Borland VisiBroker™ 8.0 VisiTelcoLog Guide

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

VB 80 VisiTelcoLog Guide
April 2007

Borland®

Contents

1

VisiTelcoLog Service overview

The VisiTelcoLog Service is Borland's OMG compliant implementation of the OMG Telecom Log Service specification version 1.1.2. It supports all of the features defined by the OMG specification, including all operations of the log interfaces, their factories, and their detailed semantics. This document is a user guide for the VisiTelcoLog Service, and it assumes that the reader is familiar with the OMG Telecom Log Service specification.

The essential purpose of the VisiTelcoLog Service is to transparently log events passing through a channel of an event or a notification service. VisiTelcoLog Service is typically used by mission-critical distributed monitor control applications, such as a telecommunication management network (TMN). These applications not only require a high performance event or notification service to forward events with a negligible overhead, but also require the ability to log a portion or all of these events efficiently and transparently. Though the specification is called OMG Telecom Log Service and the Borland implementation is called VisiTelcoLog Service, the architecture itself is very generic and can be used by any application.

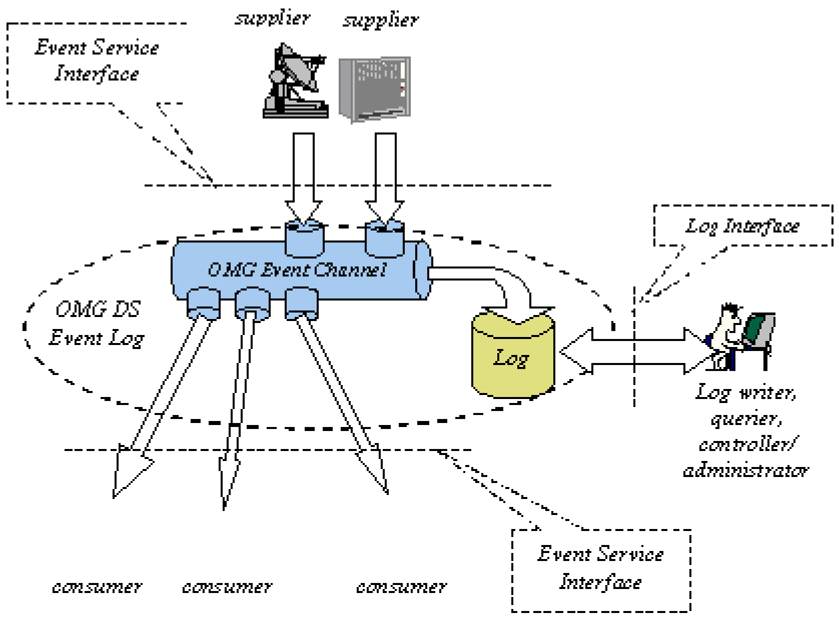
The VisiTelcoLog Service provides a high level event-logging model to shield applications from the details of event logging. This allows higher performance and application-generic log services to be implemented by third parties. It is possible for applications to implement and connect an event consumer to log transparently all received events into a conventional database or other form of external persistent repository without using the VisiTelcoLog Service, but the disadvantage of this kind of custom-built event logging at the application level is that it forces the application developer to implement a full event unmarshalling as well as application-specific record schema and events-to-records translation code. The consequences would be poor performance (namely, event throughput) and high development and maintenance costs.

With VisiTelcoLog Service, events received by an event or notification channel can be logged transparently at the application level. An event-logging object (also referred to in this document as *DsEventLog* object, or an event-based log object) is also a conventional OMG event channel (in other words, it extends from OMG event channel). This allows applications to be designed and developed without depending on whether or how events are to be logged. Existing event-based applications can also utilize the event logging of VisiTelcoLog Service with neither application code change nor redeployment.

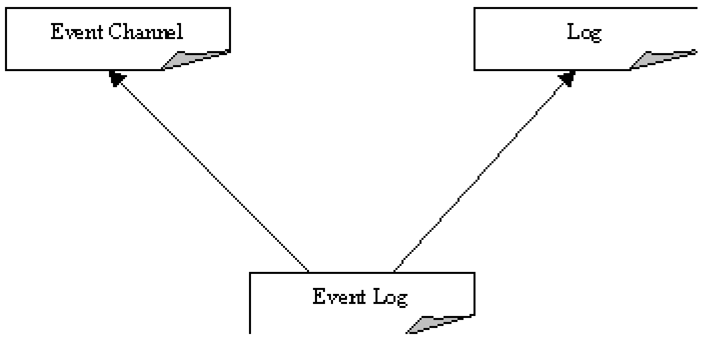
Besides the transparency for event and notification-based applications, DsEventLog is also extended from the Log object. On this log object, explicit non-event record logging, as well as log record querying, updating, deleting, log object control and administration operations can be performed. A DsEventLog object is simply extended from a conventional event channel and the log object.

For every kind of OMG defined event channel, such as event channel, typed event channel, notification channel, and typed notification channel, there is a corresponding log object. For applications that are not event-aware, a BasicLog object is also provided.

Architecture and interface inheritance views of VisiTelcoLog Service's EventLog are illustrated in the following figures. The first figure shows how an event supplier can log its events while at the same time forwarding events to all the consumers. Using the Log interface another user can also query the logged events.



The following figure describes an event-based log object's hierarchy.



2

Logging for event aware applications

This chapter discuses how an event or notification service-based application (or any *event aware* application in general) can use VisiTelcoLog Service to log events. VisiTelcoLog Service is basically an event logger. *Log*, in this context, is an event channel that propagates events apart from logging the events to a persistent store.

There are four kinds of event-based log objects that an event-aware application can use:

- EventLog
- NotifyLog
- TypedEventLog
- TypedNotifyLog

The following table describes the VisiTelcoLog Service module and interface names and the features available for event and notification service-based applications.

Feataures	OMG Event Service application	OMG Notification Service application
Module name	DsEventLogAdmin	DsNotifyLogAdmin
Factory interface name	EventLogFactory	NotifyLogFactory
Log interface name	EventLog	NotifyLog
Factory service name	EventLogService	NotifyLogService
Typed Events Module name	DsTypedEventLogAdmin	DsTypedNotifyLogAdmin
Typed Events Factory Interface name	TypedEventLogFactory	TypedNotifyLogFactory
Typed Event Log Interface name	TypedEventLog	TypedNotifyLog
Typed Events Factory service name	TypedEventLogService	TypedNotifyLogService
Log forwarding	Yes	Yes
Filtering while log forwarding	No	Yes
Filtering while storing	No	Yes

In this chapter, the following topics will be explained:

- Using log factories to obtain event based log objects
- Logging events on event based log objects
- Forwarding logged events to consumers

– Filtering events to be logged

Using log factories

For an event aware application that wishes to log events, an event-based log is first bootstrapped using the log's factory. For example, a notification service-based application first resolves to `NotifyLogFactory` using the object name `NotifyLogService`, and then obtains a log of type `NotifyLog`. For other types of event-based applications, see the table above. This section explains the steps to be taken to obtain reference to an event-based log object.

