday    = 64;

typedef short DaysOfWeek;// Bit mask of week days
```

```
struct WeekMaskItem {
    DaysOfWeek        days;
    IntervalsOfDay    intervals;
};
typedef sequence<WeekMaskItem> WeekMask;

WeekMask get_week_mask();
void set_week_mask(in WeekMask masks)
                raises (InvalidTime, InvalidTimeInterval, InvalidMask);
```

The **get_week_mask()** operation returns the fine grained time intervals during which an unlocked and enabled Log is functional.

- The **start** field of **Time24Interval** indicates the hour and minute of the day when an unlocked and enabled log starts functioning.
- The **stop** field of **Time24Interval** indicates the hour and minute of the day when an unlocked and enabled log stops functioning.
- The **days** field of **WeekMaskItem** indicates which days of the week the start and stop time fields are valid (a bit mask OR'ed together to specify week days, with the 1st bit indicating Sunday and the 7th bit indicating Saturday).

These fields (**start**, **stop**, **days**) are contained in a **WeekMaskItem** structure. A **week_mask** may contain more than one **WeekMaskItem** structure.

The **set_week_mask()** operation sets the fine grained time intervals during which an unlocked and enabled log is functional. These fine grained time intervals are only valid within the coarse grained time interval specified by the log duration operations.

The weekly scheduling only applies during log duration time (start/stop time). A log performs its logging function when the logging time falls within the log duration and one or more of the log scheduling times is indicated in the weekly mask.

The weekly schedule does not effect the status to event forwarding (see Section 1.2.4.12, "Forwarding State," on page 1-17).

By default, the weekly mask is empty when a log is created. An empty weekly mask means that logging should take place during the whole week.

An **AttributeValueChange** event is generated whenever the week mask of a log is set.

1.2.4.8 *Availability Status*

```
struct AvailabilityStatus {
    boolean off_duty;
    boolean log_full;
};
```

```
AvailabilityStatus get_availability_status();
```

The **get_availability_status()** operation returns a struct that reflects the availability status of the **Log**.

The **log_full** field indicates whether the log is full or not. If **log_full** equals TRUE, then records can only be retrieved; however, no new records can be added.

The **off_duty** field indicates whether the log is scheduled to log events or not. A log is considered "on duty" only if all of the following are true:

- Operational state is "enabled."
- Administrative state is "unlocked."
- Current time falls within the log duration time.
- Current time falls within one (or more) of the log scheduling times.

1.2.4.9 *Log Record Compaction*

```
unsigned long get_max_record_life();
void set_max_record_life(in unsigned long life);
```

The **get_max_record_life()** operation returns the maximum number of seconds a record is stored in a log.

The **set_max_record_life()** operation sets the maximum number of seconds a record is stored in a log.

Log record compaction can be achieved by setting a maximum record lifetime (in seconds) to automatically clean out stale records. A value of 0 effectively disables this feature and is interpreted to store the log records indefinitely (until the log is destroyed).

By default, the maximum record life is set to 0 upon log creation.

An **AttributeValueChange** event is generated whenever the max record life of a log is set.

1.2.4.10 *Quality of Service*

```
typedef unsigned long QoSType;
typedef sequence<QoSType> QoSList;

const QoSType QoSNone        = 0;
const QoSType QoSFlush       = 1;
const QoSType QoSReliability = 2;

QoSList get_log_qos();
void set_log_qos(in QoSList qos) raises(UnsupportedQoS);
void flush() raises(UnsupportedQoS);
```

The Notification Service has specified a very elaborate QoS administrative framework. This framework is leveraged to handle the QoS for the notification channel aspect of logs.

In addition to the QoS framework defined by the Notification Service, The Telecom Log Service defines a lightweight QoS framework that adds some additional QoS properties that apply only to logs. This framework can easily be extended to handle proprietary QoS features that various implementations deem appropriate.

The following quality of service properties are supported by the log interface:

- QoSNone: No quality of service is promised by the log implementation.
- QoSFlush: Log records are made persistent by the log. It is left up to the implementation to decide on the various persistence strategies (e.g., write through vs. write back) to support. A log provides its client with a degree of control by supporting the QoSFlush property and the **flush()** operation.
- QoSReliability: All log records sent to Log are guaranteed to be available. This QoS may suffer a performance hit, but will gain high availability (crash recovery).

The **get_log_qos()** operation returns a list of the quality of service properties supported by the log.

The **set_log_qos()** operation sets the quality of service properties of the log. If an implementation does not support the specified quality of service property, it raises the **UnsupportedQoS** exception.

The **flush()** operation guarantees that all events sent to the log prior to the invocation of the **flush()** operation will be written to final storage medium before the **flush()** operation completes. If a log implementation does not support QoSFlush property, then invoking the **flush()** operation will raise a CORBA **UnsupportedQoS** exception.

By default, a log has QoSNone property upon creation.

An **AttributeValueChange** event is generated whenever the quality of service of a log is set.

1.2.4.11 Log Capacity Alarm Threshold

```
CapacityAlarmThresholdList get_capacity_alarm_thresholds();
void set_capacity_alarm_thresholds(in CapacityAlarmThresholdList t)
                                   raises (InvalidThreshold);
```

The **get_capacity_alarm_thresholds()** operation returns a sequence of value that specifies, as a percentage of max log size, the points at which a **ThresholdAlarm** event will be generated.

The **set_capacity_alarm_threshold()** operation sets the points at which a **ThresholdAlarm** event will be generated.

Log capacity alarm thresholds are used to warn clients when a log is approaching full. If the capacity of a log exceeds that of one of its log capacity alarm thresholds, then a **ThresholdAlarm** event is generated to indicate that the log is approaching full.

When a log object is created with the wrap option, the capacity threshold alarms are triggered as if coupled to a gauge that counts from zero to the highest capacity threshold value defined and then resets to zero.

Note – A wrapping log can be viewed as a circular buffer. The margin between the highest capacity alarm threshold and 100% can be regarded as a safety factor, to allow sufficient time for log users to retrieve log records upon receipt of a capacity alarm, before those records that entered the log after the previous capacity alarm are overwritten. Resetting the hidden gauge to zero each time the highest threshold is crossed, ensures the behavior that a capacity alarm will be generated every time the same fraction of the log capacity is written to the log, and therefore the same safety factor is maintained. In other words, every time a fixed percentage of the log capacity is written to the wrapping log, a capacity alarm is emitted.

Log objects always forward capacity threshold alarms if they have been programmed to halt when the log full condition occurs. For a log that has a log full action of halt, the user must set the capacity alarm threshold; otherwise, the log assumes a threshold set at 100%.

An **AttributeValueChange** event is generated whenever the capacity alarm threshold of a log is set.

Clients using a log that has a log full action of halt should register with the log factory to receive the **ThresholdAlarm** event in order to be informed of a log full and halt condition. New events delivered to a log in the “halt” state will not be logged, but the events may still be forwarded if the log forwarding state (described below) is “on.”

Note – BasicLogs are simple logs that are not aware of events nor are **BasicLogFactorys** capable of generating events. Therefore an implementation needs to choose a different technique to announce that a log is approaching full. Some example techniques are for the **BasicLog** to write to stderr or syslog.

1.2.4.12 Forwarding State

