

R E M A Y S

Remote Event Monitor and AnaLYsis System

Oscar Medina Duarte
www.medina-web.com

Introducción

REMAYS esta basado en la especificación de REMAS¹, el objetivo de este sistema de monitoreo, es tener la capacidad de diagnosticar problemas en *tiempo real*² y que a su vez en los casos para los que los problemas son fáciles de resolver estos sean resueltos de manera eficiente e incluso remota.

Esta nueva especificación tiene una orientación modular en código objeto, es decir, que todo el sistema esta escrito por partes independientes de código, cuya función es bien definida y predecible, de este modo, es más fácil hacer adaptaciones del sistema, sin tener que recompilar todo el código además de que los módulos pueden estar escritos en diferentes lenguajes e incluso en algunos casos funcionar de manera distribuida. La comunicación entre módulos es independiente de la plataforma, ya que usa sockets TCP conectados.

Usos

REMAYS puede ser usado para monitorear cualquier cosa, con solo algunas modificaciones a diferencia de REMAS que fue diseñado específicamente para funcionar como un IDS (Intrusion Detection System) o Sistema de Detección de Intrusos. Los ambientes en los que REMAYS puede ser útil es en los que se generan demasiados datos de distintos aspectos relacionados entre si, como una línea de producción, un experimento científico o claro, un SDI.

¿Que hace REMAYS ?

El flujo inicia en un grupo de sensores que generan datos para que REMAYS los procese, estos sensores, deben estar conectados a una red TCP/IP con capacidad para comunicarse con un sistema REMAYS, imaginemos una planta de energía geotérmica donde todos los sensores están conectados con un modulo REMAYS, es evidente que es de vital importancia saber como se comporta la planta en cuanto a presión, temperatura, velocidad de las turbinas, etc... para cada uno de estos aspectos, será necesario tener un sensor conectado a la misma red. Cuando los sensores generan eventos, estos son recibidos por un REMAYS, donde pueden ser analizados por un componente especialmente diseñado para ello, este componente debe tener la capacidad de reaccionar y tomar cartas en el asunto, ya sea ejecutando un proceso o llamando la atención de la persona responsable de ponerle atención al suceso.

Ver diagrama : <http://mailweb.udlap.mx/~is111936/oto02/cursos/is332/diagramays.ps>
(Al final de este documento)

1

2 En realidad el tiempo real es efectivo solo para los casos más simples, en los demás casos solo pretende serlo, ya que para ambientes muy complejos podrían llegar a necesitarse recursos muy grandes para actuar en tiempo real, además de unas condiciones optimas de la red.

Análisis

Este modulo puede ser diseñado de muchas maneras, algunas más experimentales que otras, y así mismo algunas más confiables y/o efectivas que otras, mi propuesta para el modulo por defecto, es un sistema de inferencias basado en reglas que examine las características y propiedades de los eventos actuales y su relación con los ya almacenados en un tiempo definido, en el caso de que una regla se dispare, esta podrá realizar una de varias acciones, las acciones estarán definidas como funciones en un archivo separado, pero para este prototipo, la única acción configurada por defecto será echo(), que imprimirá en la salida estándar sus parámetros.

Si seguimos con el ejemplo de la planta de energía geotérmica podríamos tener que si un sensor de temperatura sube más allá de los 90° se abra la válvula de presión correspondiente a la misma, y si por ejemplo, la válvula se atasca, enviamos un mail al gerente de mantenimiento de esa zona de la planta.

Para este caso, lo eventos que serian generados en español serian :

Temperatura de sensor mayor a 90.
Válvula atascada.

Para poder hacer lo que describimos anteriormente, necesitamos, en primer lugar, tener un registro de los eventos que han ocurrido en un lapso determinado tiempo anterior a cada evento, el formato del archivo para almacenar estos eventos es :

```
{Nombre, Clasificación, Origen, {Propiedades}}
```

Donde :

Nombre.- Es un el nombre del evento que lo identifica de manera intuitiva dentro del sistema, si en el ejemplo anterior la temperatura fuera para el conducto de gas principal un buen nombre podría ser algo así como main_tube_temp.