The example code below first bootstraps to `NotifyLogFactory` using the object name `NotifyLogService`. It then attempts to find a `NotifyLog` log with ID equal to 100 from the factory. If it does not find `NotifyLog` it attempts to create one. The maximum size specified is 0 (zero). This means that no predefined limit is used; however, a predefined limit is recommended.

Note

Example code is located in the <install_dir>/examples/vbroker/telcolog/primitive_cpp directory.

C++

```
// get service reference
CORBA::Object_var service =
  orb->resolve_initial_references("NotifyLogService");

DsNotifyLogAdmin::NotifyLogFactory_var factory =
  DsNotifyLogAdmin::NotifyLogFactory::_narrow(service);

// find log with id 100
DsLogAdmin::LogId id = 100;
DsLogAdmin::Log_var log = factory->find_log(id);

// if log not created, create log
if( log.in() ==  NULL )
{
  CORBA::ULongLong max_size = 4 * 1024 * 1024;
  DsLogAdmin::CapacityAlarmThresholdList thresholds;
  CosNotification::QoSProperties initial_qos;
  CosNotification::AdminProperties initial_admin;

  log = factory->create_with_id(id, DsLogAdmin::wrap,
    max_size, thresholds, initial_qos, initial_admin);
}

DsNotifyLogAdmin::NotifyLog_var notify_log=
  DsNotifyLogAdmin::NotifyLog::_narrow(log.in());
```

Note

Example code is located in the <install_dir>/examples/vbroker/telcolog/primitive_java directory.

Java

```
// get service reference
org.omg.CORBA.Object service =
  orb.resolve_initial_references("NotifyLogService");



org.omg.DsNotifyLogAdmin.NotifyLogFactory factory =
  org.omg.DsNotifyLogAdmin.NotifyLogFactoryHelper.narrow(
  service);
```

```
// find log with id 100
int id = 100;
org.omg.DsLogAdmin.Log log = factory.find_log(id);

// if log not created, create log
if( log == null )
{
  long max_size = 4 * 1024 * 1024;
  log = factory.create_with_id(id,
    org.omg.DsLogAdmin.wrap.value, max_size, new short[0],
    new org.omg.CosNotification.Property[0],
    new org.omg.CosNotification.Property[0]);
}

org.omg.DsNotifyLogAdmin.NotifyLog notify_log =
  org.omg.DsNotifyLogAdmin.NotifyLogHelper.narrow(log);
```

Logging events

Once the reference to the event-based log object is resolved, an event propagation (or forwarding) operation such as push or pull is used to propagate events. Since this channel object also has the characteristics of a log, it logs all the events that are propagated through it. Filters can also be attached to the log. See Log filtering for further details on how to selectively log events.

Furthermore, notification-based applications can use all the notification service features such as QoS framework, Event Filters, and others.

For further details on developing Notification Service supplier applications, see Developing supplier and consumer applications in the VisiBroker *VisiNotify Guide*.

VisiTelcoLog Service optimizes the event logging at the GIOP level.

On a *log full* condition, if the log full action is set to *wrap*, then the oldest events are over-written. If the log full action is set to *halt*, and if the log record expire time is specified, then all the expired events are over-written. Otherwise the following exceptions are thrown:

- **Insufficient space:** If the log space is not sufficient for logging the event then a `NO_RESOURCE` system exception with `LOGFULL` minor code (1001) is thrown.
- **Off-duty log:** If the log is off-duty then a `NO_RESOURCE` system exception with minor code `LOGOFFDUTY` (1000) is thrown.
- **Locked log:** If the log is locked then a `NO_PERMISSION` system exception with minor code `LOGLOCKED` (1003) is thrown.
- **Disabled log:** If the log is disabled, then `TRANSIENT` system exception with minor code equal to `LOGDISABLED` (1002) is thrown.

Note that if the supplier is using event batching the exceptions will not reach the supplier. See VisiBroker Event Buffering/Batch in the VisiBroker *VisiNotify Guide* for further details on event batching.

Also note that for the connected pull suppliers, the channel pulls the events and then logs those events. On a log full condition, the channel continually attempts to log until log space is available. There is no way the connected supplier application can know about this condition. Using the `vbroker.dslog.waitForLogAvailable` property a wait period can be specified for this loop. By default it is 20 seconds.

The following code sample shows a structured supplier logging TMN QoS Alarm event. The supplier application first obtains the default supplier admin from the log (as the log is also a channel in itself), and then after obtaining structured proxy push consumer, connects to it. It then creates a TMN QoS Alarm event and pushes the event through the log. When the event is pushed in the log, the log stores the event and then forwards the event based on the log's forwarding state.

Note

Example code is located in the <install_dir>/examples/vbroker/telcolog/primitive_cpp directory.

C++

```
// get default supplier admin object from the log
CosNotifyChannelAdmin::SupplierAdmin_var admin =
  notify_log->default_supplier_admin();

CosNotifyChannelAdmin::ProxyID  proxy_id;

// create a proxy consumer on the log
CosNotifyChannelAdmin::ProxyConsumer_var proxy =
  admin->obtain_notification_push_consumer(
  CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);

CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
  Consumer =
CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(proxy);

// connect to the proxy consumer
consumer->connect_structured_push_supplier(NULL);

// fill a structured event with TMN QoS Alarm event
TMN::Event event;
CosNotification::StructuredEvent structured;
TMN::QoSAlarmInfo qosalrm_info;
misc::forge_qosAlrmInfo(qosalrm_info);
event.name = (const char*)
  " TMN::Events::qosAlarm";
event.info <<= qosalrm_info;
misc::gathering(event, structured);

// push the structured event into log
consumer->push_structured_event(structured);
```

Note

Example code is located in the <install_dir>/examples/vbroker/telcolog/primitive_java directory.

Java

```
// get default supplier admin object from the log
org.omg.CosNotifyChannelAdmin.SupplierAdmin admin
  = notify_log.default_supplier_admin();

org.omg.CORBA.IntHolder proxy_id =
  new org.omg.CORBA.IntHolder();

// create a proxy consumer on the log
org.omg.CosNotifyChannelAdmin.ProxyConsumer proxy =
  admin.obtain_notification_push_consumer(
    org.omg.CosNotifyChannelAdmin.ClientType.STRUCTURED_EVENT,
    proxy_id);

org.omg.CosNotifyChannelAdmin.StructuredProxyPushConsumer
  consumer =
    org.omg.CosNotifyChannelAdmin.StructuredProxyPushConsumerHelper.narrow(
      proxy);

// connect to the proxy consumer
consumer.connect_structured_push_supplier(null);
```

```
// fill a structured event with TMN QoS Alarm event
TMN.Event event = new TMN.Event();
org.omg.CosNotification.StructuredEvent structured =
  new org.omg.CosNotification.StructuredEvent();
TMN.QoSAlarmInfo qosalrm_info = new TMN.QoSAlarmInfo();
event.header = new TMN.EventHeader();
event.info = orb.create_any();
Util.forge_event_header(event.header);
Util.forge_qosAlrmInfo(qosalrm_info);
event.name = "TMN::Events::qosAlarm";
TMN.QoSAlarmInfoHelper.insert(event.info,qosalrm_info);
Util.gathering(event, structured);

// push the structured event into log
consumer.push_structured_event(structured);
```

Forwarding logged events

The events that get pushed into a log or pulled by the log are forwarded to any down-stream consumers after the events are logged. Any consumer application can start consuming events that are propagated. See Developing supplier and consumer applications in the VisiBroker *VisiNotify Guide* for information on writing consumer applications.