```
enum ForwardingState   {on, off};

ForwardingState get_forwarding_state();
void set_forwarding_state(in ForwardingState state);
```

- The **get_forwarding_state()** operation returns the forwarding state of a log.
- The **set_forwarding_state()** operation sets the forwarding state of a log.

These operations allow clients to control whether a log should forward records to any event consumers connected to the log. The following forwarding states are defined:

- **on**: The log will forward incoming events from all suppliers to all consumers currently connected.
- **off**: The log will not forward incoming events from suppliers to any consumers.

By default, the forwarding state is set to ‘on’ when a log object is created.

Log forwarding is determined solely by forwarding state, it is not affected by the log’s administrative state, availability state, log duration, scheduling, or log full action.

A **StateChange** event is generated whenever the forwarding state of a log is set.

1.2.4.13 Log Filtering

CosNotifyFilter::Filter get_filter();
void set_filter(in CosNotifyFilter::Filter filter);

NotifyLog and **TypedNotifyLog** objects apply filtering to events they log. These filters are in addition to the filters being applied on the notification channel (i.e., filters set for the notification proxy consumer and supplier admin objects). Events that pass the filters for the notification proxy consumer and the log filter will be logged.

Log filters are set and modified via the **CosNotifyFilter::Filter** object associated with a log. The default constraint language supported by the filter is the same as the one specified by the Notification Service.

The **get_filter()** operation returns the **CosNotifyFilter::Filter** object for the log.

The **set_filter()** operation sets the **CosNotifyFilter::Filter** object for the log (e.g., supply an external filter object that implements a different constraint language).

By default, the **CosNotifyFilter::Filter** object contains no filter when a log is created and all events will be logged.

An **AttributeValueChange** event is generated whenever the filter of a log is set.

1.2.5 Log Record Manipulation

The **Log** interface allows log clients to manipulate log records in the following manner:

- write records to the log
- query records from the log
- retrieve records from the log
- delete records from the log
- query/modify log record attributes

Note – It is up to the implementation to provide the locking mechanism that supports concurrent accessing of log records.

1.2.5.1 Log Record Writing

Log Records are written to the Log as follows:

1. Via the push/pull operations defined in the CosEventService and CosNotificationService.

   Since a log “is-a” EventChannel, NotificationChannel, TypedEventChannel, or TypedNotificationChannel log records are written to the log by event suppliers using either the push or pull model. The event suppliers connect themselves to a proxy consumer located within the log.

   A log may not be able to create any new log records because it is full, locked, or disabled. If a log cannot create new log records when a push operation is invoked, then the push operation should fail and a SystemException should be raised. The Telecom Log Service proposes to define the following OMG Standard Minor exception codes for use with the CosEventService and CosNotificationService push operations:

   - LOGFULL minor code

     When a push operation is invoked and a log is full, then a **NO_RESOURCE** SystemException is raised with a LOGFULL minor code.
   - LOGOFFDUTY minor code

     When a push operation is invoked on a log that is off-duty, then a **NO_RESOURCE** SystemException is raised with a LOGOFFDUTY minor code.
   - LOGLOCKED minor code

     When a push operation is invoked on a log that is locked, then a **NO_PERMISSIONS** SystemException is raised with a LOGLOCKED minor code.
   - LOGDISABLED minor code

     When a push operation is invoked on a log that is disabled, then a **TRANSIENT** SystemException is raised with a LOGDISABLED minor code.

2. Via the **write_records()** and **write_recordlist()** operations.

