



CISCO CONFIDENTIAL

CHAPTER 20

Using the Event Distribution System

The Event Distribution System (EDS) was the event-distribution software supplied with CMF, the predecessor system of CWCS. For backward compatibility, it is also supplied with this release of CWCS. EDS provides a means for sending messages from one process to another in a distributed, networked environment.

EDS is one of two event messaging components supplied in this release of CWCS. The other – Event Services Software (ESS) – is an asynchronous messaging service providing a publish-and-subscribe infrastructure and allowing distributed, loosely coupled interprocess communications. *These two components are disjoint systems and do not work together.*



Note

EDS has been deprecated in favor of ESS (see [Chapter 19, “Using Event Services Software”](#)). CWCS currently supports EDS for applications that are still using it, but this support will be withdrawn in a future release. Cisco urges developers to avoid new development with EDS and begin using ESS as soon as possible.

The following topics describe EDS and how to use it:

- [About the EDS Components](#)
- [Using the EDS Programmatic Interface](#)
- [Using EDS to Publish Events](#)

For more information about EDS, see the *EDS v1.1 System Functional Specification*, EDCS ENG-25706.

About the EDS Components

The following topics discuss each of the EDS components:

- [About the EDS Event Server](#)
- [About the EDS Event Message](#)
- [About the EDS Atom Service](#)
- [About the EDS Manager](#)
- [About the EDS Class Loader](#)
- [About the EDS New Event Message Fields](#)
- [About the EDS Event Logger](#)
- [About the EDS Event Logger Display](#)

CISCO CONFIDENTIAL

- [About the EDS Named Event Filter Service](#)
- [About the EDS Event to Trap Converter](#)
- [About the EDS Trap to Event Converter](#)

About the EDS Event Server

The Event Distribution System (EDS) is used to distribute event messages from the creator of the events, the Event Source (ES), to the recipient of the events, the Event Consumer (EC). The event is a data packet or message that contains pertinent information associated with an incident in the things being monitored or managed. Each EC registers an Event Profile (EP) or filter, with the EDS that describes the events in which the EC is interested.

The EC defines a Java boolean function that is used to evaluate the events as they come in. If the boolean function returns true, the event is passed to the EC.

About the EDS Event Message

The Event Message is implemented as a Java class. For C++ event sources, the event message is an IDL data structure. This IDL data structure is sent to EDS. The event sources can provide a Java class that can read the IDL data structure and create an instance of the Event Message Java class. For Java event sources, they can create the Event Message Java class and send that directly. After the event Java class is initialized, it is serialized and sent through the EDS. It is de-serialized in the EDS and sent through the event filter. If the event passes the filter, the serialized event is sent to the interested EC. The Java class is de-serialized in the EC and then the event is processed in the EC. Not only is syntactical information sent with the event message, such as event creation time and event ID, but semantical information can also be sent, such as methods to display user readable messages or conversion to useful traps.

About the EDS Atom Service

The EDS Atom Service maintains the definition of the atoms used to define the elements of the events. Atoms are packages of information that associate an integer value atom number with a string atom name, description, and a locale dependent message. An atom may also have a hierarchical relationship with other atoms. The main purpose of an atom is to let a small piece of data (integer) flow in the event, instead of a much larger stream of data such as a text string. It also allows for internationalization of these text strings.

The Atom Service keeps the definition of each atom as well as a locale-correct message that can be displayed about the atom. It can be dynamically updated so that the system does not have to be stopped and started to define new atoms that appear in the event message.

About the EDS Manager

The EDS Manager keeps track of all the CORBA objects that are part of the EDS system. It allows the EDS system to be administered.

CISCO CONFIDENTIAL

About the EDS Class Loader

Since the filters provided by the event consumers and the event messages themselves are instances of Java classes, a class loader is needed to ensure that EDS and the event consumers can get access to the Java class files.

About the EDS New Event Message Fields

New fields added to event messages are delivery type, time to live and chain pointer.

About the EDS Event Logger

The Event Logger (EL) maintains the current event messages that flow through EDS. This component provides search capability on these events. Searching and grouping of historical events aids in problem determination.

About the EDS Event Logger Display

The Event Logger Display (ELD) is used to display events received from EDS as well as events from the event logger.

About the EDS Named Event Filter Service

The Named Event Filter Service provides the ability for the user to define event filters that can be used by other components of the network management system. These event filters are given a name. The Event Logger and Event to Trap facility will use the Named Event Filter Service.

About the EDS Event to Trap Converter

The Event to Trap Converter (ETC) is used to send traps to an NMS such as HP OpenView. After the events are received, the event to trap converter calls the toTrap() method of each event class. It then constructs a trap from the returned information of that method, and then sends it to the configured NMS.

About the EDS Trap to Event Converter

The EDS Trap to Event component is used to convert incoming traps to events that can be used by other event consumers.

