Instrumentation for Linux Event Log Analysis

Rajarshi Das
Linux Technology Center
IBM India Software Lab
rajarshi@in.ibm.com

Hien Q Nguyen
Linux Technology Center
IBM Beaverton
hien@us.ibm.com

Abstract

Logging is inherent to a system which handles important data and performs important transactions and even otherwise. All the information generated right from the instant when the system boots up, to the instant when a device suddenly malfunctions, right upto the point in time when it shuts down, is important and is stored as part of the log. However, all this data that is generated needs to be formatted by an appropriate log analysis mechanism so that the right amount of information is made available to the consumers of such data. An implementation of POSIX event logging known as `evlog` is currently under development. Event Log Analysis is a log analysis mechanism that works on evlog and is also currently being developed. An object oriented model called the Common Information Model (CIM) has the ability to describe computing entities over enterprise environments. The paper looks into a solution based on the above model which uses instrumentation towards making the event log analysis mechanism more accessible to system administrators,with the common goal of keeping their systems failsafe as far as possible.

1 Introduction

Informational messages and actual events generated on a system from kernel subsystems and system management applications are logged to be analysed at a later point in time. However, with an increase in the volume and types of data getting logged, the requirement for a logging mechanism which records all the pertinent information that a system administrator or user would require, provides efficient log management, and runs on enterprise class systems as well, has resulted in linux® event logging. This provides notification capabilities to clients based on their interest in one or many types of events that are generated and logged in the system. The ability to generate reports based on the logged data was an added functionality that ensured that, if a certain consumer interested in a particular device wanted to be notified if that device sent out a certain number of messages in a second, a detailed report for the device would be generated as and when the mentioned condition was satisfied. Section 2 gives a brief outline of linux event logging and event logging analysis. Section 3 outlines the requirement in the context of event log analysis. Section 4 gives a glossary of CIM terms used in the context of the solution. Section 5 gives a brief overview of the concepts in CIM towards implementing a solution to the problem mentioned. Section 6 explains the solution per se, and also looks at features which have been/ havenot been implemented within/ outside current plan. Section 7Here we look at the tangible benefits of this solution which has been built on a CIM framework.

2 Linux Event Logging and Event Logging Analysis

In Linux, sysklogd which has been used as the standard logging facility, has several shortcomings like

1. events are recorded as text only and some pertinent information is not recorded.

2. a limited set of event facilities are provided. An implementation can define only eight of them, and

3. inability to selectively remove events from a log file based on user defined criteria. Linux Event Logging overcomes all of the above and a couple of other shortcomings as well, and is also suited to be used on large, multi-processor, enterprise-class systems. Linux Event Logging provides an interface for use by software to report events. The event logging system collects additional information such as time of the event, combines it with the interface supplied data and creates an event log entry.

A good posix based linux event logging mechanism needs to:

1. optionally take printk and syslog messages and log them as event records.

2. optionally suppress logging of duplicate events being logged in rapid succession, based on certain specified criteria, to avoid system performance degradation, and several other useful options.

Evlog which is an implementation of linux event logging, aims to provide an open-source, platform independent event logging facility for the linux operating system and linux applications [evlog].It conforms to proposed `POSIX Standard 1003.25`, System API - Reliable, Available and Serviceable systems. The POSIX standard is described at [evlog posix document].

Each client that wishes to be notified of certain events, needs to register for a certain notification.

Evlnotify is a command used to register actions to be taken when a specified event occurs. An action is executed on behalf of the client who registered it.

Evlnotifyd is a part of the logging system and serves as a daemon towards the basic registration mechanism for event notification, monitors the two logs (eventlog and privatelog) where event records are written, for events that match the notification criteria, and notifies the appropriate client(s) when there is a match.

Evlactiond is a client for evlnotifyd that reads entries from the persistent action registry (/var/evlog/action_registry) and registers with evlnotifyd to receive notification of the indicated events. When evlactiond receives a notification from the evlnotifyd daemon, it determines which notify_action (this is a part of every entry in the action registry file)to execute. Notify_action is a command line or script that is executed. Every client that has registered to be notified has an entry in the action_registry file.