```
exception LogFull {short n_records_written;};
exception LogLocked {};
void write_records(in Anys records)
                raises(LogFull, LogLocked, LogDisabled);
void write_recordlist(in RecordList list)
                raises(LogFull, LogLocked, LogDisabled);
```

   Lightweight log clients or legacy applications that are “event unaware” can use the Log interface to write to the log using the **write_records()** operation. The log record is written directly to the log and is not subject to the log filter and also will not be forwarded.

   The **write_records()** operation takes a sequence of Anys as a parameter, each Any contains the event to be logged.

   The **write_recordlist()** operation is provided as a convenience operation to allow the output of a query of one log to be written to another log. In this case, only the “info” field (event content) of the RecordData struct is written to the log, the LogRecord id and logging time will be assigned by the log.

   Both the **write_records()** and **write_recordlist()** operations have the following behavior:

- If the log's availability status is "log_full" and its LogFullAction is "halt," then a **LogFull** exception is raised and the number of log records written will be returned in the exception.
- If the log's availability status is "off_duty," then a **LogOffDuty** exception is raised and no log records are written.
- If the log's administrative state is "locked," then a **LogLocked** exception is raised and no log records are written.
- If the log's operational state is "disabled," then a **LogDisabled** exception is raised and no log records are written.

1.2.5.2 Log Record Querying

Log record querying is supported based on a grammar and constraint language. The default constraint language is the one specified in Notification Service. Other query languages such as SQL and OQL could also be supported; however, they are not conformance requirements and are currently not specified.

Each log record contains the logging time, log record id (a unique number assigned by log), attributes, and the event being logged. Queries can be constructed to retrieve log records based on log time, log record id, attributes, and event contents.

RecordList query(in string grammar, in Constraint c, out Iterator i)
raises(InvalidGrammar, InvalidConstraint);

The **query()** operation searches the log for all log records that match the given constraint. The constraint parameter specifies which log records the client wishes to receive. The grammar parameter indicates how to interpret the constraint string. The default grammar is "EXTENDED_TCL" specified in the Notification Service. The log records are returned as a sequence and an iterator may be provided as an out parameter to deal with large query results. If the iterator is not needed, then the iterator will hold a nil object reference.

- An **InvalidGrammar** exception is raised if the implementation does not support the grammar specified.
- An **InvalidConstraint** exception is raised if the constraint string is invalid.

TypedRecordList typed_query(in string grammar, in Constraint c,
out TypedRecordIterator i)
raises(InvalidGrammar, InvalidConstraint);

The **typed_query()** operation searches the typed log for all log records that match the given constraint. It is similar to the **query()** operation except that the results returned are **TypedRecordList** and **TypedRecordIterator**. This allows clients to obtain **TypedLogRecords** that contain the typed information about an event (interface name, operation, argument list used for typed event communication). Exceptions are as follows:

- An **InvalidGrammar** exception is raised if the implementation does not support the grammar specified.
- An **InvalidConstraint** exception is raised if the constraint string is invalid.

unsigned long match(in string grammar, in Constraint c)
raises(InvalidGrammar, InvalidConstraint);

The **match()** operation works just like the **query()** operation except that only the number of log records that match the constraint is returned. The log records themselves are not returned.

1.2.5.3 Log Record Retrieval

RecordList retrieve(in TimeT from_time, in long how_many, out Iterator i);

The **retrieve()** operation reads the log records in the log sequentially starting from any given time. The **from_time** parameter indicates which time to start from. The **how_many** parameter indicates how many log records to retrieve, and the sign of the **how_many** parameter indicates the direction (positive for forward retrieval or negative for backward retrieval). The log records are returned as a sequence and an iterator may be provided as an out parameter to deal with large retrievals. If the iterator is not needed, then the iterator will hold a nil object reference.

TypedRecordList typed_retrieve(in TimeT from_time, in long how_many,
out TypedRecordIterator i);

The **typed_retrieve()** operation reads the typed log records in the log sequentially starting from any given time. It is similar to the retrieve operation except that the results returned are **TypedRecordList** and **TypedRecordIterator**. This allows clients to retrieve **TypedLogRecord** that contains typed information about an event.

If a client invokes either the **query()** or **retrieve()** operations (defined in **Log** interface), then the typed log implementation returns the log records as **LogRecord** structures.

If a client invokes either the **typed_query()** or **typed_retrieve()** operations, then the typed log implementation returns the log records as **TypedLogRecord** structures. The **query()** and **retrieve()** operations allow a client to get untyped and typed events in a uniform record format.

1.2.5.4 Log Record Deletion

unsigned long delete_records(in string grammar, in Constraint c)
raises(InvalidGrammar, InvalidConstraint);

The **delete_records()** operation deletes log records from the log. The constraint parameter specifies which log records the client wishes to delete. The grammar parameter indicates how to interpret the constraint string. The default grammar is “EXTENDED_TCL”as specified in the Notification Service. The return value is the number of records deleted. Exceptions are as follows:

- An **InvalidGrammar** exception is raised if the implementation does not support the grammar specified.
- An **InvalidConstraint** exception is raised if the constraint string is invalid.

- An **InvalidAttribute** exception is raised if one of the attributes is invalid.

unsigned long delete_records_by_id(in RecordIdList ids);

The **delete_records_by_id()** operation deletes specific log records from the log. This operation takes a sequence of log record ids as a parameter, and returns the number of records deleted. If an empty sequence of records is specified, then no records are deleted and the operation returns a value of 0.

1.2.5.5 Log Record Attribute Query/Modification

Log records can have attributes (see Sect i on1.2.2, "LogRecord and TypedLogRecord," on page 1-6 for more details). These attributes are readable and writable. A log record attribute is a name/value pair. It is up to clients to define name/value pairs that are meaningful to their applications. For example, a ("trouble ticket id," id) attribute associates a log record with trouble ticket information and a ("comment," text) attribute allows clients to annotate a log record.

NVList get_record_attribute(in RecordId id) raises(InvalidRecordId);

The **get_record_attribute()** operation returns the attributes of a given log record indicated by the id parameter. If the log record does not exist, then an **InvalidRecordId** exception is raised.

void set_record_attribute(in RecordId id, in NVList attr_list)
raises(InvalidRecordId, InvalidAttribute);
unsigned long set_records_attribute(in string grammar, in Constraint c,
in NVList)
raises(InvalidGrammar, InvalidConstraint, InvalidAttribute);

The **set_record_attribute()** operation sets the log record indicated by the id parameter to the attributes specified by the **attr_list** parameter. Exceptions are as follows:

- If the log record does not exist, then an **InvalidRecordId** exception is raised.
- If one of the attributes is not valid, then an **InvalidAttribute** exception is raised. The **InvalidAttribute** exception is raised only when an implementation supports some predefined attributes. The log implementation will not be able to validate client defined attributes at run time, therefore it is the client's responsibility to make sure the attributes specified are valid.
- An **InvalidGrammar** exception is raised if the implementation does not support the grammar specified.
- An **InvalidConstraint** exception is raised if the constraint string is invalid.

The **set_records_attribute()** operation sets the attributes of all the log records that match the given constraint. The constraint parameter will specify which log records the client wishes to set attributes. The grammar parameter indicates how to interpret the

constraint string. The default grammar is “EXTENDED_TCL” specified in the Notification Service. If successful, then the number of records whose attributes are set is returned. Exceptions are as follows:

- An **InvalidGrammar** exception is raised if the implementation does not support the grammar specified.
- An **InvalidConstraint** exception is raised if the constraint string is invalid.
- An **InvalidAttribute** exception is raised if one of the attributes is invalid.

1.2.5.6 The Iterator and TypedRecordIterator Interfaces

The **Iterator** interface allows clients to a log to iterate over a list of **LogRecords** returned from a **query()** or **retrieve()** operation. The elements are accessed in order.

```
interface Iterator {
    RecordList get(in unsigned long position, in unsigned long how_many)
                raises (InvalidParam);
    void destroy();
};
```

The iterator is returned in case the results of a **query()** or **retrieve()** operation are too long to fit in the synchronous response to the operation.

When invoking the **get()** operation on the iterator it returns at most **how_many** records, starting at the position indicated by the position argument. A value of zero for position indicates the first value in the complete iterator sequence.

The **get()** operation raises the **InvalidParam** exception when the position parameter is past the end of the iterator or the requested position is lower than the largest position already requested (i.e., you cannot request values before the position of the last request, so that the iterator cannot be backed up).

If the **get()** operation returns a zero length sequence, then the end of the iterator has been reached and there are no more values to be processed. Any calls to **get()** before the iterator has reached its end must return at least one **LogRecord** and at most **how_many** LogRecords. If **how_many** is supplied as 0, then the Iterator might return an arbitrary number of records (but at least one).

If a client receives a system exception (such as **IMPL_LIMIT**) when invoking the **get()** operation, then the client may reinvoke the **get()** operation providing the same position, but requesting a lower number of records. This means that the iterator must keep track of records from the last position that was retrieved (that is, records might be re-accessed, but only from the last set of returned records).

If the user wants to discard the results of an operation without exhausting the iterator, then the **destroy()** operation of the iterator should be called. Any subsequent calls using a reference to the destroyed iterator will raise the **OBJECT_NOT_EXIST** system exception.

Once the iterator has been exhausted, the implementation may dispose of the iterator object at will. It is not required to invoke the **destroy()** operation on an exhausted iterator (an iterator is exhausted once it has returned an empty sequence, that is, there are no more records to access through the iterator).