CISCO CONFIDENTIAL

Using the EDS Programmatic Interface

Each EDS component has its own API. The APIs for the EDS are defined as Java classes for both event sources and event consumers. There are C++ APIs for event sources and event consumers. The Event Logger Display provides Java constructors so that it can be instantiated from within another Java application. All the other APIs are in the CORBA IDL.

The following topics discuss the EDS interface components:

- [About EDS Events](#)
- [Formatting EDS Events](#)
- [Defining and Registering EDS Event Atoms](#)
- [Using the EDS Atom Definition File](#)
- [Using the Atom Service Executables](#)
- [Using the EDS Java Interface Classes](#)
- [Registering EDS Application Events](#)
- [Using the EDS Trap to Event Service](#)
- [Using the EDS Trap Receiver Framework](#)
- [Using the Generic Consumer Framework](#)

About EDS Events

EDS events must have a required set of attributes to permit generic processing of the event data. Allowing event creators to provide the data in any format they choose makes it harder for users to specify the events they want to view. For example, if the user wants to see all the events pertaining to a router named “MainRouter.enterprise.com”, it will be difficult to filter on or search for them if one event creator calls the attribute that contains the router name “DeviceName”, while another chooses “Device” and yet another chooses “resource”. The same can be said for event categories, such as security events. If we have only a huge list of all events, the user will have to iterate through that list to find all the security events, and are likely to miss at least some of the security events in which they are interested.

In the CWCS event model, the resource each event concerns is defined in a resource list. Each resource in the list has a type and a unique name. Each resource in the list provides details defining what the event is about. Each event must have a resource list and have at least three resources defined: the device, the device component, and the device resource. The device must also contain the IP address of the device. These fields should be populated. If these fields are not filled in, it will be impossible to search on these events in the event logger, and the user will not be able to tell what object the event refers to when shown in the event log display. If it is not possible to set the resource list to anything, it can be set to null.

Most of the data in the event can be defined as integers, with mappings between each integer and its meaning. An atom registry is required to define these integer mappings. The integers need to be defined company-wide and assigned by an “event police” group. The event atom registry allows retrieval of display messages given an integer mapping, and also provides updates to currently running systems.

Some event situations may not permit you to populate all the required fields. In these situations, set the fields to an empty value, such as 0 for an atom value or a string with a value of “”. Do not use a null string; EDS uses CORBA for interprocess communications and IIOP does not permit sending null strings. Be aware that your users will be able to see the contents of all events, and will set filters and

CISCO CONFIDENTIAL

perform searches on those events. The better the required fields are filled in, the easier it is for users to understand what is happening in the network, since they will be applying their filters or queries to a more complete group of events.

Most events will have additional data that describe what the event is about. This data is defined as attributes in the Java class that extends the event base class and as an IDL data structure. This data should be described using metadata.

Formatting EDS Events

The required event fields are listed here. A detailed explanation of each field follows the list.

- Event Category: Category of this event
- Event ID: Unique identifier of the event in this category
- Event Severity: Severity of this event
- Resource List: List of resources
- Time Stamps: GMT time the event was created and sent
- Time to Live: Time to live for this event
- Delivery Type: Delivery type for this event
- Chain Pointer: Unique identifier of related event
- Event UID: Unique identifier of the instance of this event
- Object ID: CORBA object reference of the service that is responsible for managing what this event is about
- Application UID: Unique identifier of the instance of the application sending this event
- Meta-data: Generic definition of the unique data included with this event.
- URL List: List of URLs to provide additional information about this event

Defining and Registering EDS Event Atoms

CWCS needs a service to give definition to atoms. For example, event categories, event IDs, resource types and address types (not a complete list) can all be defined using these atoms. Then, by using the atoms in the event, the size of the event can be minimized. Applications must have a way to define these atoms and update them dynamically when the application is running.

Atom definitions use the following defined hierarchy to allow them to be grouped (the location in the event where this value should appear is given in parenthesis):

- Event Atoms
- Event Category
- Threshold (eventCategory)
 - event IDs*
 - Security (eventCategory)
 - event IDs*
 - Status (eventCategory)
 - event IDs*

CISCO CONFIDENTIAL

- Topology (eventCategory)
 - event IDs*
- Configuration (eventCategory)
 - event IDs*
- Service (eventCategory)
 - event IDs*
- Informational (eventCategory)
 - event IDs*
- Control (eventCategory)
 - event IDs*
- Event Severity
 - Critical (eventSeverity)
 - Major (eventSeverity)
 - Minor (eventSeverity)
 - Informational ((eventSeverity)