By setting its forwarding state to *off*, the log object can be configured so that it does not forward logged events. The following code snippet shows how an application can disable forwarding on a `NotifyLog` object and check the current forwarding state of the log.

All the features of an event service and a notification service can be used for event propagation such as attaching filters, QoS, etc.

C++

```
notify_log->set_forwarding_state(DsLogAdmin::off);

DsLogAdmin::ForwardingState current_state =
  notify_log->get_forwarding_state();
```

Java

```
notify_log.set_forwarding_state(
  org.omg.DsLogAdmin.ForwardingState.off);

org.omg.DsLogAdmin::ForwardingState current_state =
  notify_log.get_forwarding_state();
```

Filtering events

A filter set for a `NotifyLog` or a `TypedNotifyLog` can also filter events being logged to the log. The log uses the filter object defined by the notification service, `CosNotifyFilter::Filter`. See Setting the Quality of Service and Filters in the VisiBroker *VisiNotify Guide* for information about how to create a filter and how to write constraints.

Note that only one filter object can be associated with a log. By default, no filter objects are associated with the log and all events are logged. Also, whenever a `set_filter()` method is called the log will generate an `AttributeValueChange` event.

The following example shows how to create a filter, set a filter on the log, and get a filter from the log.

C++

```
// MAKE USE OF FILTERS
// STEP 1) Get default filter factory
```

```
CosNotifyFilter::FilterFactory_var ffact =
  log->default_filter_factory();

// STEP 2) Create filter
CosNotifyFilter::Filter_var filter1;
filter1 = ffact->create_filter("EXTENDED_TCL");

// STEP 3) Create constraint
CosNotifyFilter::ConstraintExpSeq constr_seq1;
constr_seq1.length(1);
constr_seq1[0].constraint_expr = CORBA::string_dup(
  "$type_name == 'TMN::Events::qosAlarm'"
);

// STEP 4) Add constraint to filter
filter1->add_constraints( constr_seq1 );

// STEP 5) Set filter on the log
log->set_filter( filter1 );

// STEP 6) Get the filter associated with the log
CosNotifyFilter::Filter_var filter2;
Filter2 = log->get_filter();
```

Java

```
//Make Use of Filters
//[1] Get a filter factory
org.omg.CosNotifyFilter.FilterFactory ffact =
  channel.default_filter_factory();
//[2] Create a filter
org.omg.CosNotifyFilter.Filter filter = null;
filter = ffact.create_filter("EXTENDED_TCL");

//[3] Create a constraint
org.omg.CosNotifyFilter.ConstraintExp [] constraints =
  new org.omg.CosNotifyFilter.ConstraintExp[1];
constraints [0] =
  new org.omg.CosNotifyFilter.ConstraintExp();
constraints [0].constraint_expr =
  new String ("$type_name == 'TMN::Events::qosAlarm'");

//[4] Add constraint to filter
org.omg.CosNotifyFilter.ConstraintInfo[] info = null;
info = filter.add_constraints(constraints);

//[5] Set filter on the log
log.set_filter (filter);

//[6] Get the filter associated with the log
org.omg.CosNotifyFilter.Filter filter2 = null;
filter2 = log.get_filter();
```

3

Logging for event unaware applications

Legacy applications and *event unaware* clients can also use the VisiTelcoLog Service. Using the `BasicLog` interface and explicit write operations using CORBA `Any`, an event unaware application can make use of the VisiTelcoLog Service. These applications, however, will not be able to use features such as log filtering, forwarding, and event generation.
The following table describes the VisiTelcoLog Service module and interface names and the log features available for event unaware applications.

Features	Event unaware application
Module name	DsLogAdmin
Factory Interface name	BasicLogFactory
Log Interface name	BasicLog
Factory service name	BasicLogService
Log forwarding	No
Filtering while log forwarding	No
Filtering while storing	No

In this chapter, the following topics will be explained:

- Using the log factory to obtain the log object for event unaware applications
- Writing log records for event unaware applications

Using the log factory

In order to log, an event unaware application needs to get a reference to the `BasicLog` from its factory, `BasicLogFactory`. Apart from creating the basic log object, the factory interface also supports some other basic management operations such as find and list.

Resolving the `BasicLogService` name gets the `BasicLogFactory` object reference. In the following code snippet, the application looks for a `BasicLog` with an ID equal to 100, and if it does not find one a `BasicLog` is created with size equal to 0 (zero). A size equal to

0 (zero) means that there is no predefined size limit. Note that by setting log size to zero, the log continues to expand till all the disk space is used. Specifying a more meaningful value is recommended.

C++

```
// get service reference
CORBA::Object_var service =
  orb->resolve_initial_references("BasicLogService");

DsLogAdmin::BasicLogFactory_var factory =
  DsLogAdmin::BasicLogFactory::_narrow(service);

// find log with id 100
DsLogAdmin::LogId id = 100;
DsLogAdmin::Log_var log = factory->find_log(id);

// if log not created, create log
if( log.in() ==  NULL )
{
  CORBA::ULongLong max_size = 4 * 1024 * 1024;
  // max_size=0 (zero) leaves the max log size unbounded.

  log = factory->create_with_id(id, DsLogAdmin::wrap,
    max_size);
}
```

Java

```
DsLogAdmin::BasicLog_var basic_log=
  DsLogAdmin::BasicLog::_narrow(log.in());
// get service reference
org.omg.CORBA.Object service =
  orb.resolve_initial_references("BasicLogService");

org.omg.DsLogAdmin.BasicLogFactory factory =
  org.omg.DsLogAdmin.BasicLogFactoryHelper.narrow(
  service);

// find log with id 100
int id = 100;
org.omg.DsLogAdmin.Log log = factory.find_log(id);

// if log not created, create log
if( log == null )
{
  long max_size = 4 * 1024 * 1024;
  // max_size=0 (zero) leaves the max log size unbounded.
log = factory.create_with_id(id,
    org.omg.DsLogAdmin.wrap.value, max_size);
}

org.omg.DsLogAdmin.BasicLog basic_log =
  org.omg.DsLogAdmin.BasicLogHelper.narrow(log);
```

Writing log records

The `write_records` operation is used to write records to logs. The input parameter for this operation is a sequence of CORBA `Any`. Each `Any` in the sequence denotes an individual log record.

If the log is full while writing, then the `LogFull` user exception is thrown. The exception also contains the number of records written from the original sequence of `Anys`.

If the log's state is `off_duty` the `LogOffDuty` user exception is thrown. If the log's state is `locked` the `LogLocked` user exception is thrown. If the log is `disabled` the `LogDisabled` exception is thrown.

The following code snippet shows steps to write some TMN events using the `write_records` operation.