A sample evlnotify command looks like this :

1. evlnotify –add "removeoldrecords.sh" –filter "facility=LOGMGMT".

2. This will register removeoldrecords.sh to be called when an event of facility type logmgmt is logged. The script may contain %recid% which will be replaced by matching record's id so that the script can be run for every matching record. Also, evlnotify doesnot wait for removeoldrecords.sh to complete. This may require removeoldrecords.sh to become reentrant according to requirements.

1. In case of Event Log Analysis, the evlnotify command would look something like evlnotify –add "make_a_new_ela_report %recid%" –filter 'facility=e1001' –threshold 10 –interval 60m.

2. This translates into "run 'make_a_new_ela_report' when 10 messages with facility type 'e1001' get logged in 60 minutes"

When a evlnotify command is issued, the following occur in sequence.

1. evlnotify sends a request to evlactiond
2. evlactiond registers for a notification with evlnotifyd
3. when a message with facility type e1001 arrives, evlnotifyd notifies evlactiond via signal
4. evlactiond keeps track of how many times the message has arrived and fires the registered action when 10 messages are received in the specified time interval of 60 minutes.

A event log analysis report generator writes reports to the location /var/evlog/ela_report.

3 Requirement

Linux Event Log Analysis writes reports to a specified location but a client mechanism is required to read this file and display the information to the end client who had originally asked to be notified. The requirement is that of a model which is already in place or use an existing model which can easily map the event log analysis mechanism and use the model to make the report available to the client.

Event notification is an integral part of the common information model itself and the model has well defined interfaces for an external program to make use of the notification process. Since this model has its own built in mechanism to handle event notification, it can be considered appropriate for providing a framework to present the linux system's event log analysis report to the client. The next few sections outline the information model from the event notification perspective.

4 Glossary

Client: A process that creates subscriptions on behalf of a client application.

Delivery: The process of transporting one or more Indications to an Indication Consumer. The delivery destination, encoding and transport protocol are defined as part of the definition of the subscription.

Event: An occurrence of a phenomenon of interest.

Filter: An instance of CIM_IndicationFilter that defines the set of Indications of interest to an Indication consumer.

Handler: An instance of CIM_IndicationHandler that describes the location, encoding and transport protocol of an Indication consumer.

Indication: The representation of the occurrence of an Event.

Indication Consumer: A process that consumes Indications.

Query: A description of the interesting characteristics of one or more Indications. A query is a component of a Filter.

Subscription: The process of registering to receive Indications.

Client application: This is the end user application which wants to subscribe to a certain event.

5 Concepts in CIM

The Common Information Model aims at describing computing and business entities over enterprise, and service provider enviroments. Event Logging Analysis in itself, is required to keep a history of events over a time interval and initiate some action when a threshold is exceeded. One of the actions that can be taken is to alert a system administrator via a event notification that an event of importance has occurred. The Common Information Model commonly referred to as CIM comprises of a specification which defines the details for integration with other management models and a schema which provides the actual model descriptions. The scope of the specification and the schema covers scenarios and use cases in which notifications based on events are used. CIM's ability to thus model event notifications in a simple-to-understand-and-implement way puts it as a natural choice to present a linux system's event log analysis data to the customers.

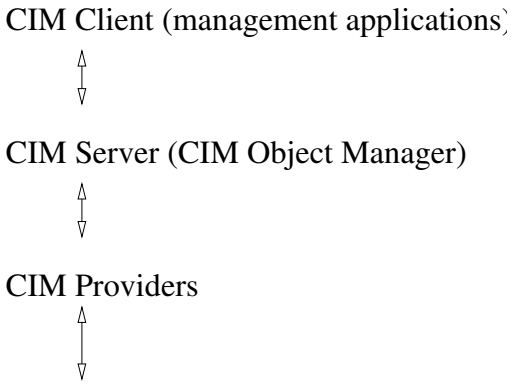
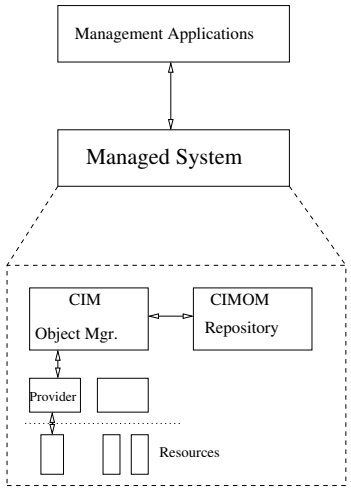


Figure 1: The three tiers of Web Based Enterprise Management



Note: 1) The CIMOM Repository contains the data model and static objects.
2) The management apps use the CIM API.