```
interface TypedRecordIterator {
    TypedRecordList get(in unsigned long position,
                        in unsigned long how_many)
                raises (InvalidParam);
    void destroy();
};
```

The **TypedRecordIterator** is similar to the **Iterator** interface except that it is used to retrieve results of **TypedRecordList**.

1.2.6 Log Network

Logs can form networks of various topology, by connecting them as event channels. A topology of DAG is normally recommended. Each log stores events it receives on the consumer end (**ConsumerAdmin** interface), and forwards events to the logs connected to it via the supplier end (the **SupplierAdmin** interface).

1.2.7 Log Lifecycle Management

Log supports the following Lifecycle related operations:

- copy
- destroy

1.2.7.1 copy operations

```
Log copy(out LogId id) raises (NoResources);
Log copy_with_id (in LogId id)  raises (NoResources, LogIdAlreadyExists);
```

The **copy()** operation creates an empty log with its attributes initialized to the same values as the log on which the operation was invoked. The log id of the created log is generated and returned as an out parameter. The log object reference is the return value. The log factory will be notified of the existence of this new log. A **NoResources** exception is raised if the log server does not have enough resources to create the log.

The **copy_with_id()** operation creates an empty log with its attributes initialized to the same values as the log on which the operation was invoked. The log id is specified as an in parameter. The log object reference is the return value. The log factory will be notified of the existence of this new log. Exceptions are as follows:

- A **LogIdAlreadyExists** exception will be raised if the log id already exists within the scope of the log factory.

- A **NoResources** exception is raised if the log server does not have enough resources to create this log.

An **ObjectCreation** event is generated whenever a log is copied.

1.2.7.2 *destroy operation*

The **destroy()** operation is inherited from **CosEventChannelAdmin::EventChannel** and disconnects any consumers or suppliers connected via its event or notification channel interface, destroys the log object and all its contained log records, and frees the associated persistent storage.

An **ObjectDeletion** event is generated whenever a log is destroyed.

1.3 *The Log Factory Interfaces*

Log factories create log instances and serve as collection managers for logs. Log factories provide operations such as find log by id and list logs. For convenience, each log has a reference to the log factory that created it. Log factories are also event or notification **ConsumerAdmin** objects, which allows clients to subscribe to log generated events emitted by logs created by the log factory. Examples of log generated events are alarm threshold, log state, attribute change events, and are not to be confused with events that are forwarded through the event or notification channel of the log.

Figure 1-4 gives a graphical representation of log factory.

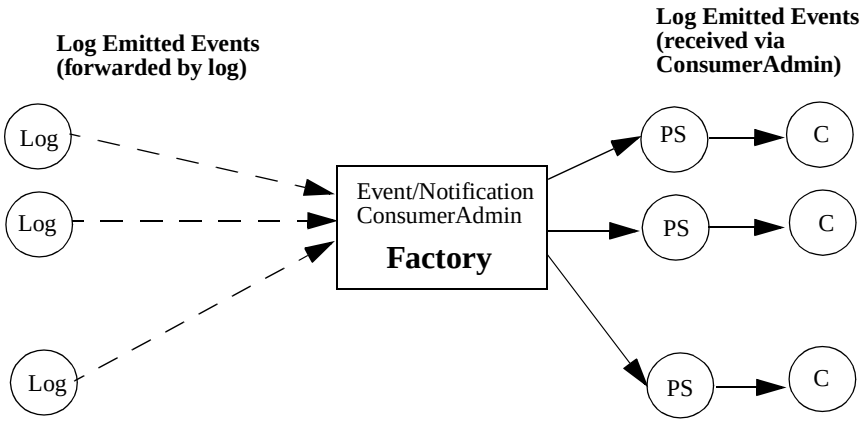

Figure 1-4 The Log Factory

Figure key:

- PS: proxy supplier
- C: event/notification consumer

1.3.1 Log Factory Inheritance

There are five different log interfaces: **BasicLog**, **EventLog**, **NotifyLog**, **TypedEventLog**, and **TypedNotifyLog**. Each log interface has its own corresponding log factory. The log factory inheritance hierarchy is represented graphically in the following figure. All of the factories will inherit from the abstract interface **LogMgr**.

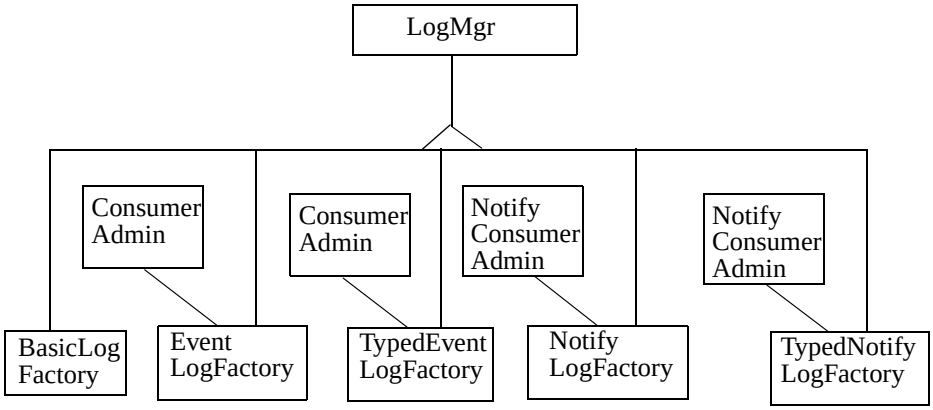

Figure 1-5 Factory Inheritance

The **LogMgr** interface serves as a collection manager for logs. **LogMgr** provides the following functions:

- list logs
- list logs by id
- look up log by id

The **BasicLogFactory** interface inherits from **LogMgr**. **BasicLogFactory** provides the following additional functions:

- create Log
- create Log with id

The **EventLogFactory** interface inherits from **LogMgr** and **CosEventChannelAdmin::ConsumerAdmin**. **EventLogFactory** provides the following additional functions:

- Create EventLog.
- Create EventLog with id.
- Emit log creation events and forward log generated events as untyped events.

The **NotifyLogFactory** interface inherits from **LogMgr** and **CosNotifyChannelAdmin::ConsumerAdmin**. **NotifyLogFactory** provides the following additional functions:

- Create NotifyLog.
- Create NotifyLog with id.
- Emit log creation events and forward log generated events as untyped events.
- Clients can specify filters via the **NotifyConsumerAdmin** interface to receive filtered events.

The **TypedEventLogFactory** interface inherits from **LogMgr** and **CosEventChannelAdmin::ConsumerAdmin**. **TypedEventLogFactory** provides the following additional functions:

- Create TypedEventLog.
- Create TypedEventLog with id.
- Emit log creation events and forward log generated events as untyped events.

The **TypedNotifyLogFactory** interface inherits from **LogMgr** and **CosNotifyChannelAdmin::ConsumerAdmin**. **TypedNotifyLogFactory** provides the following additional functions:

- Create TypedNotifyLog.
- Create TypedNotifyLog with id.
- Emit log creation events and forward log generated events as untyped events.
- Clients can specify filters via the **TypedNotifyConsumerAdmin** interface to receive filtered events.

1.3.2 Log Lookup

```
LogList list_logs();
Log find_log(in LogId id);
LogIdList list_logs_by_id();
```

The **list_logs()** lists all existing logs that have either been created through the LogMgr or are copies of those logs created through the LogMgr.

The **find_log()** operation returns a reference to the log that has the supplied log id. If the log id is not found, then a nil object reference is returned.

The **list_logs_by_id()** operation is similar to the **list_logs()** operation except that it returns a list of log ids instead of logs.

1.3.3 Log Creation

```
BasicLog create (
    in LogFullActionType full_action,
    in unsigned long max_size,
    out LogId id
    ) raises (InvalidLogFullAction);
```

The **create()** operation in the **BasicLogFactory** interface allows clients to create new **BasicLog** objects. The log full action, maximum size, and threshold list must be specified at creation time, but can be changed later. The log id will be generated and returned as an out parameter. Exceptions are as follows:

- An **InvalidLogFullAction** exception is raised if the **full_action** specified is not valid.

The **create()** operation in the **EventLogFactory**, **NotifyLogFactory**, **TypedEventLogFactory**, and **TypedNotifyLogFactory** interfaces creates new **EventLog**, **NotifyLog**, **TypedEventLog**, and **TypedNotifyLog** objects respectively. Each **create()** operation for the log factories takes arguments and raises exceptions that are appropriate for the type of log being created.