C++

```
// TMN events
TMN::Event event;
TMN::AttrValChgSeq attrvalchg_info;
TMN::AttrValSeq objcrt_info;
TMN::AttrValSeq objdel_info;
TMN::QoSAlarmInfo qosalrm_info;

// Fill TMN events with some data
misc::forge_event_header(event.header);
misc::forge_attrValChgInfo(attrvalchg_info);
misc::forge_objCrtInfo(objcrt_info);
misc::forge_objDelInfo(objdel_info);
misc::forge_qosAlrmInfo(qosalrm_info);

// Sequence of Anys to be written
DsLogAdmin::Anys  anys;
anys.length(4);

// Insert the TMN events into Any Sequence
event.name = (const char*)
  "TMN::Events::attributeValueChange";
event.info <<= attrvalchg_info;
anys[0] <<= event;

event.name = (const char*)
  "TMN::Events::objectCreation";
event.info <<= objcrt_info;
anys[1] <<= event;

event.name = (const char*)
  "TMN::Events::objectDeletion";
event.info <<= objdel_info;
anys[2] <<= event;
event.name = (const char*)
  "TMN::Events::qosAlarm";
event.info <<= qosalrm_info;
anys[3] <<= event;

// Write the sequence of Anys to log
basic_log->write_records(anys);
```

Java

```
// TMN events
TMN.Event event = new TMN.Event();
TMN.AttrValChgSeqHolder  attrvalchg_info =
  new TMN.AttrValChgSeqHolder();
TMN.AttrValSeqHolder objcrt_info =
  new TMN.AttrValSeqHolder();
TMN.AttrValSeqHolder objdel_info =
  new TMN.AttrValSeqHolder();
TMN.QoSAlarmInfo qosalrm_info =
  new TMN.QoSAlarmInfo();

// Fill TMN events with some data
event.header = new TMN.EventHeader();
event.info = orb.create_any();
Util.forge_event_header(event.header);
Util.forge_attrValChgInfo(attrvalchg_info);
Util.forge_objCrtInfo(objcrt_info);
Util.forge_objDelInfo(objdel_info);
Util.forge_qosAlrmInfo(qosalrm_info);

// Sequence of Anys to be written
org.omg.CORBA.Any[] anys =
  new org.omg.CORBA.Any[4];
for (int i = 0; i < 4; i++)
{
  anys[i] = orb.create_any();
}

// Insert the TMN events into Any Sequence
event.name = "TMN::Events::attributeValueChange";
TMN.AttrValChgSeqHelper.insert(event.info,
  attrvalchg_info.value);
TMN.EventHelper.insert(anys[0],event);

event.name = "TMN::Events::objectCreation";
TMN.AttrValSeqHelper.insert(event.info,objcrt_info.value);
TMN.EventHelper.insert(anys[1],event);

event.name = "TMN::Events::objectDeletion";
           TMN.AttrValSeqHelper.insert(event.info,objdel_info.value);
TMN.EventHelper.insert(anys[2],event);

event.name = "TMN::Events::qosAlarm";
           TMN.QoSAlarmInfoHelper.insert(event.info,qosalrm_info);
TMN.EventHelper.insert(anys[3],event);

// Write the sequence of Anys to log
basic_log.write_records(anys);
```

4

Understanding the Log interface

Log characteristics are the same for both event-based log objects and basic log objects. These characteristics are captured in the `DsLogAdmin::Log` interface. All log objects inherit from this interface and therefore have common characteristics.

In this chapter, the following topics will be explained:

- Log and Typed Log records
- Log Quality of Service
- Log size and manipulation
- Setting log attributes
- Copying logs
- Log record query, retrieval and iterators
- Deleting log records

Log and Typed Log records

When an event aware or event unaware application uses the VisiTelcoLog Service to write records to logs using `push`, `pull`, or `write_record` operations, for each received event or each CORBA `Any` in the `Any` sequence a `LogRecord` is created. Similarly, `TypedLogRecord` is the log record created for each typed event received.

The `LogRecord` and `TypedLogRecord` structures are described in the following IDL snippet.

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

struct TypedLogRecord
{
  RecordId id;
  TimeT time;
  NVList attr_list;
```

```
    RepositoryId interface_id;
    Identifier operation_name;
    ArgumentList arg_list;
  };
```

For more detailed structure definitions, please see the OMG Telecom Log Service Specification.

In the structures given in the IDL snippet above, `RecordId id` is a unique number assigned to the record by the log and is unique in the log only.

`TimeT time` is the time stamp for the record, when the record was written to the underlying back end.

`NVList attr_list` can store a list of user-defined attributes for each log record. The attributes are not attached to the log records at the time of writing, but using separate `set_attribute()` API. See Setting log attributes for further information on setting attributes.

The log data itself is stored in the CORBA `Any`. For typed events, the log data is encapsulated in the argument list for the typed event operation.

`RepositoryId interface_id` and `Identifier operation_name` are the repository ID of the interface and the operation name of the operation that emitted the typed event.

Log Quality of Service

In compliance with OMG Telecom Log Service Specification, VisiTelcoLog Service provides a lightweight Quality of Service framework with `set_log_qos()` and `get_log_qos()` APIs. This is in addition to the extensive quality of service framework of the Notification Service specification.

VisiTelcoLog Service supports the following Quality of Service properties:

QoS property	Description
QoSNone	When this is specified no Quality of Service is promised. Calling `flush()` operation will not flush log records.
QoSFlush	When this is specified, calling `flush()` will flush/commit all the log records to the back end.
QoSReliability	When this is specified, log records will be written directly to the back end.

VisiTelcoLog Service takes only the highest value of the Quality of Service specified in the `set_log_qos()` operation. For example, If all the three Quality of Service properties are specified, then only `QoSReliability` is taken. This is reflected in `get_log_qos()` operation. The following code snippet illustrates this point.

C++

```
DsLogAdmin::QosList qos;
qos.length(3);
qos[0] = DsLogAdmin::QoSNone;
qos[1] = DsLogAdmin::QoSFlush;
qos[2] = DsLogAdmin::QoSReliability;

// set all the three QoS
basic_log->set_log_qos(qos);