Figure 2: Where the provider interface fits in the big picture

5.1 WBEM

The Instrumentation for Linux Event Log Analysis is based on a Web Based Enterprise Management [WBEM] framework. WBEM includes CIM and the CIM schema and in addition to this, also covers CIM Operations over HTTP as a protocol allowing remote interoperable management in a distributed enviroment. WBEM has a three tiered architecture which is the one most commonly used. It is shown in Figure 1

Our solution is based on a a CIM Interface which is an executable or library that returns or sets information about a given managed object (which is essentially a mapping of the device in the real world). This interface is also known as a Provider Interface. It is written using CMPI (Common Manageability Programming Interface) hence is also called a CMPI Provider Interface. CMPI is a C API for writing the provider interfaces and is being standardized by the Open Group.

The architecture of a CIM provider interface could be shown as in Figure 2

Notes:

1. The CIM Server actually runs on the managed system and provides access to the system resources using the CIM Operations over HTTP protocol. The CIM server supports services such as event notification, remote access, and query processing. A possible implementation of a CIM server is a CIM Object Manager which uses provider interfaces to access resources.

2. The CIM Object Manager repository contains the data model and the static objects.

3. The provider interface communicates with the resources (managed objects) to access event notifications. The event notification information thus received is forwarded to the CIM server for integration and interpretation.

4. A management application uses APIs provided by CIM to connect to the CIM server and thus access the data generated by the provider interface.

5.2 CIM object managers

The WBEMsource Initiative has resulted in a number of CIM object managers being available to the community :

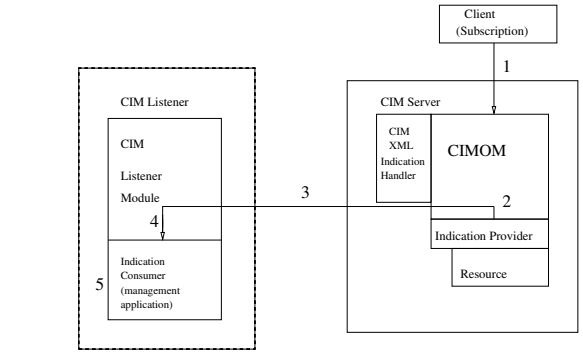
1. The Open Group^{TM}'s Pegasus^{TM} written in C++ (www.opengroup.org/pegasus)
2. The Open Group's Open Source CIM object manager (formerly Storage Network Industry Association (SNIA) CIM object manager) written in Java (www.opengroup.org/snia-cimom)
3. The OpenWBEM CIM object manager (http://openwbem.sourceforge.net)
4. The WBEM Services CIM object manager (http://wbemservices.sourceforge.net) written in Java^{TM}

There are also others like Microsoft Windows® Management Instrumentation which is an infrastructure, including a CIM object manager and a object manager repository. All the above offerings and more have resulted in a wide spectrum over which provider interfaces could be written and tested out.

5.3 CIM Event Model

Event notification in CIM terms, translates into having an *Indication provider interface* to retrieve the necessary information from the managed objects. An event is the occurrence of a phenomenon of interest and an indication being the recording of an event.