```
BasicLog create_with_id (
    in LogId id,
    in LogFullAction full_action,
    in unsigned long max_size
    )
    raises (LogIdAlreadyExists, InvalidLogFullAction, InvalidThreshold);
```

The **create_with_id()** operation in the **BasicLogFactory** interface allows clients to create new Log objects. The log full action, maximum size, and threshold list must be specified at creation time, but can be changed later. The log id is specified as an **in** parameter. Exceptions are as follows:

- An **InvalidLogFullAction** exception is raised if the **full_action** specified is not valid.
- A **LogIdAlreadyExists** exception will be raised if this log id exists within the scope of the log factory.

The **create_with_id()** operation in the **EventLogFactory**, **NotifyLogFactory**, **TypedEventLogFactory**, and **TypedNotifyLogFactory** interfaces create new **EventLog**, **NotifyLog**, **TypedEventLog**, and **TypedNotifyLog** objects respectively. Each **create_with_id()** operation for the log factories takes arguments and raises exceptions that are appropriate for the type of log being created.

An **ObjectCreation** event is generated whenever a log is created.

1.3.4 Log Events

Log factories generate log creation events when a log is successfully created. Each log also generates events and forwards them to its log factory so event consumers can receive all or filtered events from the log factory.

The following lists how event consumers can subscribe to receive log generated events from each of the log factories:

- **BasicLogFactory** is a log factory for the “event unaware” **BasicLog** and does not generate any events.

- **EventLogFactory** inherits from **CosEventChannelAdmin::ConsumerAdmin**. Event consumers can subscribe to receive events via the **ConsumerAdmin** interface. Events are untyped and unfiltered.
- **NotifyLogFactory** inherits from **CosNotifyChannelAdmin::ConsumerAdmin**. Event consumers can subscribe to receive events via the **ConsumerAdmin** interface. Events are untyped and may be filtered.
- **TypedEventLogFactory** inherits from **CosEventChannelAdmin::ConsumerAdmin**. Event consumers can subscribe to receive events via the **ConsumerAdmin** interface. Events are untyped and unfiltered.
- **TypedNotifyLogFactory** inherits from **CosNotifyChannelAdmin::ConsumerAdmin**. Event consumers can subscribe to receive events via the **NotifyConsumerAdmin** interface. Events are untyped and may be filtered.

1.4 Log Generated Events

The log factories (and their logs) generate the following types of events:

- **ObjectCreation**: Generated when a log is created.
- **ObjectDeletion**: Generated when a log is deleted.
- **ThresholdAlarm**: Generated when a log has reached its capacity alarm threshold.
- **AttributeValueChange**: Generated when a log's attribute (capacity alarm threshold, log full action, max log size, start time, stop time, week mask, filter, max record life, or quality of service) is changed.
- **StateChange**: Generated when a log's administrative or operational state is changed.

Note – Log factories are responsible for the creation, deletion, management of logs, and the generation of log events. A Log factory is not intended to be a consumer of events and therefore does not expose a **SupplierAdmin** interface. It is left up to an implementation detail as to how log generated events are passed from a log to its log factory.

1.4.1 ObjectCreation Event

The **ObjectCreation** event is generated by a log factory after it creates a log one of its logs is copied.

```
struct ObjectCreation {
    Log   logref;
    LogId id;
    TimeT time;
};
```

The **ObjectCreation** struct defines the log creation event. The **log** field indicates the object reference of the newly created log. The **id** field indicates the log identifier. The **time** field indicates the time when the log is created.

1.4.2 ObjectDeletion Event

The **ObjectDeletion** event is generated by a log factory when one of its logs is deleted.

```
struct ObjectDeletion {
    LogId id;
    TimeT time;
};
```

The **ObjectDeletion** struct defines the log deletion event. The **id** field indicates the log identifier. The **time** field indicates the time when the log is deleted.

1.4.3 ThresholdAlarm Event

The **ThresholdAlarm** event is generated by a log factory when one of its logs reaches its capacity alarm threshold to indicate that a log full or wrapping condition is approaching. If a log wraps when full, then the capacity threshold events are triggered as if coupled to a gauge that counts from zero to the highest capacity threshold value defined and then resets to zero.

```
typedef unsigned short PerceivedSeverityType;

    const PerceivedSeverityType critical = 0;
    const PerceivedSeverityType minor = 1;
    const PerceivedSeverityType cleared = 2;

struct ThresholdAlarm {
    Log           logref;
    LogId         id;
    TimeT         time;
    Threshold crossed_value;
    Threshold observed_value;
    PerceivedSeverityType  perceived_severity;
};
```

The **ThresholdAlarm** struct defines the log capacity threshold alarm. The **log** field indicates the object reference of the log. The **id** field indicates the log identifier. The **time** field indicates the time when the log has reached its capacity alarm threshold. The **observed_value** field indicates the current log size, as a percentage of the

maximum log size. The **crossed_value** field indicates the threshold level just being crossed. The **perceived_severity** field is minor if log is not full, and critical otherwise.

1.4.4 AttributeValueChange Event

The **AttributeValueChange** event is generated by a log factory when one of its logs changes one of the following log attributes:

- capacity alarm threshold
- log full action
- maximum log size
- start time
- stop time
- week mask
- adding/removing/changing a constraint expression on the log's filter object
- max record life
- quality of service

```
typedef unsigned long AttributeType;
const AttributeType capacityAlarmThreshold = 0;
const AttributeType logFullAction          = 1;
const AttributeType maxLogSize             = 2;
const AttributeType startTime              = 3;
const AttributeType stopTime               = 4;
const AttributeType weekMask               = 5;
const AttributeType filter                 = 6;
const AttributeType maxRecordLife          = 7;
const AttributeType qualityOfService       = 8;

struct AttributeValueChange {
    Log   logref;
    LogId id;
    TimeT time;
    AttributeType  type;
    any old_value;
    any new_value;
};
```

The **AttributeValueChange** struct defines the log attribute value change event. The **log** field indicates the object reference of the log. The **id** field indicates the log identifier. The **time** field indicates the time when the log's attribute is changed. The **type** field indicates the type of attribute being changed. The **old_value** field contains the old attribute value. The **new_value** field contains the new attribute value.

1.4.5 StateChange Event

The **StateChange** event is generated by a log factory if one of its log's administrative state is set or the operational state has changed.

```
typedef unsigned long StateType;
const StateType administrativeState = 0;
const StateType operationalState    = 1;
const StateType forwardingState     = 2;

struct StateChange {
    Log   logref;
    LogId id;
    TimeT time;
    StateType type;
    any new_value;
};
```

The **StateChange** struct defines the log state value change event. The **log** field indicates the object reference of the log. The **id** field indicates the log identifier. The **time** field indicates the time when the log's state is changed. The **type** field indicates the type of attribute being changed. The **new_value** field contains the new state value.

1.4.6 ProcessingErrorAlarm Event

The **ProcessingErrorAlarm** event is generated by a log factory if one of its logs has generated an error.

