January 09, 2019

REAL-TIME NETWORK EVENT CORRELATION BASED ON DEPENDENCY GRAPH FOR NETWORK ISSUE ROOT CAUSING

Alok Kumar Sinha

Zach Cherian

Kaushik Dam

REAL-TIME NETWORK EVENT CORRELATION BASED ON DEPENDENCY GRAPH FOR NETWORK ISSUE ROOT CAUSING

AUTHORS:
Alok Kumar Sinha
Zach Cherian
Kaushik Dam

ABSTRACT

Techniques are described for a graph based approach to co-relate events in a network. The root cause of any network failure may be determined in real time.

DETAILED DESCRIPTION

Typically any network issue or fault is reported with some minimum data. It is a painful job for network administrators to drill down the logs and work with the devices to find the underlying root cause of the network issue. Since merely addressing the symptoms of the issue does not help, administrators look to find the root cause of the fault. This helps in mitigating the issue quicker and also understand the network better.

Determining root causes is a painful process today. It requires administrators to perform debugging sessions with controllers and devices, and to collect logs and data. The techniques presented herein aim to make this root cause process proactive and almost real time. Administrators no longer spend time determining the root cause. Rather, they are shown the root cause of an issue within a few seconds/minutes of an issue occurring.

A simple approach to the problem would be to process each event based on where and when it is happening, and look for events related to this across the network. A rule engine may be built for such correlation. However, it will be impractical to run due to high Central Processing Unit (CPU) and memory requirements. Handling thousands of events would be impractical in such a way.

Alternatively, the network configuration and topology may be pre-processed and cached, and events may be processed based on this pre-processed cache.

Techniques described herein provide one such approach.

Events in a network happen at various network entities. Network entities may be physical such as interfaces or devices (e.g., switch, Access Point (AP), interface, client, servers, links, etc.), or logical (e.g., Quality of Service (QoS), Open Shortest Path First

(OSPF), Access Control List (ACL), Intermediate System to Intermediate System (ISIS), Locator/ID Separation Protocol (LISP), Authentication, Authorization, and Accounting (AAA), 802.1x, etc.). The event sources of various network entities may include syslogs, traps, Simple Network Management Protocol (SNMP) Management Information Base (MIB) polling based event, Command Line Interface (CLI) based events, edge analytics (e.g., states/counters crossing threshold on the box), etc.

In a given network, network entities appear at various parts of a topology. Network events may be a result of user initiated configuration changes, network events like link failures, protocol events, etc. A given event may in turn lead to more events. In other words, events happening at these network entities may be seen as a chain of events. The primary task of determining a root cause of any network issue boils down to finding the beginning of this chain of events.

One key aspect of this chain of events is that it is location aware and the events often occur close to each other chronologically within known bounds. Consider the following chain of events: PortDown → APDown → ClientDown. A given PortDown may lead to APDown for the AP connected to that port, or all APs connected down the line. But it does not affect the other APs. Network topology plays a crucial role in determining chaining of these events.

A dependency graph is provided with graph nodes (vertices) and graph links. The graph nodes may be the network entity instances as described above. A given network entity class (e.g., interface) may have multiple instances. Graph links may be nodes in the graph with a set of edges coming out, each one corresponding to possible events that can happen at these entities. These links may terminate at another network entity node if they lead to one or more event at that other entity. For example, a link corresponding to LINK-3-UPDOWN at an interface of a network entity node may terminate at the AP network entity node for a syslog event of an AP going down.

This dependency link among network entity classes may be hand coded (e.g., specified as a JavaScript Object Notation (JSON) file that may be read during graph building. These are the links that are based on network service operations on network switches/routers. The dependency links may also be learnt based on network operation.

Instances of entity classes may spread across network topology. Instances of entity classes may be created as graph nodes. EntityClassEvents may define entity classes and their events.

The above dependency graph of network entities may be built as follows: read network topology; read all device configuration; split each device configuration into various network entity classes, creating nodes for each instance on this device; read dependency JSON describing event dependency between classes of network entities (e.g., static network knowledge encoded using syslogs, traps, MIB, etc. forming the dependency).

Network entity nodes may connect based on dependency between entity classes and also locations of entity nodes in the network topology. Graph nodes may be connected by event dependency, based on network topology and event dependencies of entity classes.

This gives a graph based on network topology, device configuration, and network services operations (networking knowledge). The resulting graph gives an event co-relation graph. The originating node events generate others (only outgoing). The terminating node events do not generate others (only incoming). The transient node has both incoming and outgoing. Paths in the graph are sequences of events that may occur in a given time window. Event co-relation find graph paths as events occur.

Device configuration may be split into entity classes, with one node for every class instance. Node properties may include an entity class and NodeID based on location. The entity class determines event links. The NodeID may include the switch Universally Unique Identifier (UUID) (for switch resident entities), switch UUI and interface name (for interfaces on the switch), AP UUID (for AP resident entities and wireless clients), and/or client UUID (for clients). A node may be identified using NodeID and entity class.

Figure 1 below illustrates node creation.