Once an application subscribes (requests) for information e.g. it is interested in being notified if a network error is registered on device 'netdev' twenty times within 5 minutes, this information is passed to the provider interface library which starts a separate thread that essentially waits for information that is related to device netdev. Once the mentioned condition is satisfied, that is e.g. twenty events have occurred mentioning 'network error' on netdev within 5 minutes, the event log analysis mechanism writes a report that contains information on the error that has occurred on netdev. This is in turn read by the concerned thread and delivers an indication to the application. The delivery of this indication is synonymous



Index :
1) Indication Subscription
2) Indication generation and processing
3) Indication delivery
4) Indication Receipt
5) Indication Consumption

Figure 3: The sequence of steps for an indication

with the application getting the notification. The indication might result in a popup message containing the desired information, or the app might send a mail to the concerned system administator asking him to take action. The sequence from the generation of an indication to the instant when it is delivered is shown in Figure 3

The subscription that is created by the CIM server once an app subscribes, actually translates into an instance of a class called CIM_IndicationSubscription. This class has two main members:

1. *CIM_IndicationFilter :* which has a property called 'Query' which describes the condition e.g. "select * from reportinstance where neterroriterations = '20' and neterrorinterval = '300'". This basically defines 'What to Send'.
2. *CIM_IndicationHandler :* this describes the indication consumer and the communication protocol. This answers the question 'How and Where to send'.

An indication provider interface needs to implement the following CIM interfaces:

1. *MustPoll :* This interface is for the CIM server to determine if it can poll a provider. This is the

first to be called once a client application asks for a notification.

2. *AuthorizeFilter :* This interface needs to be implemented in case we need to ensure that the user wanting to subscribe is suitably authorized. This gets executed after MustPoll.

3. *ActivateFilter :* This is invoked by the CIM server to ask the provider interface to look for events. This gets called after AuthorizeFilter.

4. *DeactivateFilter :* This interface is called when the user (subscriber) decides that he doesnt need any more relevant notifications.

The first three interfaces are called when a client asks for a notification.

6 The Event Log Analysis provider

Instrumentation for linux event log analysis translates into an implementation of a `CIM provider interface for linux event log analysis`. This has been referred here as the event log analysis provider. This is in itself a single provider interface[1]which is used to delivers complete ela reports to the subscribing client application as and when the condition matches. Currently, a client application gets notification on all event log analysis reports that are written by the ela mechanism irrespective of the condition specified.

6.1 Design Issues

The common information model offered two types of classes on the basis of which the indication provider interface could be implemented. They are:

1. *Lifecycle Indication classes* These classes are used whenever the information flow being modelled can be correlated to a life cycle event. It could represent class creation e.g. CIM_InstCreation or class deletion like CIM_InstDeletion.

2. *Process Indication classes* This class is used to represent the occurrence of a phenomenon of interest or in other words an event and especially alert notifications associated with objects that may/ maynot be completely modelled in CIM and cannot be represented using the lifecycle indication classes. e.g. CIM_ProcessIndication.

For the purposes of the event log analysis provider, a class had to be used which could model the delivery of indications (refer section 4). This class would also have to contain the information(the contents of the event log analysis report) that was to be shown to the client. The class was called `ela_reportindication` which was derived from the class CIM_ProcessIndication, since an event was central to the model being designed.

Also, there were a couple of mechanisms that were considered towards implementing the interaction between the provider library and the daemon (refer figure 4).

1. The ActivateFilter function in the provider library (refer subsection 5.3) could spawn a thread which would wait for a signal from the daemon. As soon as the e event log analysis report generator (refer figure 4) wrote a report, it would signal the daemon, which in turn would signal the waiting thread which in turn would read the event log analysis report and deliver an indication.

2. ActivateFilter would include a call to fork a child from the daemon. The child would get the signal from the report generator, read the report and then deliver an indication.

3. ActivateFilter would spawn a thread which would wait for data on a socket that the daemon had opened. On receiving a signal from the report generator, the daemon would write the record id of the event log analysis report to the socket. The thread would read the report and deliver an indication.