Using the EDS Atom Definition File

Atoms are defined in an atom definition file. A utility is provided to parse the atom definition file, populate the atom service, and (optionally) generate Java and C++ code that defines “constants” to be used in creating the event sources and consumer applications. The keywords and format of the atom definition file are as follows:

```
INCLUDE atom_defintion_file
atomname
LOCALE locale_name
MESSAGE "user readable message"
DESCRIPTION "some description"
= { parent_atomname atomnumber }
```

where:

- *atomname* is the assigned atom name
- *atomnumber* is the assigned atom number
- *parent_atomname* is the name of the parent atom (to define the hierarchy)
- *atom_definition_file* is the file that defines the parent atom
- *locale_name* is one of the following keywords, used to define the locale of this message:
 - INCLUDE - include this atom definition file for defining parent names
 - LOCALE - locale value for this message
 - MESSAGE - the user understandable message
 - DESCRIPTION - description of this message

The atom definition file for the base atom definitions is BaseAtomDef.atom.

The main Java class that parses the atom definition file is com.cisco.cmf.eds.atom.ParseMain. It has the following usage statement:

CISCO CONFIDENTIAL

```
java com.cisco.cmf.eds.atom.ParseMain [-J] [-C] [-U [-R]] [-I include_dirs ...] -F
atom_definition_file.atom -h
```

where:

- -J generates a Java file.
- -C generates a C++ include file.
- -U updates the atom service.
- -R replaces the atom definition in the atom service (if it exists).
- -I looks in the include_dirs for include files.
- -F specifies the file that contains the atom definition file (atom_definition_file).
- -h prints a usage message.

The generated Java file defines attributes for each atom in the atom definition file. These are defined as follows:

```
public class atom_definition_file {
    public static final atomname = #;
}
```

Java “constants” of the form `atom_definition_file.atomname` can be used in Java event sources and in event consumers.

The generated C++ include file defines attributes for each atom in the atom definition file. These are defined as follows:

```
class atom_definition_file {
    public:
        enum atom_definition_fileEnum {
            atomname = #
        };
};
```

C++ “constants” of the form `atom_definition_file::atomname` can be used in C++ event sources.

The atom service uses Java Resource Bundles to access the atom information. The program receiving the event should use the `getBundle()` static method, passing in as the base name “com.cisco.nm.cmf.eds.atom.EventAtomBundle”. Then, turn the atom integer value into a string, and use that as the key in the `getString()` and `getStringArray()` instance methods of the bundle. The following keys values are defined:

```
key - get message, if the message is not defined, return the key value
key.name - get atomname
key.description - get atom description
key.fields - get string array of keys defined under this key in the hierarchy
```

**Note**

`key` is the string value of the integer passed in the event. `key.name` is the key value with the string “.name” appended on the end (ie. `key+”.name”`). `key`, `key.name` and `key.description` return strings and `key.fields` returns a string array.

See JDK information for exact method syntax and exceptions thrown for using Resource Bundles.

CISCO CONFIDENTIAL

Using the Atom Service Executables

The main Java class for the atom service is `com.cisco.cmf.eds.atom.Main`. It has the following runtime options:

```
java com.cisco.cmf.eds.atom.Main [-D dir_name] [-h]
```

where:

- `dir_name` is the name of directory where the atom service stores its data. It is stored in a file called `atom.src`. If no directory is specified, the directory where the atom service is started is used.
- `-h` prints usage message.

The atom service stores its data in a file called `atom.str` in the atom service's working directory. If the store file is not there, or if the atom service has a problem reading the store file, all atom definition files (files that end with `*.atom`) are read in to initialize the service.

When a new network management application is installed, it should copy its `*.atom` definition file into the atom service working directory. It should then make sure the atom service is running and then run the atom parse program using the `-U` and if necessary, the `-R` option to update the atom service with the new atoms used by the application.

Using the EDS Java Interface Classes

Java classes are provided for both the event source and event consumer to interface with EDS. These classes are defined in the `com.cisco.cmf.eds.system.*` package. They are as follows:

```
public final class EventSource {
    // Public Constructor
    public EventSource(String name);
    public EventSource(String name, String hostname);
    public EventSource(String name, int queuesize);
    public EventSource(String name, String hostname, int queuesize);
    public EventSource(String name, EventSourceCallbackInterface callback);
    public EventSource(String name, int queuesize,
        EventSourceCallbackInterface callback);
    public EventSource(String name, String hostname, int queuesize,
        EventSourceCallbackInterface callback);
    // Public Instance Variables
    public boolean init;
    // Public Instance Methods
    public EventUID sendEvent(EventBase event);
    // send event
    public EventUID sendEventNoBlock(EventBase event)
        throws QueueFullException;
    //send event, do not block
    //if queue is full, throw
    // exception if queue is full
    public void setCallback(EventSourceCallbackInterface callback);
    // set callback interface
    public void deleteConnection();
    // delete connection to
    // EDS
}

public interface EventSourceCallbackInterface {
    public void eventSent(EventBase event, Exception ex);
    // this event has been sent if the
    // event was sent, ex was null if
    // there was an exception and this
```


