

AN EFFICIENT ALGORITHM FOR ROOT CAUSE ANALYSIS

Krishanu Bhattacharya¹, N.Usha Rani²

NMSWorks Software Limited, Chennai – 600 036

Email: {krishanu_usha@nmsworks.co.in}

Timothy A.Gonsalves³, Hema A. Murthy⁴

TeNeT Group, Dept of CS & E, IIT Madras, Chennai – 600 036

Email: {tag_hema@tenet.res.in}

Abstract

A common problem in the management of large networks is that a single fault manifests as multiple alarms in the Network Management System (NMS). This makes manual analysis of faults costly and also diverts attention from really major problems. This is a major motivation for applying analysis and correlation in NMS. In this paper, we present a rule based analysis engine for doing Root Cause Analysis. The general approach is discussed and the implementation of a rule that is applicable to wide range of situations is described in detail.

1. Introduction

One critical management functionality for an NMS (Network Management System) is fault management. An NMS detects faults in networks in two ways. First, it periodically polls the element for its status. Second, an NMS can be configured to receive traps or notifications (from specified network elements) that are spontaneously sent when fault conditions occur. [1]

A NMS polls the status of every element in a managed network after certain time interval (called polling interval). The status of an element can be either *Up* or *Down*. If after a poll, the NMS finds that the status of an element has changed (either *Up* to *Down* or *Down* to *Up*) it generates an alarm. A transition from *Up* to *Down* is registered as an alarm and a transition from *Down* to *Up* clears the alarm. We assume a centralized NMS that monitors the whole network from a single location. We call the interface between the NMS and the host where it is running as the ‘NMS interface’.

A typical managed network has a large number of elements and often a large number of fault conditions or abnormalities occur. A single fault in the network may be manifested as multiple alarms in the NMS. These can result in tens of thousands of alarms getting generated. For example, when a switching element, such as a router in a network goes *Down*, the NMS will generate alarms for the router and also for all the elements which were

accessible only through the router. Making sense of all the alarms and identifying significant faults can become tedious for the human operator/administrator. It is important to pinpoint the *root cause* for all the alarms by localizing the fault and generating an alarm only for the faulty elements, as it will reduce the volume of information thrown to the network manager from NMS. Alarm correlation, to find root cause, is useful in a situation like this.

1.1 Organization of the Paper

This paper is organized in six sections. The next section describes some common techniques for Root Cause Analysis (RCA). The third section illustrates the algorithm developed by us for root cause analysis. The fourth section addresses some general issues about the proper time for applying the analysis in a real network. The fifth section presents a case study where the technique. We conclude in Section 6.

2. Some Common Techniques for RCA

Correlation may be done at several levels - starting from the level of individual network element up to a level involving the entire network [2]. Some commonly employed techniques at a lower level are mentioned below. The technique "Compression" involves defining a time window and observing all the alarms that occur within it. Multiple occurrences of Alarms due to the same event are replaced by a single alarm. "Suppression" [3] of alarms is a technique that inhibits generation of alarms corresponding to an event based on some criterion. For example, if a certain interface is in an unstable condition and keeps going *up* and *down*, the resulting alarms can be suppressed. Likewise, it is possible to generate an alarm only when a certain event occurs a particular number of times, i.e. exceeds a certain count.

There are various approaches to Alarm correlation at a higher level in the literature. Of these, a significant

number are variants of the rule-based approach. In this approach the knowledge of a certain area is contained in a set of if-then rules (stored in a database). A rule consists of two expressions linked by an 'implication' (\Rightarrow) connective, that links prerequisite and the action to be taken.

Another common approach is Model Based Reasoning. Here, we represent the system by a structural model including network description, topology and a functional model that describes the process of event propagation. Correlation can also be done based on Case Based Reasoning [4]. In this case relevant aspects of past episodes are stored. These are retrieved, adapted and used to solve new problems. ANN (Artificial Neural Network) based techniques are also used for RCA.

Several commercial NMS products have a root cause analysis system which uses variants of the approaches outlined. These include HP OpenView [5], IBM Tivoli [6], and Cabletron Spectrum [7]. A set of rules can be established either topologically or functionally to correlate observed events to isolate and localize the fault(s).