```
struct ProcessingErrorAlarm {
    long error_num;
    string error_string;
};
```

The **ProcessingErrorAlarm** struct defines the error generated by the log. The **error_num** field indicates the error number associated with the error. The high order 20 bits of **error_num** contain a 20-bit vendor specific error id (VSEID); the low order 12 bits contain the rest of the error number. A vendor (or group of vendors) who wish to define a specific set of error numbers should obtain a unique VSEID from the OMG, and then define a specific set of error numbers using the VSEID for the high order bits. The **error_string** field contains a textual description of the error.

1.5 Conformance Criteria

This section specifies the compliance points for this specification. In order to be in conformance with this specification a conforming implementation:

- Must support all the interfaces in the **DsLogAdmin**, **DsEventLogAdmin**, and **DsNotifyLogAdmin** modules:
  - **Log**
  - **BasicLog**
  - **EventLog**
  - **NotifyLog**
  - **Iterator**
  - **LogMgr**

  - **BasicLogFactory**
  - **EventLogFactory**
  - **NotifyLogFactory**

- May also support, in addition to the interfaces enumerated above, interfaces in the **DsTypedEventLogAdmin** and **DsTypedNotifyLogAdmin** modules:
  - **TypedEventLog**
  - **TypedNotifyLog**
  - **TypedEventLogFactory**
  - **TypedNotifyLogFactory**
- May also support log generated notifications defined in the **DsLogNotification** module:
  - **ThresholdAlarm**
  - **ObjectCreation**
  - **ObjectDeletion**
  - **AttributeValueChange**
  - **StateChange**
- Will provide implementations of the **CosNotifyFilter::Filter** interface that supports the constraints expressed in the default constraint grammar specified in the OMG Notification Service.
- Will provide implementations of the **query()**, **match()**, **delete_records()** operations of the **Log** interface that support the constraints expressed in the default constraint grammar specified in the OMG Notification Service.
- All QoS properties defined in this specification must at least be understood by any conforming implementation. However, a conforming implementation may choose not to implement all standard QoS properties and/or QoS property settings. In cases where a client requests a standard QoS property with a setting that is not supported by a conforming implementation, the implementation should raise the **UnsupportedQoS** exception.

Complete OMG IDL A

Note – Editorial changes are in this color.

A.1 DsLogAdmin Module

This module defines the **Log**, **LogMgr**, **BasicLog**, **BasicLogFactory**, and **Iterator**.

```
#include <TimeBase.idl> // CORBA Time Service
#pragma prefix “omg.org”

module DsLogAdmin
{
    exception InvalidParam {string details;};
    exception InvalidThreshold {};
    exception InvalidTime {};
    exception InvalidTimeInterval {};
    exception InvalidMask {};
    exception LogIdAlreadyExists {};
    exception InvalidGrammar{};
    exception InvalidConstraint{};
    exception LogFull {short n_records_written;};
    exception LogOffDuty {};
    exception LogLocked {};
    exception LogDisabled {};
    exception InvalidRecordId {};
    exception InvalidAttribute {string attr_name; any value;};
    exception InvalidLogFullAction {};

    typedef unsigned long LogId;
    typedef unsigned long long RecordId;
    typedef sequence<RecordId> RecordIdList;
```

```
const string default_grammar = “EXTENDED_TCL”;
typedef string Constraint;

typedef TimeBase::TimeT TimeT;


struct NVPair {
     string name;
     any    value;
};
typedef sequence<NVPair> NVList;

struct TimeInterval {
     TimeT start;
     TimeT stop;
};

struct LogRecord {
     RecordId id;
     TimeT    time;
     NVList   attr_list; // attributes, optional
     any      info;
 };
 typedef sequence<LogRecord> RecordList;
 typedef sequence<any>       Anys;

// Iterator with bulk operation support; returned as a
// result of querying the Log

interface Iterator {
     RecordList get(in unsigned long position,
                    in unsigned long how_many) raises(InvalidParam);
     void destroy();
 };

struct AvailabilityStatus {
   boolean off_duty;
   boolean log_full;
};

typedef unsigned short LogFullActionType;

const LogFullActionType wrap = 0;
const LogFullActionType halt = 1;

struct Time24 {
     unsigned short hour;// 0-23
     unsigned short minute;// 0-59
};

struct Time24Interval {
```

```
        Time24 start;
        Time24 stop;
};
typedef sequence<Time24Interval> IntervalsOfDay;

const unsigned short Sunday    =  1;
const unsigned short Monday    =  2;
const unsigned short Tuesday   =  4;
const unsigned short Wednesday =  8;
const unsigned short Thursday  = 16;
const unsigned short Friday    = 32;
const unsigned short Saturday  = 64;

typedef unsigned short DaysOfWeek;// Bit mask of week days

struct WeekMaskItem {
        DaysOfWeek     days;
        IntervalsOfDay intervals;
};
typedef sequence<WeekMaskItem> WeekMask;

typedef unsigned short Threshold; // 0-100 %
typedef sequence<Threshold> CapacityAlarmThresholdList;

interface LogMgr;

enum OperationalState    { disabled, enabled };
enum AdministrativeState { locked, unlocked }; // logging on/off
enum ForwardingState     { on, off };

typedef unsigned short QoSType;
typedef sequence<QoSType> QoSList;
exception UnsupportedQoS { QoSList denied; };

const QoSType QoSNone        = 0;
const QoSType QoSFlush = 1;
const QoSType QoSReliability   = 2;

interface Log
{
        LogMgr my_factory();

        LogId id();

        QoSList get_log_qos();
        void set_log_qos(in QoSList qos) raises(UnsupportedQoS);

        // life in seconds (0 infinite)
        unsigned long get_max_record_life();
        void set_max_record_life(in unsigned long life);
```

```
// size in octets
unsigned long long get_max_size();
void set_max_size(in unsigned long long size) raises (InvalidParam);

unsigned long long get_current_size(); // size in octets
unsigned long long get_n_records();    // number of records

LogFullActionType get_log_full_action();
void set_log_full_action(in LogFullActionType action)
    raises(InvalidLogFullAction);

AdministrativeState get_administrative_state();
void set_administrative_state(in AdministrativeState state);

ForwardingState get_forwarding_state();
void set_forwarding_state(in ForwardingState state);

OperationalState get_operational_state();

// log duration
TimeInterval get_interval();
void set_interval(in TimeInterval interval)
    raises (InvalidTime, InvalidTimeInterval);

// availability status
AvailabilityStatus get_availability_status();

// capacity alarm threshold
CapacityAlarmThresholdList get_capacity_alarm_thresholds();
void set_capacity_alarm_thresholds(in CapacityAlarmThreshold
                                        List threshs)
    raises (InvalidThreshold);