// Only QoSReliability
qos = basic_log->get_log_qos();
```

Log size and manipulation

This section explains how to control the log size, determine the log full action, and control log record life.

Controlling the log size

The maximum size (in bytes) of the log can be specified at log creation time. All the log factory log creation operations take a log size parameter (see code snippets in Using log factories for examples). Log size is the maximum size the log can grow to. Size of 0 (zero) means that there is no predefined limit, and the log can grow indefinitely. Once the size has been set it can be altered again by using the `set_max_size()` and `get_max_size()` operations. The maximum size of the log is different from current size. Current size is the number of bytes taken up by the log records.

Calling the `set_max_size()` with a new value less than the current size of the log throws `InvalidParam` user exception. Calling `set_max_size()` with any value less than 1 MB will also throw `InvalidParam`. A minimum of 1 MB is required for the maximum size value. This is an implementation limit. Attempting to create a log with initial maximum size less than 1 MB will automatically set the maximum size to 1 MB.

Log full action

If the current size of the log reaches the maximum size, then the log is said to be in a *log full* condition. Under such a log full condition, VisiTelcoLog Service specifies the log full action that needs to be taken. The default log full action of any log is specified when the log is created.

By calling `set_log_full_action()`, the action to be taken in a log full condition can be specified to `wrap` or `halt` the log. When the log full action is `wrap`, the oldest log records are deleted until there is enough space that the new log record can be written.

When the log full action is `halt`, and if the maximum record life for the log is specified, then all the log records that have expired are deleted from the log. Once the expired records are deleted the write operation attempt is repeated. If the write fails again appropriate exceptions are thrown. See Using logs for “event aware” applications and Using logs for “event unaware” applications for the exceptions thrown and detail on write operations.

Log record life

Log record life can be specified by the `set_max_record_life()` API, with units in seconds. Specifying a value of 0 (zero) for maximum record life creates a condition where no log records ever expire.

If the log record life is specified, a garbage collector thread will attempt to delete all expired log records periodically. By default the garbage collector thread starts every 60 minutes. The time interval for this thread can be configured using the property `vbroker.dslog.backend.garbageCollectorInterval`.

Setting log attributes

In compliance with OMG Telecom Log Service Specification, VisiTelcoLog Service allows client applications to define an attribute list of name-value pairs that are meaningful to the application for log records. These log record attributes (as shown in the log record structure) are readable and writable.

Using the log record ID or grammar and constraint, attributes can be set or retrieved for log records. Using the `set_record_attribute()` API, attributes can be set on log records based on log record ID. Similarly, using the `set_records_attribute()` API, attributes can be set on multiple log records which meet the constraint expression specified in the grammar and constraint parameters.

Please note that VisiTelcoLog Service is optimized for log *writing*. For this reason these operations are comparatively expensive. While setting attributes, the entire log is copied and then replaced.

Copying logs

In compliance with OMG Telecom Log Service Specification, VisiTelcoLog Service provides two copy operations to make a copy of an existing log object. The `copy()` operation creates an empty log with similar characteristics as the original log. The log ID of the new log object copy is returned in the out parameter.

The `copy_with_id()` operation takes a log ID and creates an empty log with the input log ID with characteristics similar to the original log. If a log with the input log ID already exists, the `LogIdAlreadyExists` user exception is thrown. Both of the operations throw `NO_RESOURCES` system exception if the log factory cannot create a new log because of resource constraints.

Log record query, retrieval and iterators

In compliance with OMG Telecom Log Service Specification, VisiTelcoLog Service provides two methods to query for log records:

- The `retrieve` method retrieves records based on time.
- The `query` method retrieves records based on constraint.

For typed log records the corresponding methods are:

- The `typed_retrieve` method retrieves records based on time.
- The `typed_query` method retrieves records based on constraint.

The `retrieve` and `query` methods return an iterator as an out parameter to handle large record retrievals. Please note that the `query` and `retrieve` operations are sequential in nature, and they may be time consuming if the number of log records is very large.

Retrieving records based on time

The Log interface provides the `retrieve()` and `typed_retrieve()` methods to perform queries based on time. You can also specify how many records in sequence forwards or backwards to retrieve from the specified time. An iterator may be provided to handle large record retrievals. The following code snippet is an example of how to retrieve records based on time.

C++

```
DsLogAdmin::TimeT from_time;
DsLogAdmin::RecordList_var time_recs;
DsLogAdmin::Iterator_var time_itr;
...
// Starting from 'from_time' retrieve 10 records backwards (i.e -10).
// Store any remaining records in an Iterator 'time_itr'
// if the number of records to retrieve is greater than 1000

time_recs =
  log->retrieve( from_time, -10, time_itr.out() );
...
```

Java

```
org.omg.DsLogAdmin.TimeT from_time;
org.omg.DsLogAdmin.RecordList time_recs = null;
org.omg.DsLogAdmin.Iterator time_itr = null;
...
// Starting from 'from_time' retrieve 10 records backwards (i.e -10).
// Store any remaining records in an Iterator 'time_itr'
// if the number of records to retrieve is greater than 1000
time_recs =
  log.retrieve( from_time, -10, time_itr );
...
```

Querying for records based on constraint

The Log interface provides the `query()` and `typed_query()` methods to perform queries based on a given constraint. The constraint is based on the VisiBroker VisiNotify Filter Constraint. See Writing Filter Constraint Expressions in the VisiBroker VisiNotify Guide for information about writing constraints using the Extended Trader Constraint Language (Extended TCL). A `query` call takes in a grammar to use and the constraint expression, and an iterator may be provided to deal with a large number of records.

When you write constraints to query `LogRecord` or `TypedLogRecord` structures see Log and Typed Log records for their definition.

The following example illustrates how to query using constraints. Note that VisiTelcoLog Service only recognizes the default `EXTENDED_TCL` as the grammar for constraints.

C++

```
DsLogAdmin::RecordList_var recs_found;
DsLogAdmin::Iterator_var itr;
...
// Query using the "EXTENDED_TCL" grammar and
// search for log records with an id below 100 "$.id
```

Java

```
omg.org.DsLogAdmin.RecordList recs_found = null;
omg.org.DsLogAdmin.Iterator itr = null;
...
// Query using the "EXTENDED_TCL" grammar and
// search for log records with an id below 100 "$.id
```

Iterators

Iterators are returned by a `retrieve()` or `query()` method when a large number of log records is returned. The number of records that a `retrieve()` or `query()` method should return before using an iterator is controlled by the `vbroker.dslog.getRecMaxList` property. If the number of records matched from a `query()` or a `retrieve()` operation is greater than the value specified by `vbroker.dslog.getRecMaxList` the excess matched log records will be added to an iterator. Note that when `typed_retrieve()` or `typed_query()` is called a `TypedRecordIterator` is returned.

A log iterator provides two methods: `get()` and `destroy()`. The `get()` method allows the caller to retrieve the records stored by the iterator. When you call the `get()` method you need to indicate the position and how many records to obtain from the specified position. Note that the position in the iterator moves forward only, therefore you cannot request values before the position of the last request. Requesting for invalid values will throw an `InvalidParam` exception.

The following code snippet is an example of how to use an iterator's `get()` method.

C++

```
DsLogAdmin::RecordList_var recs_found;
DsLogAdmin::Iterator_var itr;
...
// Query using the "EXTENDED_TCL" grammar and
// search for log records with an id below 100 "$.id
```

Java

```
omg.org.DsLogAdmin.RecordList recs_found = null;
omg.org.DsLogAdmin.Iterator itr = null;
...
// Query using the "EXTENDED_TCL" grammar and
// search for log records with an id below 100 "$.id
```

When an iterator has been exhausted, and we call `get()` and use the position of the last record in the iterator, the `get()` method will return an empty log record list to the caller. This indicates that the iterator has been exhausted. The application must ensure that the `destroy()` method is called in order to destroy the object from the VisiTelcoLog Service.

Deleting log records

The Log interface allows deletion of log records and typed log records using either grammar and constraint expression or by ID. Two APIs, `delete_records()` and `delete_records_by_id()`, are provided for this purpose and are described in the following table.

Method	Description
delete_records()	Deletes log records based on grammar and constraint expression.
delete_records_by_id()	Deletes log records based on log record ID numbers.

VisiTelcoLog Service optimizes event log records and typed event log record deletion by not deleting them immediately, but marking them as deleted. Over time, the log can become fragmented because of this optimization. For this reason, when the fragmentation exceeds a fragmentation limit, the default for which is 75 percent (which can be configured by the user using property See the section on Properties), the delete operation automatically kicks in the defragmentation thread. Defragmentation logic is essentially a copying operation, where all the log records are reflowed. Please note that the defragmentation operation is expensive.

The following code snippet illustrates deleting a log record of ID 200 using grammar and constraint expressions. The same thing can also be achieved using `delete_records_by_id()`.

C++