3. Our Algorithm for Root Cause Analysis

3.1 Overview

Our algorithm is based on a dependency relation between network elements. The criterion we consider here is reachability from the NMS. The status of a network element 'a' depends on the status of a network element 'b' if 'b' can be reached only via 'a'. In this situation if element 'a' goes *Down*, in the absence of correlation, the NMS recognizes element 'b' also as *Down*. This technique relies on the topology of the network being managed. The topology of the network is represented as a graph. Given this topology and the status of the various nodes (as returned by a status poll of all the elements), an efficient algorithm is proposed to determine the elements which are actually *Down*. This results in the generation of alarms only for those which are known to be *Down*. The status of its dependents is declared as *Unknown* as the NMS cannot reliably determine their states.

An advantage of this technique is that it makes use of the topology information that is already present in a typical NMS. The NMS need not collect or store any additional information just for RCA. This is a significant advantage in terms of storage, bandwidth overhead and processing in the NMS.

3.2 The Algorithm

The algorithm has two parts. The first part forms a reachability graph from the physical topology. It also takes the full list of status events from the NMS as its input. From this list, it determines the status of individual elements in the network. After the execution of the first part, the output is the real status of each element in the network. The various real statuses can be *Up*, *Down* and *Unknown*. An element with real status *Down* is the root cause for itself (and an alarm will be generated for this element).¹

The second part assigns a particular *Unknown* node to a set of *Down* node(s) where the *Down* contour is the root cause(s) for the particular *Unknown* element. That is, given any *Unknown* node 'a', it returns the list of *Down* nodes $\{b \mid a \text{ depends on } b \text{ is true}\}$. Conversely, for a particular *Down* node it returns a list of *Unknown* nodes where the *Down* node has caused all those nodes to be *Unknown*, i.e. for a given node it lists out all nodes that depend on it in that particular network situation.

3.2.1. Part 1 of the algorithm

Input Specification

1. The physical topology information is available as an n-node undirected graph represented as an adjacency list, adjList [8]. Specifically, all the nodes of the managed network are numbered from 1 to n. The adjacency list (adjList) is an array (size n) of linked lists where index i of the array contains the linked list of all the nodes directly connected by a network link to node i. Hence, adjList(i) is a linked list which contains all the nodes directly connected to node i by one link.

2. The NMS interface is directly connected to a single known node, say k.

3. The status table (statTable) is available as a convenient data structure which simply has two columns viz. 'Node Number' (possible values belongs to (1,2,..., n)) and 'Status' (possible values belongs to (*Up*,*Down*)). We assume the presence of a look-up function statTable.getStatus(i) which returns the status of node i for $1 \leq i \leq n$.

Data Structures

1. boolean stat[n] - status of each node.
2. boolean reachable[n] - reachability of every node from the NMS node.
3. boolean unknown[n] - the final result, an entry is 1 means it is having state *Unknown*.

¹ We have retained the option of generating alarms for elements whose status is returned as *Unknown*.

4. Declare a boolean array added[n]. This keeps track of what all elements has already been added in the list L(in declaration 5)

5. list L of nodes - temporary buffer of nodes. L has four methods:

- i) L.create() which creates an empty list L.
- ii) L.delete() which returns the first element from L and deletes it from the list L.
- iii) L.insert(i) which puts the element i into the end of the list and
- iv) L.isEmpty() returns true if L is empty else returns false.

Logic

Step 1:

```
for i = 1 to n
    stat[i] = statTable.getStatus(i);
```

Step 2:

```
for i = 1 to n
    {
        reachable[i] = 0;
        unknown[i] = 0;
        added[i] = 0;
    }
    L.create();
```

Step 3:

```
reachable[k] = 1;
if(stat[k] == down) go to Step 5;
else
    {
        // k is the NMS node. L is already
        // initialized.

        L.insert(k);
        reachable[k] = 1;
        added[k] = 1;
    }
```

Step 4:

```
while(L.isEmpty == false)
    {
        tempNode = L.delete();

        for all i in the linked list
        adjList(tempNode)
        {
            reachable[i] = 1;
            if((stat[i] == up) and
                (added[i] == 0))
            {
                L.insert(i);
                added[i] = 1;
            }
        }
    }
// end of for
//end of while
```

Step 5:

```
for i = 1 to n
    {
        if( (stat[i] == down and
            reachable[i] == 0 )
            unknown[i] = 1;
    }
```

Output Specification

For each *Down* node i for which no alarm should be generated set unknown[i] = 1.