Clasificación .- Este es de la forma genérica NombreProceso:ProcessID, el nombre de estos campos es derivado de las raíces de la especificación del sistema como un IDS³, pero como su uso puede ser modificado, en un caso como el del ejemplo podría ser útil asignar para este campo un nombre genérico junto con un numero serial que lleve el conteo del uso de este sensor, lo que podría ser usado también para llevar un registro del comportamiento de ese tubo.

Origen .- Este campo es una dirección IP en notación de punto pe: 192.168.0.9

{Propiedades} .- Es una lista separada por comas de las propiedades del generador del evento, el orden de estas es importante por que el sistema de reglas trata estos datos de acuerdo a su posición en el arreglo.

Estos dos eventos, entonces tendrían una apariencia similar a la siguiente.

```
{tube_temp , maintube:3 , 192.168.0.9, {25/12/2002, 11:23:33, 102}}  
{tube_stock , maintube:1, 192.168.0.9, {25/12/2002, 11:24:04, 20}}
```

El siguiente paso para poder analizar el evento es tener las reglas que relacionen los eventos entre si, unas reglas adecuadas para ello podrían ser como las siguientes.

Si la temperatura sobrepasa los 90°, abrir válvula correspondiente.
Si la válvula se atasca, avisar al gerente de mantenimiento

3 En un IDS, el NombreProceso es el nombre del programa que genero el evento como /usr/sbin/su y el ProcessID es el numero de proceso que identifica al proceso creador del evento dentro del sistema.

Si la válvula se atasca en menos del 20 % de su apertura total y la temperatura sobrepasa los 90°, evacuar la planta.

Para definir estas reglas, necesitamos definir su forma general, para esta propuesta, el formato es :

```
nombre_del_trigger {  
    (expresion);  
    step = N;  
    funcion0({parametros});  
    ...  
    funcionn({parametros});  
}
```

Donde :

nombre_del_trigger .- Es el nombre identificador de la regla.

(expresión) .- Esta es la regla que determina si la acción de la regla es llevada a cabo o no. Esta es una expresión lógica/booleana que relaciona detalles del ambiente con detalles del evento actual con detalles de eventos ocurridos con anterioridad.

Operadores :

Aritméticos :

+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo

Lógicos :

^	Xor, Bit por Bit
	or, Bit por Bit
	or, Lógico
&	And, Bit por Bit
&&	And, Lógico
==	igual a
!=	diferente de
>	mayor que
<	menor que
>= , =>	mayor o igual a
<= , =<	menor o igual a
!	Negación lógica
eq	Igual a, comparador entre Cadenas
ne	Diferente de, comparador entre Cadenas
gt	mayor que, comparador entre Cadenas
lt	menor que, comparador entre Cadenas
ge	mayor o igual a, comparador entre Cadenas
le	menor o igual a, comparador entre Cadenas

step .- esta es una variable a la que se le sumara un numero especificado cada que se dispare

el evento, este puede ser cualquier entero, incluyendo el cero.

```
funcion(Param0, Param1, ... , ParamN);
```

Las siguientes, son funciones, que estarán configuradas desde un archivo ejecutable externo, estas pueden o no estar presentes.

Los argumentos serán variables de acuerdo al archivo de configuración que tendrá el siguiente formato:

```
path:nombre:numeroargs
```

Donde

path - es la ruta del programa que contiene la función.

Nombre - es el nombre con el que se va a identificar la función dentro del archivo de reglas.

numeroargs - es el numero de argumentos que se le van a pasar.

Para este prototipo, la parte de configuración de funciones no estará presente, por lo que se definirá una función solamente, `echo("")`, la cual imprimirá su parámetro único a la salida estándar.

Las entradas que corresponden a las reglas que mencionamos antes se verían como esto :

```
hi_temp {
    ((e.name = tube_temp)^(e.2 > 90));
    step = 1;
    abrir(e.class.name, e.class.pid);
}

vlv_stock {
    (e.name = tube_stock);
    step = 1;
    smpt("mant@plantageo.com.mx", "Subject: válvula atascada " ,
        e.name, e.class, e.source);
}

vlv_stock {
    ((e.name = tube_stock)^(e.2 <= 20)^(hi_temp.origen = e.origen));
    step = 1;
    alarm("Alarma la de tos !!!");
    smpt("mant@plantageo.com.mx", "Subject: Evacuar la Planta " ,
        e.name, e.class, e.source);
}
```

Con esto ya tenemos suficiente información para poder responder de la manera especificada anteriormente.