```
// constraint for log rec with id = 200
const char* grammar = "EXTENDED_TCL";
const char* constraint = "$.id == 200";

// delete the log record matching the constraint
basic_log->delete_records(grammar, constraint);
```

Java

```
// delete the log record where the log record id = 200
basic_log.delete_records("EXTENDED_TCL", "$.id == 200");
```

5

Advanced features

This section covers the following advanced topics:

- Log duration
- Log scheduling
- Log generated events

Log duration

Setting a log duration interval allows users to create a coarse-grained time interval (window) during which an unlocked and enabled log object is functional. When the log duration is set the log object will only allow writing log records or events to the log within the specified time interval.

The log duration time interval is set and retrieved with the following methods:

```
set_interval(in DsLogAdmin::TimeInterval interval);
```

and

```
DsLogAdmin::TimeInterval get_interval();
```

The input parameter and return value are an IDL structure defined as:

```
module DsLogAdmin {
 typedef TimeBase::TimeT TimeT;
 struct TimeInterval {
  TimeT start;
  TimeT stop;
 };
};
```

The `start` and `stop` fields of a time interval are of type `CORBA::ULongLong`. Their values are numbers of 10^{-7} seconds (or 100 nanoseconds) counted from 00:00:00, Oct 15, 1582 using Greenwich Mean Time (GMT).

Although the `start` and `stop` time unit is specified by OMG as 10^{-7} second, the actual time resolution supported by VisiTelcoLog is in seconds. `Start` and `stop` values specified in `set_interval()` will be rounded to the nearest value of full seconds by the VisiTelcoLog Service.

If the `start` and `stop` values are both set to 0 (zero), or rounded to zero seconds, the log will always be in a functional state.

To retrieve the current log duration setting, users can call the `get_interval()` operation on the target log.

Log scheduling

Log scheduling allows users to set a series of fine-grained weekly time intervals (weekly masks) on a given log object. When scheduling is set up the log object will only allow writing log records or events to the log within these time intervals, if it is within a log duration (see Log duration above), and the log is in an unlocked and enabled state.

Log scheduling time intervals are set and retrieved via the following methods:

```
set_week_mask(in DsLogAdmin::WeekMask weekmask);
```

and

```
DsLogAdmin::WeekMask get_week_mask();
```

The input parameter and return value of above methods are an IDL sequence of an IDL structure `WeekMaskItem`. They are defined as:

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

 struct Time24Interval {
  Time24 start;
  Time24 stop;
 };

 typedef sequence<Time24Interval> IntervalsOfDay;
 typedef unsigned short DaysOfWeek;

 struct WeekMaskItem {
  DaysOfWeek days;
  IntervalsOfDay intervals;
 };

 typedef sequence<WeekMaskItem> WeekMask;
};
```

Greenwich Mean Time zone (GMT) is used by default. The user can choose to use the local time zone of the log server by starting the VisiTelcoLog Service with the following property setting:

```
vbroker.dslog.scheduleByServerLocalTime=true
```

For diagnostic purposes the log schedule setting changes and active behavior can be observed on the Console stdout by starting the VisiTelcoLog Service with the following property setting:

```
vbroker.dslog.timerDebug=true
```

VisiTelcoLog Service is shipped with an example of log schedule in the following directory:

```
<install_dir>/examples/vbroker/telcolog/primitive_cpp/scheduler.C
```

The following C++ code snippet illustrates how to use `set_week_mask()`:

```
// 7:30 am to 12:00 am
DsLogAdmin::Time24Interval morning = {{7,30},{12,0}};

// 13:30 (1:30 pm) to 17:30 (5:30 pm)
DsLogAdmin::Time24Interval afternoon = {{13,30},{17,30}};
```

```
// 21:00 (9:00 pm) to 23:30 (11:30 pm)
DsLogAdmin::Time24Interval night = {{21,0},{23,30}};

//19:30(7:30pm)to22:30(11:30pm)
DsLogAdmin::Time24Interval evening = {{19,30},{22,30}};

// 9:00 am to 16:30 (4:30 pm)
DsLogAdmin::Time24Interval wkend_day = {{9,0},{16,30}};

DsLogAdmin::WeekMask new_weekmask;
new_weekmask.length(2);

// weekday schedule in the 0th weekmask item
new_weekmask[0].days = (DsLogAdmin::Monday
                       | DsLogAdmin::Tuesday
                       | DsLogAdmin::Wednesday
                       | DsLogAdmin::Thursday
                       | DsLogAdmin::Friday );

new_weekmask[0].intervals.length(3); // 3 intervals
new_weekmask[0].intervals[0] = morning;
new_weekmask[0].intervals[1] = afternoon;
new_weekmask[0].intervals[2] = night;

// weekend schedule in the 1st weekmask item
new_weekmask[1].days = (DsLogAdmin::Sunday
                       | DsLogAdmin::Saturday );

new_weekmask[1].intervals.length(2); // 2 intervals
new_weekmask[1].intervals[0] = wkend_day;
new_weekmask[1].intervals[1] = evening;

// set new week mask on the log
log->set_week_mask(new_weekmask);
```

The following C++ code snippet illustrates how to use `get_week_mask()` and process the result:

```
// retrieve current week mask from the log
DsLogAdmin::WeekMask_var holder;
holder = log->get_week_mask();

const char* day_names[7] = {
 "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
};

const DsLogAdmin::WeekMask& mask = holder.in();
CORBA::Short day, daybit;
CORBA::ULong i, j;




// print retrieved schedule by days.
for(day=0,daybit=1;day<7;daybit = daybit*2, day++) {
    cout << "    " << day_names[day] << ": ";
    for(i=0;i<mask.length();i++) {
        const DsLogAdmin::WeekMaskItem& item = mask[i];

        if( (daybit & item.days) == 0 ) {
            continue;
            }

            for(j=0;j<item.intervals.length();j++) {
                const DsLogAdmin::Time24Interval& interval =
                                                item.intervals[j];
                char buf[32];

                sprintf(buf, "[%02u:%02u-%02u:%02u] ",
                    interval.start.hour,
                    interval.start.minute,
                    interval.stop.hour,
                    interval.stop.minute);

        cout << buf;
            }
        }

        cout << endl;
        }
    }
```

On processing `set_week_mask()` requests, the log object server validates the input weekly mask parameter. Exceptions that are raised on `set_week_mask()` and their corresponding weekly mask setting errors are explained in the following table.