Time Complexity

The costliest step of the algorithm is Step 4. Each time, for a node the inner loop runs once for all the edges from the node. So, the complexity of this loop is $O(d)$, where d = the average degree of a node in the topology graph.

The outer while loop runs once for every *Up* node reachable from the NMS interface. At worst the count of that = n , when all nodes are *Up*. The worst case complexity of the outer loop is $O(n)$.

Hence, the worst case time complexity of the algorithm is $O(nd)$ where, n = number of nodes in the topology d = average degree of all the nodes.

Now, for an undirected graph say e , = total number of edges. Then, $n \times d = n \times 2e / n = 2 \times e$

Hence, the algorithm is of complexity $O(e)$.

The average case time complexity may be much less since the algorithm does not traverse through any of the descendants of any *Down* node, the complexity will be much less than $O(e)$ where obviously there are some *Down* nodes from which alarms have come.

3.2.2 Part 2 of the algorithm

Now, for the second phase of the algorithm we have the following input specification. The physical topology and the status table are available as described in section 1.1, only difference being the introduction of another state in the status table as *Unknown*.

Lets say the node for which the root cause has to be determined as targetNode.

Data Structures

1. set S – it finally contains the collection of nodes which is/are the root cause of the node targetNode. The set has,

- i) S.create() which creates an empty set S.
- ii) S.insert(e) which inserts the object e into S.
- iii) S.remove() which randomly removes and returns one element from the set

2. list L of nodes – The same list defined under “Data Structures” in Section 3.2.1

3. boolean addedToList[n]. - This keeps track of what all elements has already been added in the list L(in declaration 2)
4. boolean addedToSet[n] - This keeps track of what all elements has already been added in the set S(in declaration 1)
5. A function output (string + object) which shows the string to the user, this may be an error message or some other message. For objects it prints the value of it.

Logic

```

Step1:
    for i = 1 to n
        stat[i] = statTable.getStatus(i);
Step2:
    for i = 1 to n
        {
            addedToList[i] = 0;
            addedToSet[i] = 0;
        }
    L.create();
    S.create();
Step3:
    if(stat[targetNode] == up)
        output("This node" + targetNode + "is
            up, it cannot have a root cause.");
    return from this function;
Step4:
    if(stat[targetNode] == down)
        output("This node" + targetNode + "
            itself is the root cause for it.");
    return from this function;
Step5:
    L.insert(targetNode);
    addedToList[targetNode] = 1;
Step6:
    while(L.isEmpty == false)
    {
        tempNode = L.delete();
        for all i in the linked list
            adjList(tempNode)
        {
            if((stat[i] == down)
                and (addedToSet[i] == 0))
            {
                S.insert(i);
                addedToSet[i] = 1;
            }
        }
        else if(addedToList[i] == 0)
        {
            L.insert(i);
            addedToList[i] = 1;
        }
    } // end of for
} //end of while

```

Step7:

```

Output ("The nodes " + S + "are/is the root cause
for the node " + targetNode );

```

Output Specification

The element(s) of the set S are the root cause of the status of the targetNode.

Time Complexity

The worst case time complexity is $O(e)$, i.e. in the order of the number of edges.

After getting a mapping for an *Unknown* element to a list of *Down* element(s) the reverse mapping can also be formed where the other type of lookup is also possible, i.e. given a *Down* element, we can specify a list of *Unknown* elements for which it is the root cause. Hence, the operator will know that if he services that *Down* node all its dependent nodes are also expected to be serviced if none of them have any other root cause.

4. Important Issues

Our algorithm is a generalized approach and is applicable for any system because it depends on the topology of the network and the list of events which all NMSes can maintain. The algorithm can be applied to very large networks also because of its $O(e)$ time complexity. But as the network grows the time taken to perform RCA increases. Hence, it cannot be applied after each event received. The scheme for applying the RCA algorithm involves deciding about when the algorithm has to be triggered to correlate the accumulated events. Its a jumping window scheme where a configurable time window of w minutes is started when the first event comes to the RCA module. The events collected for the time period (the duration of w) are correlated by calling the RCA module. The time window is reset then and it is restarted again when the first event comes after it. By waiting for the time w we make sure that the events for all the related network elements (related in dependency graph by parent child relationship) also appear within the time period w . This w is necessarily a function of the average polling interval of the elements.