CISCO CONFIDENTIAL

```

// event was unable to be sent, ex
// will not be null
}
public final class FilteredEventConsumer {
// Public Constructor
public FilteredEventConsumer(String name);
public FilteredEventConsumer(String name, EventCallbackInterface
Eventcallback);
public FilteredEventConsumer(String name, String hostname, EventCallbackInterface
eventcallback);
// Public Instance Methods
public void setCallback (EventCallbackInterface eventcallback);
// used to define java callback
// interface
public void setClassInstance (EventFilterInterface object) throws
com.cisco.nm.cmf.eds.exceptions.InvalidInterface, java.io.IOException;
// use to define the filter class
// instance
public void setClassSource(String class_name, String class_source) throws
com.cisco.nm.cmf.eds.exceptions.InvalidInterface,
com.cisco.nm.cmf.eds.exceptions.CompilerError;
// used to define the filter class
// source
public void setQueryFilter(String filter_string) throws
com.cisco.nm.cmf.eds.exceptions.InvalidInterface,
com.cisco.nm.cmf.eds.exceptions.CompilerError,
com.cisco.nm.cmf.eds.exceptions.QueryParseException;
// used to define a "query" to be
// used to evaluate events for
// possible interest by this
// consumer
public void setFilterName(String filtername) throws
com.cisco.nm.cmf.eds.exceptions.FilterNameNotFound;
// used to set a named filter to be
// used to evaluate events for this
// consumer
public void setEventFilter(EventFilter filter) throws java.io.IOException;
// used to set a filter straight from
//the event filter repository
public void deleteConnection();
// delete connection to EDS
}
public interface EventCallbackInterface {
public void eventReceived (EventBase event, EventData s_event);
// after the FilteredEventClass
// has received an event, it calls
// this method of the registered
//callback interface
}
public interface EventFilterInterface {
public boolean evaluateEventData(EventBase event);
// the filter class must implement
// this interface. The consumer
// connector will call this method
// for every event received
}

```

CISCO CONFIDENTIAL

Registering EDS Application Events

All event source applications need to register the events they plan on sending. This registration allows a more detailed and complete filtering of events generated in the system. The event sources need to instantiate a “dummy” copy of events it plans on sending. An exhaustive list of events is required that encompasses all the variations of Java event classes, Event Categories, Event IDs, Event Severities, Resource Types and Resource ID Types.

This is the Java class that interfaces to the event repository:

```
public class EventHash extends java.util.Hashtable{
//Public Constructor
    .public EventHash();
//Public Instance Methods
    .public Object put(EventBase event);
//add “dummy” event to event repository
}
```

To store and access events in the repository, an instance of the EventHash class must be created. The defined hash methods can then be used to access the events. If the application is going to store the events in the repository, it must use the defined put() method above.

Using the EDS Trap to Event Service

EDS includes a generic trap receiver that receives SNMP traps and then acts on those traps. The trap receiver supports receiving SNMP traps from a defined port (the default is the SNMP trap port 162), or by connecting to an NMS product . The connection to the NMS is via a native interface using the Java Native Interface protocol. The connection to the NMS is dependent upon the types of APIs provided by the NMS for receiving event information.

The supported platforms are:

- HP OpenView on HP/UX
- HP OpenView on Solaris
- HP OpenView on NT
- NetView on AIX
- NetView on NT
- SunNet Manager
- MicroSoft Trap Service

Using the EDS Trap Receiver Framework

Upon startup, the Trap Receiver Framework loads a trap receptor that is responsible for connecting to an NMS and receiving the traps. The trap receptor is used to identify the source for the incoming traps. Once loaded, the trap receptor receives traps and passes them onto the main trap receiver component. The trap receiver then determines what actions to take upon receipt of the trap.

The Trap Receiver can receive up to 2,000 traps within 30 seconds and process them all within a 200-second window. In conjunction with the TrapToEDS action, the generated events can be produced at a rate of 10 events per second in sustained mode. Note that performance numbers will be gated by the rate at which the NMS provides the traps to the Trap Receiver Framework.

CISCO CONFIDENTIAL

The Trap Receiver ties into other standard EDS utilities, including:

- The standard debug/logging facility for logging of trace messages
- The CWCS Daemon Manager, for graceful startup/shutdown.