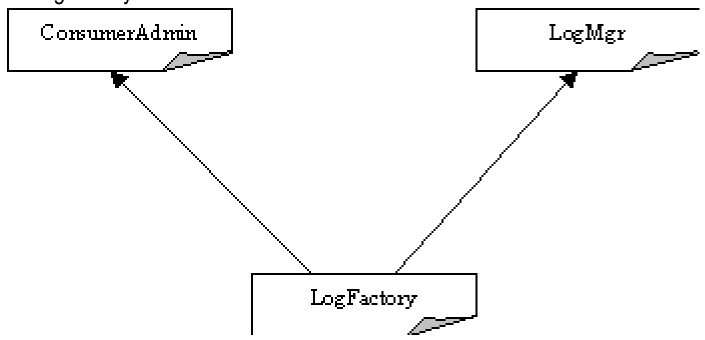
Exception	Description
DsLogAdmin::InvalidTime	Hour or minute field in one of the interval's start or stop fields is out of range. The valid range for hour is 0 to 23, and the valid range for minute is 0 to 59.
DsLogAdmin::InvalidTimeInterval	*Case 1:* Start time is later than stop time. Therefore, an interval starting at midnight and stopping after midnight is not supported. The effect of an interval that spans days should be done using two intervals: one that stops before just before midnight (23:59) and another that starts just after midnight on the next day (00:00). *Case 2:* Time intervals overlap. Start or stop time of one scheduled interval is within the bounds of another scheduled interval in the same weekly mask parameter.

On failure of `set_week_mask()` due to errors, the log's existing weekly mask will remain and a `DsLogNotification::ProcessingErrorAlarm` log event (see Log generated events) will be sent. On success of `set_week_mask()` the existing weekly mask will be completely replaced by the new weekly mask. Therefore, to completely erase an existing weekly mask, the application can invoke `set_week_mask()` with an empty weekly mask that is a weekly mask of length zero. A log with an empty weekly mask will accept logging during the whole week.

Log generated events

According to the OMG Telecom Log Service specification, event-aware Log factories and logs can generate events on log object creation and deletion, state and attribute change, threshold crossover, and processing error. A value-added extension of the VisiTelcoLog Service allows a `BasicLog` object to generate these events. These log generated events are called *log events*. Therefore, in VisiTelcoLog Service, a log factory (Basic, Event, TypedEvent, Notify, or TypedNotify factory) is a `CosNotifyChannelAdmin::ConsumerAdmin`.

Figure 5.1 Log factory inheritance interface



The purpose of `LogFactory` “is a” `ConsumerAdmin` is to expose downstream or consumer-side functionality of an event channel inside each log factory. This event channel is called a *log event channel*. Log events generated from a log factory and from its logs are all sent to the log event channel of this factory. To receive log events an application can create consumer-side proxies on the log factory through its operations inherited from `ConsumerAdmin` and connect to these proxies.

The following C++ code (also located in <install_dir>/examples/vbroker/telcolog/primitive_cpp/logEventReceiver.C) illustrates how to connect an event consumer to log event channel of a `NotifyLogFactory`:

```
int main(int argc, char** argv)
{
 CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // get service reference (the Notify Log Factory)
 CORBA::Object_var service
  = orb->resolve_initial_references(
                                  "NotifyLogService");

 // directly narrow the factory to consumer admin.
 CosNotifyChannelAdmin::ConsumerAdmin_var admin
        = CosNotifyChannelAdmin::ConsumerAdmin
                                   ::_narrow(service);

 CosNotifyChannelAdmin::ProxyID proxy_id;

 // create a proxy
 CosNotifyChannelAdmin::ProxySupplier_var proxy
  = admin->obtain_notification_push_supplier(
   CosNotifyChannelAdmin::ANY_EVENT, proxy_id);

 CosNotifyChannelAdmin::ProxyPushSupplier_var supplier;
 supplier =  CosNotifyChannelAdmin::ProxyPushSupplier
                                        ::_narrow(proxy);


// allocate the consumer implementation
 PushConsumerImpl* servant = new PushConsumerImpl;

 // activate it on root poa
 CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
 PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);
 poa->activate_object(servant);

 // activate the root poa
 PortableServer::POAManager_var poa_manager
```

```
        = poa->the_POAManager();
 poa_manager->activate();

 // get consumer object reference
 CORBA::Object_var ref
        = poa->servant_to_reference(servant);
 CosNotifyComm::PushConsumer_var consumer =
   CosNotifyComm::PushConsumer::_narrow(ref);

 // connect the consumer to the supplier proxy
 supplier->connect_any_push_consumer(consumer);

 cout << "log event receiver is ready" << endl;

 // work loop
 orb->run();
   }
   catch(CORBA::Exception& e) {
   cout << "caught exception:" << endl << e << endl;
   }

   return 0;
}
```

The following Java code illustrates how to connect an event consumer to a log event channel of a `NotifyLogFactory`:

```
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.PortableServer.*;
import org.omg.CosNotifyComm.*;

public class logEventReceiver {

    public static void main(String[] args) {
      try {
        org.omg.CORBA.ORB orb
            = org.omg.CORBA.ORB.init(args, null);

        // get service reference (the Notify Log Factory)
        org.omg.CORBA.Object service
            = orb.resolve_initial_references(
                                 "NotifyLogService");

        // directly narrow the factory to a consumer admin.
        ConsumerAdmin admin
         = ConsumerAdminHelper.narrow(service);

        org.omg.CORBA.IntHolder proxy_id
            = new org.omg.CORBA.IntHolder();
        // create a proxy
        ProxySupplier proxy
            = admin.obtain_notification_push_supplier(
                          ClientType.ANY_EVENT, proxy_id);

        ProxyPushSupplier supplier
            = ProxyPushSupplierHelper.narrow(proxy);

        // allocate the consumer implementation
        PushConsumerImpl servant = new PushConsumerImpl();

        // activate it on root poa
```

```
            org.omg.CORBA.Object obj
                = orb.resolve_initial_references("RootPOA");
            POA poa = POAHelper.narrow(obj);
            poa.activate_object(servant);

            // activate the root poa
            POAManager poa_manager = poa.the_POAManager();
            poa_manager.activate();

            // get consumer object reference
            org.omg.CORBA.Object ref
                = poa.servant_to_reference(servant);
            PushConsumer consumer
                = PushConsumerHelper.narrow(ref);

            // connect the consumer to the supplier proxy
            supplier.connect_any_push_consumer(consumer);

            System.out.println("untyped push consumer is ready");

            // work loop
            orb.run();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Possible log events and their meanings have been specified by OMG as described in the following sections.

Object Creation Event

This event is emitted from a log factory itself on a successful log object creation. The new log ID and the log creation time is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

```
module DsNotification {
struct ObjectCreation
   {
        LogId   id;
        TimeT   time;
   };
};
```

Object Deletion Event

This event is emitted from a log factory itself on a successful log object deletion. The deleted log ID and the log deletion time is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

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

Attribute Value Change (AVC) Event

This event is emitted from a log on a successful log attribute value change. Information about the attribute value change is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

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

In this structure

- `logref` is the reference of the log object itself.
- `id` is the log ID of the log object.
- `time` is the time the attribute value change was made.
- `type` indicates the type of the changed attribute. See discussion below.
- `old_value` encapsulates the original value of the attribute before the change.
- `new_value` encapsulates the new value of the attribute after the change.

OMG specifies following attribute types of log object:

Attribute type	**Description**
capacityAlarmThreshold (type = 0)	This type of AVC event is triggered by a successful `set_capacity_thresholds()` invocation on a log object and changes its previous capacity alarm threshold setting.
logFullAction (type = 1)	This type of AVC event is triggered by a successful `set_full_action()` invocation on a log object and changes its previous log full action setting.
maxLogSize (type = 2)	This type of AVC event is triggered by a successful `set_max_size()` invocation on a log object and changes its previous log max size setting.
startTime (type = 3)	This type of AVC event is triggered by a successful `set_interval()` invocation on a log object and changes its log interval start time setting.
stopTime (type = 4)	This type of AVC event is triggered by a successful `set_interval()` invocation on a log object and changes its log interval stop time setting.
weekMask (type = 5)	This type of AVC event is triggered by a successful `set_week_mask()` invocation on a log object.
filter (type = 6)	This type of AVC event is triggered by a successful set_filter() invocation on a log object and changes its filter.
maxRecordLife (type = 7)	This type of AVC event is triggered by a successful `set_max_record_life()` invocation on a log object and changes its max record life setting.
qualityOfService (type = 8)	This type of AVC event is triggered by a successful `set_log_qos()` invocation on a log object and changes its log QoS setting.

State Change Event

This event is emitted from a log on OMG specified log state change. Information about the state change is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