[1]There can be more than one type of provider interface at a time. In this case, it is a single indication provider interface

The first option had a slight overhead in the daemon not knowing that there was a thread waiting for information from it. So the daemon would have to explicitly figure out a way to send the recid of the report to the waiting thread. The second option would require ActivateFilter to be able to fork a child of the daemon (which is a separate running process). The child would get a signal from the report generator but then, it would have to be able to call the CIM interface call to deliver an indication. Basically, this points to a possibility of calling CIM functions from within the daemon, which was thought to be a cumbersome process. The third option was a modification of the first, with the "ActivateFilter thread - daemon" interaction being considered to be happening over a socket. Irrespective of whether the client application was remotely started or locally, the provider library (and the CIM server), the daemon and the report generator had to reside on the same physical machine. The information flow between "ActivateFilter" and the daemon had to happen on the same machine, thus unix domain sockets were chosen as the socket mechanism. TCPIP sockets were avoided and all the overhead that came with it. The following section details the third option and how it was used to implement the solution.

6.2 Implementation Details

With reference to the previous section, the MustPoll and the AuthorizeFilter interfaces have not been implemented. The others have been implemented as mentioned below:

1. *ActivateFilter :* A subscription is added with the filter expression and the handler as specified by the subscriber. The required process indication class is loaded here.

2. *DeactivateFilter :* Check if the number of subscriptions is null and if it is not, decrement. Finally, the number of subscriptions should be zero.

The indication that is delivered is actually an instance of a class called `ela_reportindication` which

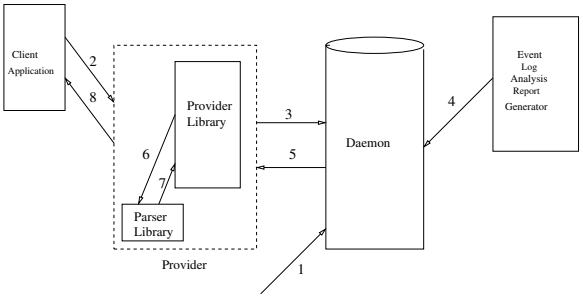


Figure 4: The Event Log Analysis provider model

is derived from a base CIM class (which is traditionally used to represent indications relating to processes in general.

The provider in essence comprises of four components.

1. The Client Application

2. The Provider library which is dependent on a parser library

3. The Daemon and

4. The Event Log Analysis report generator.

The process of a client application subscribing for an event, right upto the client application receiving an indication, in the current solution is shown in Figure 4 Also, the numbers shown against the arrows in the figure actually correspond to the steps as outlined below.

The sequence:

1. The Daemon is started. It initialises itself and listens for connections from any client that wishes to connect. Then the CIM server is started (this is not shown in the figure).

2. The client application sends a request to be notified when certain conditions occur. This is currently a script which connects to the CIM server. The 'IndicationFilter information' is passed to the CIM server. This results in a new instance of

CIM_IndicationSubscription being created. As of now, the client asks for a generic notification where it is notified as soon as there is any ela report that is created.

3. This request results in the ActivateFilter function getting called, where a thread is spawned which connects to the daemon and waits for data. When the thread connects to the daemon, the daemon in turn forks a child for this new client connection and continues to wait for new clients to connect.

4. Once the ela report generator writes a report to the desired location, it sends a signal to the daemon and sends the recid of the newly written report as part of the signal to the daemon.

5. The child of the daemon process (responsible for the client connection) receives the signal, retrieves the recid from the signal, and writes this recid to a socket.

6. The waiting thread in the provider library reads this recid, constructs the filename of the ela report from this recid, and sends this filename to the parser.

7. The parser in turn, opens the file with the sent filename, reads the fields and fills up a structure with these fields.

8. The thread in the provider library now sets the properties of the ela_reportindication class, with the fields that it reads from the structure, and delivers an indication. This results in a popup window at the client application containing all the information that was present in he ela report and in addition to that the hostname of the system as well (which we are interested in and on which the provider is running).

The client application used here has been downloaded from the Standards Based Linux Instrumentation for Manageability (SBLIM) site [SBLIM] and suitably modified to suit the purpose.

6.3 Current State

1. The current solution ensures that a single client application can subscribe at a time.

2. The client application can receive indications as long as it desires, and can exit whenever it doesnot intend to receive any more indications.

3. If the CIM server is started without starting the daemon first, it will fail to do so and will exit with a message indicating the sequence to be followed for the server to be able to start again.