The following topics explain the Trap Receiver Framework:

- [Using the Trap Receptor](#)
- [Using the Trap Receiver Configuration File](#)
- [Using TrapInclude/TrapExclude Statements](#)
- [Creating Trap Actions](#)
- [Matching Trap Records](#)
- [Using the TrapToEDS Converter](#)
- [How the TrapToEDS Conversion Table is Used](#)
- [Using the TrapLaunch Action](#)
- [Using the TrapEcho Action](#)
- [Setting Trap Receiver Properties](#)

Using the Trap Receptor

The trap receptor provides the interface for communicating with specialized NMS products and receiving traps. The trap receptor is extensible, so other NMS products can be integrated at a later time. Specialized TrapReceptor classes are provided to communicate with many different types of trap sources (such as HP Openview on trap port 162).

Loading of the TrapReceptor often involves loading native libraries for interfacing to proprietary NMS trap APIs.

The TrapReceptor class must extend the following class:

```
public abstract class TrapReceptor {
public abstract void startListening(); // start the receptor
public void registerListener(TrapListener listener);
}
```

The TrapListener is an interface provided between the receptor and the remainder of the Trap Receiver Framework. The TrapReceptor, upon receiving a trap from the NMS (or any source), will make appropriate calls on the TrapListener to send the trap.

The TrapReceptor to be loaded is determined via a variable in the TrapReceiver's property file. Since loading trap receptors typically involves loading native libraries, any modification of the trap receptor requires a restart of the trap receiver.

The properties file can be configured to define the behaviour that the TrapReceptor should perform if the NMS is not running or has stopped sending events. The default is that the TrapReceptor will enter a retry/delay state while attempting to connect to the NMS. Once the retries have been exhausted, the TrapReceiver will terminate.

CISCO CONFIDENTIAL**Using the Trap Receiver Configuration File**

The trap receiver reads a configuration file that identifies the actions to be performed upon receipt of a trap. The format identifies the actions to be performed, and for those actions, the TrapBlock defining the traps to send to that action. A TrapBlock specification of “-“ indicates that all traps should be sent. Specifying a TrapBlock on an action line that does not exist will result in an error, and the action line will be discarded. [Example 20-1](#) shows a sample trap receiver configuration file.

Example 20-1 Sample Trap Receiver Configuration File

```
# Action> <ActionName> <TrapBlock> <Action Class> <Args>
#
Action TrapToEDS EDSTraps com.cisco.nm.cmf.eds.trap.TrapToEDS.class
Action MyApp MyAppTraps
com.cisco.nm.cmf.eds.trap.TrapLaunch.class/usr/bin/myApp $1 $2 $3
Action TrapEcho com.cisco.nm.cmf.eds.trap.Echo.class
#
# TrapBlock <ActionName> {
#   TrapInclude <trapName> <oid> <generic> <specific>
#   TrapExclude <trapName> <oid> <generic> <specific>
# }
#
# Traps to send to the TrapToEDS action
# This says to send all traps, but exclude any traps that matches an EDS trap.
TrapBlock EDSTraps {
TrapInclude AllTraps 1.3.6.1.4.1.* * *
TrapExclude EDSTrap 1.3.6.1.4.1.9.1.1.1.1 6 1
}
# traps to send to the MyApp action
TrapBlock MyAppTraps {
TrapInclude MyTrap .1.3.6.1.4.1.9.2.2.2.2 6 100
}
# All traps will be sent to the TrapEcho action because it has no TrapBlock specification.
```

Using TrapInclude/TrapExclude Statements

TrapInclude and TrapExclude statements in the Trap Receiver configuration file identify the traps to be passed to an action. For a trap to be passed to an action, the trap must satisfy the equation:

(Match any TrapInclude statement) AND NOT (Match any TrapExclude statement)

The configuration file format allows for easily extending the TrapReceiver’s actions without affecting other registered actions.

The order in which the TrapInclude and TrapExclude statements are provided in a trap block has no bearing on whether a received trap will match (it will match according to the above formula). However, it can affect performance. When a trap is received, it is compared against the TrapInclude statements in sequential order. When a TrapInclude statement matches, the remainder of the TrapInclude statements are ignored. After a match for a TrapInclude has been determined, a similar match in sequential order is performed against the TrapExclude statements. If a match is found for a TrapExclude, the trap is discarded, and no further matching is performed.

CISCO CONFIDENTIAL**Creating Trap Actions**

Actions are Java classes that perform specific functions based upon receipt of a trap. All actions must extend the following class:

```
public interface TrapBaseAction {
    public void processTrap(TrapPDU trap,
        String args[]);
}
```

When a trap is passed to an action, a list of translatable “dollar” variables, (\$), can be specified as arguments to the action. These variables are provided as a convenience for processing information contained within the trap. The specification for the variables is modeled after the \$-vars used in HP Open View. Strings that do not start with a \$, or that start with a \$ but do not match the list of variables shown in [Table 20-1](#) and [Table 20-2](#), are passed verbatim to the processTrap action command.

The notation \$n implies the Nth variable contained within the trap, where 1 represents the first variable contained within the trap.