A continuación presento un borrador del BNF para estos lenguajes.

Base de eventos

```
<Eventos>
<Eventos> ::= <Evento> | <Evento><Eventos>
<Evento> ::= {<Nombre>, <Clasificacion>, <Origen>, {<Propiedades>}}
<Nombre> ::= <Cadena> // OJO !!!
<Cadena> ::= [\32-\126]+
<Clasificacion> ::= <Cadena>:<Numero>
<Numero> ::= -[0-9]+|[0-9]+
<Origen> ::= <Octet>.<Octet>.<Octet>.<Octet>
<Octet> ::= <Numero>
<Propiedades> ::= <Propiedad> | <Propiedad>, <Propiedades>
<Propiedad> ::= <Numero> | "<Cadena>"
```

Gramática Libre de Contexto

Sea un <Eventos> reconocido por un GLC denominado $Ev = (N, E, P, S)$

Donde

N es el conjunto de los no terminales

```
N = {<Eventos>, <Evento>, <Nombre>, <Cadena>, <Clasificacion>,
      <Numero>, <Origen>, <Octet>, <Propiedades>, <Propiedad>}
```

E es el conjunto de los terminales

```
E = {\", [0-9], [\32-\126], '{', '}', '.', ',', '}'
```

P sean las producciones

S es el símbolo inicial

S = <Eventos>

Base de reglas

Gramática BNF :

```
<Triggers>
<Triggers> ::= <Trigger> | <Trigger><Triggers>
<Trigger> ::= <Nombre> {<Expresion>; <Step>; <Funciones>}
<Nombre> ::= <Cadena> // Ojo aca... ser mas especificos en lex/yacc
<Cadena> ::= [\32 - \126]+
<Expresion> ::= (<Termino>) | (<Termino><Op><Expresion>)
<Termino> ::= <Numero> | <Boolean> | <ObjProp>
<ObjProp> ::=
    | e.name | e.class | e.source | e.class.name | e.class.pid |
e.<Numero>
    | c.name | c.class | c.source | c.class.name | c.class.pid |
c.<Numero>
<Op> ::= <AritOP> | <BoolOP>
```

```

<AritOP> ::= + | - | * | / | %
<BoolOP> ::= ^ | '|' | '||' | & | && | == | != | > | < | >= | => | <= |
=< | ! | eq | ne | gt | lt | ge | le
<Numero> ::= -[0-9]+ | [0-9]+
<Boolean> ::= <True> | <False>
<True> ::= true | 1
<False> ::= false | 0 | -1
<Step> ::= step=<Numero> | step\32=<Numero>
<Funciones> ::= <Funcion> | <Funcion><Funciones>
<Funcion> ::= <Fnombre>(<Parametros>);
<Fnombre> ::= <Cadena>
<Parametros> ::= " | <Parametro> | <Parametro>, <Parametros>
<Parametro> ::= <Numero> | <String> | <ObjProp>
<String> ::= "<Cadena>" // OJO !!!

```

Gramática Libre de Contexto

Pertenezca un Triggers a un GLC denotado por $T = (N,E,P,S)$

Donde

N es el conjunto de los símbolos no terminales tenemos que

```

N = { <Triggers>, <Trigger>, <Nombre>, <Expresion>, <Termino>, <Op>
      , <AritOP>, <BoolOP>, <Numero>, <Boolean>, <True>
      , <False>, <Funcion>, <Funciones>, <Parametros>, <Parametro>
      , <Cadena>, <ObjProp> }

```

E es el conjunto de los símbolos terminales tenemos que

```

E = { '{' , '}' , ';' , step=, step\32=, e.name, e.class, e.source
      , e.class.name, e.class.pid, e., c.name, c.class, c.source
      , c.class.name, c.class.pid, c., + , - , * , / , % , ^ , | , ||
      , & , && , == , != , > , < , >= , => , <= , <= , ! , eq , ne , gt , lt , ge , le
      , [0-9] , [\32-\126] , '(' , ')' , false , true , ',' }

```

P sean las producciones

S símbolo inicial S e N

S = <Triggers>