4. The solution has currently been tested on the SNIA CIM server alone.

5. The first code drop for this solution is due shortly when it is to be hosted on the evlog site (http://evlog.sourceforge.net).

6.4 Features in scope, to be implemented

1. Multiple client applications should be able to subscribe to the same event. The Ela provider design allows for multiple client applications to subscribe. The client code that is being used doesnot allow multiple client applications to subscribe for the same event as of now. Once the code is modified as required, it will be tested on SNIA which is one of the many CIM object managers available under the WBEMSource initiative (www.opengroup.org/wbemsource) The Ela provider needs to be tested to work on Pegasus as well.

2. If the parent process (the daemon) receives a signal to terminate either accidentally or otherwise, it should be able to terminate all its children before it terminates.

3. Also, if the daemon gets killed accidentally, the cimom server should be able to close the connection and exit gracefully (Note below).

4. The DeactivateFilter interface needs to be implemented. This should be called once the client

application dies, and should be capable of deleting the existing subscription. Also, once the client application (the runit.sh script in evsub) dies, the child (of the daemon) responsible for servicing this client application should be able to exit without user intervention (unlike manually killing the children as per current implementation). Whether the DeactivateFilter code (if called by the exiting client application) can cause the corresponding child (of the daemon) to exit in such a scenario, needs to be investigated.

Note : The daemon needs to be started first, followed by the cimom server and finally the client application which intends to receive information. Once a client application has received the desired information, during cleanup, the client application should be killed first, followed by the cimom server and finally the daemon. Also, any existing subscriptions need to be deleted before restarting the cimom server (if the cimom server needs to be started without starting the daemon first) else the server will not start. To delete the subscriptions, the daemon has to be restarted, then the server. After the subscriptions have been deleted, the cimom server can be restarted.

6.5 Nice to have, not in current scope

1. The client application as mentioned previously, should be able to request for information e.g. being notified if a network error is registered on device 'netdev' twenty times within 5 minutes, and get the specific information. As of now, the client application gets information pertaining to netdev, and all other devices pertaining to which the event log analysis mechanism writes reports.

7 Conclusions

Usage of a CIM based framework ensures

1. faster adoption in the industry
2. also allows for remote management capability which could be used in scenarios where mission critical servers need to be monitored regularly from failure condition
3. that arbitrary methods can be applied on objects, apart from the normal set and get operations
4. a beneficial integrated data/event model
5. uniform base management for different platforms, and architectures while allowing a architecture or platform specific instrumentation as well
6. it is easy to extend the existing models to accomodate complex relationships and hierarchies and avoid cross referenced data tables to depict dependencies, component connection associations.

An event log analysis provider ensures that each and every device that is not supposed to have minimum critical downtime can be constantly monitored so that the interested party can be informed by means of mail or popup messages or other mechanisms so that appropriate action can be taken. IBM^{TM} Director^{TM} is a product which is used for intelligent systems management, and instrumentation for linux event log analysis could prove useful to its hardware event monitor towards monitoring errors on all IO adapters. This apart, all other systems management applications which are interested in consuming data based on the common information model are potential customers of this solution.

Increased acceptance of the common information model as a reliable, easy-to-setup-and-maintain framework towards interfacing systems management applications (and their requirements) would ensure greater availability of such applications to a wider customer base.

8 Acknowledgements

We are extremely grateful to Viktor Mihajlovski for assisting us in developing working knowledge on the Common Information Model. We owe our thanks to Dipankar Sarma and R. Sharada for their support.

References

[evlog] The event logging website. A full description of the features available and the ongoing work is available here. http://evlog.sf.net

[evlog posix document] A description of the posix standard adopted while developing evlog http://evlog.sf.net/posix_rationale.html

[WBEM] The website for Web Based Enterprise Management. http://www.dmtf.org/standards/wbem

[SBLIM] Standards Based Linux Instrumentation for Manageability. http://oss.software.ibm.com/sblim

[Indication concepts] The CIM Indications whitepaper. http://www.dmtf.org/standards/published_documents

[CIM Tutorial] A comprehensive tutorial on the Common Information Model http://www.wbemsolutions.com/tutorials/CIM

9 Trademarks