Table 20-1 *Trap Information Variables*

Variable	Description
\$#	Number of variables in the variable binding list
\$*	All variable bindings in the format of: [1] name (type): value [2] name (type): value ...
\$n	The nth variable’s value, printed as a string
-\$n	Print the nth variable as: [n] name (type): value
+\$n	Print the nth variable as: name: value

Table 20-2 *Trap Header Information Variables*

Variable	Description
\$A	Print the Agent address from the trap as a hostname, if possible. Otherwise, print the IP address.
\$a	Print the Agent address as an IP address.
\$C	Print the community string contained within the trap.
\$E	Print the Enterprise OID as a translated oid name, if possible.
\$e	Print the Enterprise OID as a dotted decimal string.
\$G	Print the Generic trap number.
\$S	Print the Specific trap number.
\$T	Print the trap’s SysUptime timestamp.

At startup, the trap receiver creates a single instance of each action to process all trap requests; it *does not* create a new action object as each trap is received. A single action object will be created at startup, and the action’s processTrap() method will be invoked to process the trap. This facilitates actions, such as TrapToEDS, that need to open a persistent communication channel for the passing of event information.

CISCO CONFIDENTIAL

A TrapAction may be loaded multiple times if it is associated with unique actions in the configuration file. For instance, the TrapLaunch.class listed above provides unique, separate TrapLaunch functions for invoking separate commands.

Matching Trap Records

When a trap is received, it is compared against the TrapInclude and TrapExclude statements. For a trap to be passed to an action, the trap must match at least one TrapInclude statement, and it must *not* match any TrapExclude statements.

Wildcards (the '*' character), can be used at the end of an OID value, or in place of generic or specific trap numbers. If no wildcard is specified in the OID, then the OID must match exactly.

Using the TrapToEDS Converter

The TrapToEDS Converter action is an EDS Event Source that will convert an SNMP trap into an EDS event. A new event, called TrapEvent, will be created to contain the trap information. The TrapEvent will then be sent to EDS for distribution to any registered event consumers. The conversion of the trap to the TrapEvent is performed as shown in [Table 20-3](#).

Table 20-3 **TrapToEDS Conversion**

Event Attribute	Action
EventID	See the “How the TrapToEDS Conversion Table is Used” section on page 20-14
Event Category	See the “How the TrapToEDS Conversion Table is Used” section on page 20-14
Event Severity	See the “How the TrapToEDS Conversion Table is Used” section on page 20-14
ApplicationUID	“TrapToEDSConverter”
EventCreateTime	Compute current timestamp
Resource List	See the “How the TrapToEDS Conversion Table is Used” section on page 20-14
Event Data	{trap OID, generic number, specific number, sysUpTime, Community string, Varbinds}

How the TrapToEDS Conversion Table is Used

The mappings shown in [Table 20-4](#) equate traps generated by Cisco devices to appropriate EDS events. Each arriving trap will be matched against its OID, generic number, and specific number for a match into the lookup table. When found, the corresponding EventID, Event Category, and Event Severity will be retrieved from the table. Also contained within the table lookup will be a variable number of Resource Item records defined for each trap. These records will be used for the assignment of the ResourceList contained within the event. Each ResourceItem will consist of a 3-tuple that will contain:

- ResourceType
- Resource IDType
- Resource IDValue

CISCO CONFIDENTIAL

The Resource Type and Resource ID Type are integer values that map to atoms created with the atom service. The Resource ID Value is a string value that can be specified via a literal string, or the \$-variables, to retrieve information from the trap.

By default, every event has one resource item with a Resource Type equal to Device. If one is not specified in the table lookup, then one will be provided with the following assignments

- Resource Type = Device
- Resource ID Type = IP Address
- Resource ID Value = IP address contained within the trap

To further support complete control over the conversion of a trap into a specific event, an alternative format can be specified in the lookup table, where the name of a conversion class is specified. This class can be loaded and used for converting the trap to an event upon matching an OID, generic number and specific number. A single instance of this class will be created, and will be called for converting all traps to events that match its trap signature. This class must extend the following class:

```
public interface class TrapConverter {
public EventBase convertToEvent(TrapPDU trap);
}
```

If the trap is not found within the lookup table, then the following algorithm is used:

```
EventID:
Coldstart Trap:BaseAtomDef.Coldstart_Trap
Warmstart Trap:BaseAtomDef.Warmstart_Trap
Linkdown Trap:BaseAtomDef.LinkDown_Trap
Linkup Trap:BaseAtomDef.LinkUp_Trap
Auth Trap:BaseAtomDef.Authentication_Trap
EGP Neighbor:BaseAtomDef.EGPNeighborLoss_Trap
Enterprise Trap:BaseAtomDef.Enterprise_Trap
Event Category:BaseAtomDef.SNMPTrap
Event Severity:BaseAtomDef.EventSeverity_Informational
```