// weekly scheduling
WeekMask get_week_mask();
void set_week_mask(in WeekMask masks)
    raises (InvalidTime, InvalidTimeInterval, InvalidMask);

RecordList query(in  string grammar,
                 in  Constraint c,
                 out Iterator i)
    raises(InvalidGrammar, InvalidConstraint);

// negative how_many indicates backwards retrieval
RecordList retrieve(in TimeT from_time,
                 in long how_many,
                 out Iterator i);

// returns number of records matching constraint
unsigned long match(in  string grammar,
                 in  Constraint c)
```

```
            raises(InvalidGrammar, InvalidConstraint);

        // returns number of records deleted
        unsigned long delete_records(in  string grammar,
                    in  Constraint c)
            raises(InvalidGrammar, InvalidConstraint);

        unsigned long delete_records_by_id(in RecordIdList ids);

        void write_records(in Anys records)
            raises(LogFull, LogOffDuty, LogLocked, LogDisabled);
        void write_records(in Anys records)
            raises(LogFull, LogOffDuty, LogLocked, LogDisabled);

        // set single record attributes
        void set_record_attribute(in RecordId id,
            in NVList      attr_list)
                raises(InvalidRecordId, InvalidAttribute);

        // set all records that matches the constraints with same attr_list
        // returns number of records whose attributes have been set
        unsigned long set_records_attribute(in string grammar,
                    in Constraint c,
                    in NVList      attr_list)
            raises(InvalidGrammar, InvalidConstraint, InvalidAttribute);

        // get record attributes
        NVList get_record_attribute(in RecordId id)
            raises(InvalidRecordId);

        Log copy(out LogId id);
        Log copy_with_id (in LogId id) raises (LogIdAlreadyExists);

        void flush() raises (UnsupportedQoS);
    };

    interface BasicLog : Log {
        void destroy();
    };

    typedef sequence<Log> LogList;
    typedef sequence<LogId> LogIdList;

    interface LogMgr
    {
        LogList list_logs();
        Log find_log(in LogId id);
        LogIdList list_logs_by_id();
    };

    interface BasicLogFactory : LogMgr
```

```
    {
        BasicLog create (
            in LogFullActionType full_action,
            in unsigned long long max_size,
            out LogId id)
            raises (InvalidLogFullAction);

        BasicLog create_with_id
      (
            in LogId id,
            in LogFullActionType full_action,
            in unsigned long long max_size,
      )
            raises (LogIdAlreadyExists, InvalidLogFullAction);
    };

};
```

A.2 DsLogNotification Module

This module defines log generated notifications.

```
#include <DsLogAdmin.idl>
#pragma prefix "omg.org"

module DsLogNotification {

    typedef DsLogAdmin::Log             Log;
    typedef DsLogAdmin::LogId           LogId;
    typedef DsLogAdmin::Threshold       Threshold;
    typedef TimeBase::TimeT             TimeT;

    // definition of ThresholdAlarm, the event generated by Log when
    // Log reaches its capacity alarm threshold

    typedef unsigned short PerceivedSeverityType;

    const PerceivedSeverityType critical = 0;
    const PerceivedSeverityType minor = 1;
    const PerceivedSeverityType cleared = 2;

    struct ThresholdAlarm {
        Log         logref;
        LogId       id;
        TimeT       time;
        Threshold crossed_value;  // the threshold level just being crossed
        Threshold observed_value; // the current percentage
        PerceivedSeverityType perceived_severity;
    };
```

```
// the events generated by
// Log when a Log object is created or deleted

struct ObjectCreation {
    LogId    id;
    TimeT    time;
};

// NOTE: cannot say “typedef ObjectCreation ObjectDeletion because
// type would be lost in current C++ mapping for Anys.

struct ObjectDeletion {
    LogId        id;
    TimeT        time;
};

// definition of AttributeValueChange notification, the event generated by
// Log when a Log’s attribute has changed

typedef unsigned short AttributeType;
const AttributeType capacityAlarmThreshold = 0;
const AttributeType logFullAction          = 1;
const AttributeType maxLogSize             = 2;
const AttributeType startTime              = 3;
const AttributeType stopTime               = 4;
const AttributeType weekMask               = 5;
const AttributeType filter                 = 6;
const AttributeType maxRecordLife          = 7;
const AttributeType qualityOfService       = 8;

struct AttributeValueChange {
    Log          logref;
    LogId        id;
    TimeT        time;
        AttributeType type;
        any old_value;
        any new_value;
};

// definition of StateChange notification, the event generated by
// Log when a Log’s state has changed

typedef unsigned short StateType;
const StateType administrativeState   = 0;
const StateType operationalState      = 1;
const StateType forwardingState       = 2;

struct StateChange {
    Log   logref;
    LogId id;
    TimeT time;
```

```
        StateType type;
        any new_value;
    };

    struct ProcessingErrorAlarm  {
        // Event generated by a log when a problem occurs within the log.
        // The highest 20 bits of error_num are reserved for vender
        // specific Ids.
        long error_num;
        string error_string;
    };
};
```

A.3 DsEventLogAdmin Module

This module defines the **EventLog** and **EventLogFactory** interfaces.

```
#include <CosEventChannelAdmin.idl>// CORBA Event Service
#include <DsLogAdmin.idl>
#pragma prefix "omg.org"

module DsEventLogAdmin
{
    interface EventLog : DsLogAdmin::Log,
            CosEventChannelAdmin::EventChannel {};

    interface EventLogFactory : DsLogAdmin::LogMgr,
            CosEventChannelAdmin::ConsumerAdmin
    {
        EventLog create (
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds,
            out DsLogAdmin::LogId id
        ) raises (DsLogAdmin::InvalidLogFullAction,
                    DsLogAdmin::InvalidThreshold);

        EventLog create_with_id (
            in DsLogAdmin::LogId id,
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds
        ) raises (DsLogAdmin::LogIdAlreadyExists,
                DsLogAdmin::InvalidLogFullAction,
                DsLogAdmin::InvalidThreshold);
    };

};
```

A.4 DsNotifyLogAdmin Module

This module defines the **NotifyLog** and **NotifyLogFactory** interfaces.

```
#include <DsEventLogAdmin.idl>
#include <CosNotifyChannelAdmin.idl>
#include <CosNotifyFilter.idl>
#include <CosNotification.idl>
#pragma prefix “omg.org”

module DsNotifyLogAdmin
{
    interface NotifyLog :
        DsEventLogAdmin::EventLog,
        CosNotifyChannelAdmin::EventChannel
    {
        CosNotifyFilter::Filter get_filter();
        void set_filter(in CosNotifyFilter::Filter filter);
    };

    interface NotifyLogFactory : DsLogAdmin::LogMgr,
            CosNotifyChannelAdmin::ConsumerAdmin
    {
        NotifyLog create (
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds,
            in CosNotification::QoSProperties initial_qos,
            in CosNotification::AdminProperties initial_admin,
            out DsLogAdmin::LogId id
        ) raises (DsLogAdmin::InvalidLogFullAction,
                    DsLogAdmin::InvalidThreshold,
                    CosNotification::UnsupportedQoS,
                    CosNotification::UnsupportedAdmin);

        NotifyLog create_with_id (
            in DsLogAdmin::LogId id,
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds,
            in CosNotification::QoSProperties initial_qos,
            in CosNotification::AdminProperties initial_admin
        ) raises (DsLogAdmin::LogIdAlreadyExists,
                DsLogAdmin::InvalidLogFullAction,
                DsLogAdmin::InvalidThreshold,
                CosNotification::UnsupportedQoS,
                CosNotification::UnsupportedAdmin);
    };
};
```

A.5 DsTypedEventLogAdmin Module

This module defines the **TypedEventLog** and **TypedEventLogFactory** interfaces.