```
module DsNotification {
struct StateChange
   {
        Log        logref;
        LogId      id;
        TimeT      time;
        StateType  type;
        Any        new_value;
   };
};
```

In this structure

- `logref` is the reference of the log object itself.
- `id` is the log id of the log object.
- `time` is the time of the state change.
- `type` indicates the type of the changed state. See discussion below.
- `new_value` encapsulates the new state value after the change.

OMG specifies following state change event types for a log object:

State change event type	Description
administrativeState (type = 0)	This type of state change event is triggered by a successful `set_administrative_state()` invocation on a log object and changes its administrative state, allowing or disallowing log record write operations (insert, update, delete, etc.).
operationalState (type = 1)	This type of state change event is not used by the VisiTelcoLog Service implementation in this release.
forwardingState (type = 2)	This type of state change event is triggered by a successful `set_forwarding_state()` invocation on a log object and changes its forwarding state, which enables or disables event forwarding.

Threshold Alarm Event

This event is emitted from a log object when a log write operation causes the log to grow beyond its size threshold. Information about the attribute value change is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

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

In this structure

- `logref` is the reference of the log object itself.
- `id` is the log ID of the log object.

- `time` is the time of the occurrence.
- `crossed_value` the threshold value just being crossed.
- `observed_value` the current log space usage percentage.
- `perceived_severity` critical(0), minor(1) and cleared(2).

Processing Error Alarm Event

This event is emitted from a log factory or a log object when a problem occurs within the factory or log object. Information about the attribute value change is encapsulated in the CORBA `Any` event body as an IDL structure defined as:

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

In this structure

- `error_num` is the highest 20 bits of this field which are reserved for vender specific error ids.
- `error_string` is the text string that explains the error.

6

Running the VisiTelcoLog Service

The VisiTelcoLog Service is implemented as a C++ service. VisiBroker for C++ is prerequisite for running VisiTelcoLog Service. To run the service make sure that VisiBroker Smart Agent (osagent executable) is running in the network. To start the VisiTelcoLog Service in the background use the following command:

UNIX

```
prompt> visitelcolog  &
```

Windows

```
prompt> start visitelcolog.exe
```

By default the service starts at port 14200. The port can be changed using the property `vbroker.dslog.listener.port`. Once started, the service prints the following message to the console:

```
Telco Log service is ready
```

VisiTelcoLog Service creates a directory called `visidslog.dir` to store all of its persistent data. By default it creates this directory in the current directory. The location for the data store directory can be changed using the `vbroker.dslog.dir` property. This directory also contains the log back end.

Also note that for the sake of convenience the compiled stub and skeleton code of the OMG Telecom Log Service IDLs are provided as static library. Please see the examples on how to use it. The generated skeletons are for POA.

Getting entry references

VisiTelcoLog starts up by default at port 14200. This port can be changed using `vbroker.dslog.listener.port` property.

Applications trying to bind to `BasicLogService`, `EventLogService`, `NotifyLogService`, `TypedEventLogService` or `TypedNotifyLogService` can use `corbaloc` to resolve initial reference to the service.

Applications can use the following ORB property:

```
-ORBInitRef corbaloc::<host>:<port>/BasicLogService
-ORBInitRef corbaloc::<host>:<port>/EventLogService
-ORBInitRef corbaloc::<host>:<port>/NotifyLogService
```

```
-ORBInitRef corbaloc::<host>:<port>/TypedEventLogService
-ORBInitRef corbaloc::<host>:<port>/TypedNotifyLogService
```

Properties

Property	Default	Description
`vbroker.dslog.listener.port`	14200	Specifies the listener port for the service. Valid values include any legal port value in the port range.
`vbroker.dslog.console`	true	When `true`, prints to the console when the service starts up. For daemon processes, this should be set to `false`.
`vbroker.dslog.dir`	./visidslog.dir	The service stores all of its persistent data in the specified directory. If the directory is not valid or does not have the right permissions, the service will fail to start up. Valid values include any valid directory location.
`vbroker.dslog.getRecListMax`	1000	The number of `LogRecords` that need to be matched in the query for an iterator to be returned.
`vbroker.dslog.scheduleByServerLocalTime`	false	When set to `true`, calls `tzset()` for scheduler time.
`vbroker.dslog.waitForLogAvailables`	20	Waiting period (in seconds) for the pull supplier for space to be available to log a pulled event. Valid values include any non-zero wait duration in seconds.
`vbroker.dslog.basicLogFactory.name`	VisiBasicLogFactory	The name with which the `BasicLog` factory is activated. Valid values include any object name.
`vbroker.dslog.basicLogFactory.iorFile`	null	The name of the file where the `BasicLog` factory object's `IOR` will be written. Valid values include any valid file name.
`vbroker.dslog.eventLogFactory.name`	VisiEventLogFactory	The name with which the event log factory is activated. Valid values include any object name.
`vbroker.dslog.eventLogFactory.iorFile`	null	The name of the file where the event log factory object's IOR will be written. Valid values include any valid file name.
`vbroker.dslog.notifyLogFactory.name`	VisiNotifyLogFactory	The name with which the notify log factory is activated. Valid values include any object name.
`vbroker.dslog.notifyLogFactory.iorFile`	null	The name of the file where the notify log factory object's IOR will be written. Valid values include any valid file name.
`vbroker.dslog.backend.garbageCollectorInterval`	60	The time interval (in minutes) for the log record garbage collector thread to run. When the thread runs, it garbage collects all expired log records. The thread runs only when the record life for the log is specified. Otherwise, it does not run. Valid values fall in the range of 1 to 180 minutes.

Property	Default	Description
`vbroker.dslog.backend.file.fragmentationLimit`	75%	Percentage of fragmentation that triggers automatic defragmentation. Automatic defragmentation happens only when deleting. Valid values fall in the range of 10% to 80%.
`vbroker.dslog.backend.file.dir`	null	The directory location for back end database and support files. The directory path should be valid and should have the necessary permissions. Please note that the performance of the service depends on this directory.
`vbroker.log.enable`	false	To see the debug log statements from this service, set this property to true. For the various source names options for debug log filtering, see the “Debug Logging properties” section of the *VisiBroker for C++ Developer's Guide*.

Index