Table 20-4 TrapToEDS Mappings

Trap Name	OID Value	Generic number	Specific Number	Event ID	Event Category	Event Severity	Resource Item #1	Resource Item #n
Reload Trap	Cisco	6	0	ReloadTrap	SNMP Trap	Informational	3-tuple	3-tuple
TCP Conn Close	Cisco	6	1	TcpConnClose	SNMP Trap	Informational	3-tuple	3-tuple

Using the TrapLaunch Action

The TrapLaunch action allows launching a separate application based upon receipt of a trap. The first argument to the action will be the name of the application to be invoked. All of the arguments are passed on the command line to the application. The TrapLaunch action will be invoked a single time, (as are all trap actions). However, the nature of the TrapLaunch is to invoke a command upon receipt of a trap. Therefore, for each trap sent to the TrapLaunch action, the action will invoke a new command.

Using the TrapEcho Action

The TrapEcho action takes the incoming trap and sends it to a specified trap port. This allows for the operation of the TrapReceiver with other products that want to receive traps on a given port.

CISCO CONFIDENTIAL**Setting Trap Receiver Properties**

You can modify the Trap Receiver properties shown in [Table 20-5](#).

Table 20-5 **Settable Trap Receiver Properties**

To set the	Modify this property
Trap port for the UDPTrapReceptor to listen on	trap_port=162
Name of configuration file	trap_config_file=filename
Trap port for the TrapEcho command to send traps	trap_echo_port=5000
Class that identifies the TrapReceptor to be loaded	trap_receptor=com.cisco.nm.cmf.eds.trap.SomeTrapReceptor
Number of retries to perform for connecting to NMS, or UDP port	trap_receptor_connection_retry=10000
Number of seconds to delay between retries	trap_receptor_connection_delay=300

Using the Generic Consumer Framework

The Generic Consumer Framework (GCF) provides a mechanism for providing generic event-consumers with a pluggable interface for receiving events. Using the Named Filter API, you can register a set of event consumers with the GCF. Generic event consumers use a different named filter for the reception of their events.

While your application can produce a standard event consumer for receiving EDS events, doing so requires you to write a filter mechanism and a specification for the associated filter. Using the GCF allows you to write a generic event consumer that only needs to focus on event processing; the filter work is part of the Framework. This interface also allows you to easily update the filter to be associated with a consumer and the events that it should receive.

The following topics explain the GCF:

- [Using the GCF Configuration File](#)
- [Using the GCF Admin Display](#)
- [Creating Generic Consumers](#)
- [Using the Event to Trap Converter with Generic Consumers](#)

Using the GCF Configuration File

The GCF configuration file specifies the generic consumers and the named filters registered for each consumer. The GCF will register with EDS on behalf of the consumers and then pass the appropriate events to the consumers.

A GUI interface exists to aid in updating the TrapReceiver's configuration file. Also, an IDL interface to the TrapReceiver is provided for receiving, and updating named filter information.

```
#
# Sample Generic Consumer Framework Configuration file
#
# Action <ActionName> <NamedFilter> <Generic Event Consumer> <Args>
Action EventToTrap - com.cisco.nm.cmf.eds.trap.EventToTrap.class
```

The use of the string “-” as a named filter equates to a filter that indicates to receive all events.

CISCO CONFIDENTIAL**Using the GCF Admin Display**

The GCF administrative display permits configuring the named filters to be associated with generic consumers. It also displays the description of the selected filter. These features aid the administrator in choosing the correct named filter.

The GCF Admin Display exists as an HTML interface. Upon selecting appropriate information, the GCF Admin display interfaces to web servlets communicating on the backend for updating the GCF information.

Creating Generic Consumers

Generic consumers are consumers to be registered with the GCF. Generic consumers have no knowledge of the filters associated with them. After startup, the GCF passes events to those generic consumers whenever the event passes their associated filter.

All generic consumers must extend the following class:

```
public interface GenericEventConsumer{
    // Pass the event to the consumer.
    public void processEvent(EventBase event, String args[]);
}
```

When passed an event via the processEvent() method, a list of \$-translatable variables can be specified as arguments to the generic event consumer. These variables permit information to be obtained from the event without having to access the event object itself.

Using the Event to Trap Converter with Generic Consumers

The Event to Trap Converter (ETC) service plugs into the GCF, allowing generic consumers to receive EDS events. The named filter associated with the service determines the events that the EventToTrap service receives. Once the events have been converted into a trap, the traps will then be forwarded to a network management station via a destination hostname and port.

Upon startup, the ETC will read a startup file containing hostname and destination port settings for forwarding traps. If no port is specified, the standard SNMP trap port, 162, is assumed.