The scheme also should take care of the severity (priority) of the events that occur within the time window. The severity of the events occurred are also cumulatively calculated and if it exceeds the threshold value for tolerance, the RCA algorithm is triggered independent of the time period w .

The RCA algorithm is called when the value of tolerance becomes greater than or equal to the threshold value configured for it or at the end of the time window w , whichever is earlier. After the RCA is called, the procedure starts again when the first event comes after that. Hence, value of threshold and w are reset and restarted at that time.

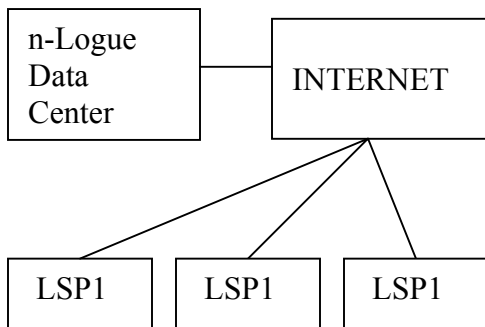
The type of the events and the sample values are:

Critical	8
Major	4
Minor	2
Warning	1
Threshold	8

Hence, two critical events or four major events or eight minor events or 16 warnings (or combinations such that the total becomes 16 or more) will trigger the RCA independent of the w configured. If the tolerance value does not touch the threshold within the time w , the RCA is triggered by default.

5. Case Study

n-Logue[9] is an emerging ISP aiming to provide low-cost service in far-flung rural areas. The topology of the n-Logue network is shown in Figure 1. n-Logue has a number of LSPs (Local Service Partners) distributed all over India. Each LSP provides voice plus Internet service in all the villages of a district or taluk.



The number of LSPs is currently around 40 and this number is soon expected to grow to hundreds. Each LSP in n-Logue has the following elements to provide voice and Internet services; corDECT WiLL [10] system Minnow servers [11], router and a leased line.

This network has resource and bandwidth constraints. Hence it has to use minimum bandwidth for management traffic. To manage this network, the NMS server is located in the national Data Center. Now if a link to one LSP is *Down* then without root cause analysis, all the machines under that LSP will show alarms. By applying root cause analysis in the NMS server only one alarm for the gateway of the LSP will be shown. It associates that alarm with all other alarms in the LSP.

6. Conclusion

In this paper, we have described a root cause analysis technique that exploits a dependency relationship that commonly exists between network elements in a typical network. Our technique uses management data which is already present in the NMS and does not impose any extra data collection overhead. The algorithm we use is efficient requiring $O(e)$ time.

The algorithm is a generalized one because even if the physical topology of the network cannot be discovered, it works correctly upon any user specified network graph. All the nodes of a disconnected component in the graph show unknown if the NMS interface is not connected to that component.

6.1 Future Work

The root cause analysis algorithm presented here is regarding the status of a network element. Root cause can also be found for another frequent problem in the network viz. congestion.

References

- [1] W.Stallings, "SNMP, SNMPV2, SNMPV3 and RMON 1 and 2, *Practical Network Management*", 3rd ed., Addison Wesley, 99.
- [2] Mani Subramanian, "Network Management : Principles and Practice", 2000.
- [3] "DownStream Suppression is Not Root Cause Analysis" http://www.smarts.com/resources/DS_WhilePaper_0802/pdf
- [4] http://www.alice-soft.com/html/prod_recall.htm
- [5] <http://www.managementsoftware.hp.com/>
- [6] <http://www-306.ibm.com/software/tivoli/>
- [7] <http://www.cabletron.com/>
- [8] Tremblay and Sorenson, "An introduction to data structures with applications"
- [9] n-Logue Communications (P) Ltd., <http://www.n-logue.com>
- [10] Midas Communication Research Ltd. <http://www.midascomm.com>
- [11] A.G.K. Vanchynathan, Ashwin Mansinhka, Kalyan, Devendra Jalihal, Timothy A. Gonsalves, "High Availability for ISP" ", *Proc. NCC 2001*, IIT Kanpur, India