```
#include <orb.idl>
#include <CosEventChannelAdmin.idl>
#include <CosTypedEventChannelAdmin.idl>
#include <DsLogAdmin.idl>
#pragma prefix “omg.org”

module DsTypedEventLogAdmin
{
    typedef sequence<any> ArgumentList;
    struct TypedLogRecord {
        DsLogAdmin::RecordId    id;
        DsLogAdmin::TimeT       time;
        DsLogAdmin::NVList      attr_list;
        CORBA::RepositoryId     interface_id; // repository id of the
                                               interface for
                                                        // sending typed event
        CORBA::Identifier  operation_name;    // operation name
        ArgumentList       arg_list;          // argument list, contains
                                                    event data
    };
    typedef sequence<TypedLogRecord> TypedRecordList;

    interface TypedRecordIterator {
        TypedRecordList get(in unsigned long position,
                                in unsigned long how_many)
            raises(DsLogAdmin::InvalidParam);
        void destroy();
    };

    interface TypedEventLog : DsLogAdmin::Log,
            CosTypedEventChannelAdmin::TypedEventChannel
    {
        // typed record query
        TypedRecordList typed_query(in  string grammar,
                    in  DsLogAdmin::Constraint c,
                    out TypedRecordIterator i)
            raises(DsLogAdmin::InvalidGrammar,
                DsLogAdmin::InvalidConstraint);

        // typed record retrieval
        TypedRecordList typed_retrieve(in DsLogAdmin::TimeT from_time,
                    in long how_many,
                    out TypedRecordIterator i);
    };

    interface TypedEventLogFactory : DsLogAdmin::LogMgr,
```

```
                    CosEventChannelAdmin::ConsumerAdmin
    {
    TypedEventLog create (
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds,
            out DsLogAdmin::LogId id
    ) raises (DsLogAdmin::InvalidLogFullAction,
                DsLogAdmin::InvalidThreshold);

    TypedEventLog create_with_id (
            in DsLogAdmin::LogId id,
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds
    ) raises (DsLogAdmin::LogIdAlreadyExists,
                DsLogAdmin::InvalidLogFullAction,
                DsLogAdmin::InvalidThreshold);
    };
};
```

A.6 DsTypedNotifyLogAdmin Module

This module defines the **TypedNotifyLog** and **TypedNotifyLogFactory** interfaces.

```
#include <CosTypedNotifyChannelAdmin.idl>
#include <DsTypedEventLogAdmin.idl>
#include <CosNotifyFilter.idl>
#include <CosNotification.idl>
#pragma prefix “omg.org”

module DsTypedNotifyLogAdmin
{
    interface TypedNotifyLog : DsTypedEventLogAdmin::TypedEventLog,
                CosTypedNotifyChannelAdmin::TypedEventChannel
    {
        CosNotifyFilter::Filter get_Ty filter();
        void set_filter(in CosNotifyFilter::Filter filter);
    };

    interface TypedNotifyLogFactory : DsLogAdmin::LogMgr,
                CosNotifyChannelAdmin::ConsumerAdmin
    {
        TypedNotifyLog create (
            in DsLogAdmin::LogFullActionType full_action,
            in unsigned long long max_size,
            in DsLogAdmin::CapacityAlarmThresholdList thresholds,
            in CosNotification::QoSProperties initial_qos,
            in CosNotification::AdminProperties initial_admin,
            out DsLogAdmin::LogId id
```

```
            ) raises (DsLogAdmin::InvalidLogFullAction,
                      DsLogAdmin::InvalidThreshold,
                      CosNotification::UnsupportedQoS,
                      CosNotification::UnsupportedAdmin);

            TypedNotifyLog create_with_id (
                in DsLogAdmin::LogId id,
                in DsLogAdmin::LogFullActionType full_action,
                in unsigned long long max_size,
                in DsLogAdmin::CapacityAlarmThresholdList thresholds,
                in CosNotification::QoSProperties initial_qos,
                in CosNotification::AdminProperties initial_admin
            ) raises (DsLogAdmin::LogIdAlreadyExists,
                      DsLogAdmin::InvalidLogFullAction,
                      DsLogAdmin::InvalidThreshold,
                      CosNotification::UnsupportedQoS,
                      CosNotification::UnsupportedAdmin);
        };
    };
```

Common Mistakes *B*

The Telecom Log Service has some very powerful and flexible features. Please note that it is very easy to get into trouble if you are not careful. Here are some common mistakes that should be kept in mind when using this service.

release vs. destroy

Log clients will have object references to log objects. When the clients are done using the log they should release this object reference using the language mapping release mechanism.

Note – Do not invoke the **destroy()** operation to destroy the object reference. This will destroy the actual object implementation on the server, as well as all the records that were stored in that log.

---

delete_records() constraint

Constraints should be specified carefully since there is no way to undo this operation. A malformed constraint could end up deleting far more records than intended. To be certain of the constraint, it would be wise to first do either a **query()** or a **match()** to give an indication of how the constraint will be executed. Alternatively, if the IDs for the records are known the **delete_records_by_id()** operation should be used.

max record lifetime

This feature was intended to automatically trim the log as the records age. Keep in mind that records are deleted unilaterally as they age past the maximum lifetime. If the maximum lifetime is set to one second, then the entire log will be deleted after only one second.

max size

If the maximum size of the log is set too small, the log will fill up quickly and incoming log records will be lost.

bad filter

If a filter is specified incorrectly (either on an incoming proxy consumer or on the log filter itself), then events may be dropped inappropriately.

scheduling

This feature is designed to allow the execution of very elaborate scheduling mechanisms. If not careful, the log may end up being scheduled "off" for more time than anticipated.

lock and forget

Logging can be turned off using the administrative state. Just remember to turn it back on again when appropriate.

off forwarding and forget

Event forwarding can be turned off using the forwarding state. Just remember to turn it back on again when appropriate.

factory object the only "private" object

Note that once the factory object (or any of the log objects) is published, a misguided client may traverse the entire log hierarchy and perform the above deeds either maliciously or by accident. Since the very nature of the event service is to decouple communication between applications, it is unlikely that each application will understand just how the other applications are using the log service. The CORBA security service may be necessary to enforce some restrictions on operations.

Index

Index

Telecom Log Service, v1.1.2
Reference Sheet

This is an editorial update to the Telecom Log Service, v1.1.1 specification. See Chapter 1 and Appendix A for the editorial changes.