When an event passes the ETC filter, the event's toTrap() method is called to return a TrapClass object for the event. The TrapClass object supports all features needed to convert the event into a trap:

```
public abstract class TrapClass {
    public abstract String getEnterpriseOid();
    public abstract int getGenericTrapNumber();
    public abstract int getSpecificTrapNumber();
    public abstract String getCommunity();
    public abstract String getIPAddr();
    public abstract long getSysUpTime();
    public abstract SnmpVarbindList getVarbindList();
}
```

Upon receiving the TrapClass object, the ETC will call the appropriate methods to create the TRAP PDU and then send the PDU to the specified NMS hosts. By overriding the TrapClass object that is returned via the toTrap() method for an event, an event writer can define its own Event-To-Trap conversions.

In the case where an event does not provide a conversion, a DefaultTrapClass() converter object will be used. Your application can extend the DefaultTrapClass() if you want to modify some aspect of its functionality. The base DefaultTrapClass conversion produces the results shown in [Table 20-6](#).

CISCO CONFIDENTIAL**Table 20-6** *DefaultTrapClass Conversion Results*

Attribute	Value
OID	1.3.6.1.4.1.9.9.127.2.0
Generic trap number	6
Specific trap number	Critical event: 1 Major event: 2 Minor event: 3 Informational: 4
Agent Addr	Inspect resource list for an IP address, or Hostname, otherwise IP address of hostname where Converter resides.
Community string	Default from EDS property file
SysUpTime	Uptime for Event Consumer
Varbind 1	Event ID number
Varbind 2	Event ID name
Varbind 3	Event Category number
Varbind 4	Event Category name
Varbind 5	Event Created Time
Varbind 6	Event Sent Time
Varbind 7	Application name
Varbind 8	Event Class name
<Repeated>	Event Resource Information
<Repeated>	Unique Data name/value

The EventToTrap converter reads a property file containing various startup parameters. The property file consists of the following:

```
# list of hosts to receive trap information
gcf_trap_receivers=hostname:port,hostname:port

# community string to use within traps
community=public
```

Using EDS to Publish Events

The CWCS Job and Resource Manager (JRM) uses EDS to publish events of interest to CWCS-based applications. These events belong to the “status” event category (EventCategory_Status). The EDS event and resource atoms are listed in com.cisco.nm.cmf.jrm.JrmEdsAtomDev.java.


The following topics explain how EDS publishes events and how to receive them:

- [About the EDS-Published Event Types](#)
- [About the EDS-Published Severity Codes](#)
- [Registering Your Application with EDS](#)

CISCO CONFIDENTIAL**About the EDS-Published Event Types**

Table 20-7 summarizes the typical JRM-related event types published via EDS.

Table 20-7 *JRM Events Published Via EDS*

Event Type	Description
Job-related	<p>All job events are the instances of the com.cisco.nm.cmf.jrm.JobEvent class. The event's resource list always contains two resources: ResourceList_Job_Type and ResourceList_Job_Id. The event's unique data contains four members:</p> <ul style="list-style-type: none"> Job szProgress field at the time the event was recorded. <p> Note For the approve/reject events szProgress will contain an approver's comments.</p> <ul style="list-style-type: none"> Job run_state field. Return code (for EventJobEnd event). Approver name (for EventJobApprove and EventJobReject).
Lock-related	<p>Lock/unlock events do not have unique data. The resource list contains two resources:</p> <ul style="list-style-type: none"> ResourceList_Address_Hostname contains the resource name. ResourceList_Job_Id contains the job name.
Process- end	<p>Process end events are instances of the class com.cisco.nm.cmf.jrm.DaemonEndEvent. The resource list contains a single resource of type ResourceList_Job_Id. The event's unique data contains the return code and signal code of the process.</p>

About the EDS-Published Severity Codes

Table 20-8 summarizes the JRM event severity codes used with EDS-published events.

Table 20-8 *JRM Event Severity Codes Published Via EDS*

Severity Level	Codes
EventSeverity_Information	EventLock EventUnlock EventJobStart EventJobEnd (if completion code is RUNST_Succeeded) EventJobCancel EventJobApprove EventJobReject EventDaemonEnd
EventSeverity_Minor	EventJobEnd (if completion code is RUNST_SucceededWithInfo)
EventSeverity_Major	EventJobEnd (if completion code is RUNST_Failed) EventJobLaunchFailed

CISCO CONFIDENTIAL

Registering Your Application with EDS

To send and receive EDS events, you must subscribe to the events you want to view.

The `SampleEventConsumer.java` file, located in the `CodeSamples` directory on the CWCS SDK CD, shows how to subscribe to EDS events. This example shows:

- How to create an instance of a filter
- How to create the source code for a filter

It also displays two lists:

- The top list displays events received from one filter.
- The bottom list displays events received from a second filter.