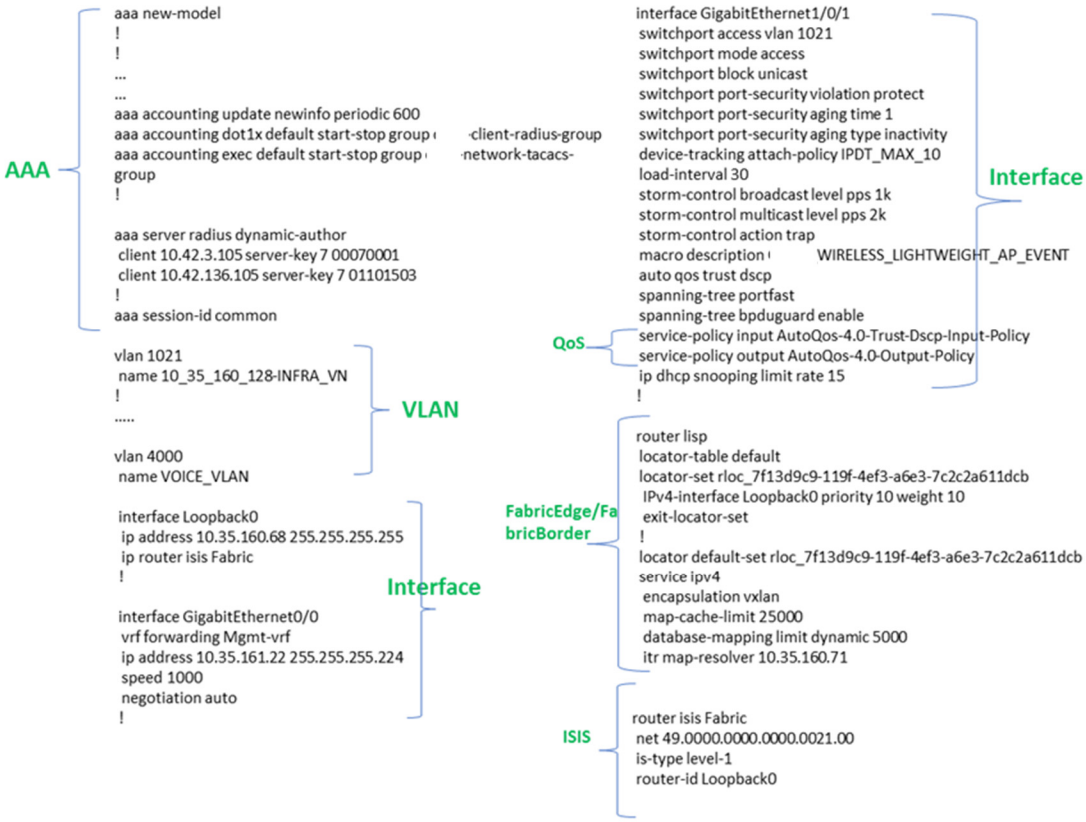

Figure 1

Nodes have events specified in EntityClassEvents. Events at the node produce links between nodes (specified in EntityClassEventDependencies). For an incoming link, some events from the other node may cause this event to happen at this node. For an outgoing link, the event at this node may cause other events on other nodes. Links may have properties such as weight and time lag. Weight may be how likely event1 causes event2 to happen. The time lag may be an expected time lag between events, and may be used for determining window timing during correlation.

The graph may be generated using network configuration, and may update when that changes. It may listen to network changes or just update periodically. Changes are patched based on locality of changes.

Figure 2 below illustrates an example high level view.

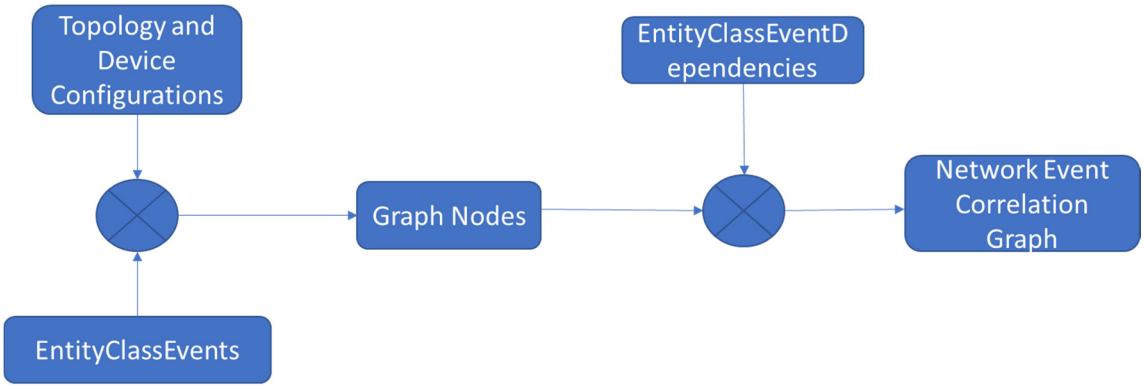

Figure 2

An example of entity classes and their events is provided as follows.

```
{
  "switch" : {
    "syslog" : [
              "SYS-5-RELOAD",
              "SYS-5-RESTART",
              "SYS-3-CPUHOG"
             ],
    "trap": [
      "reboot"
    ]
  },

  "ap" : {
    "syslog"  : [
               "LWAPP-3-IMAGE_DOWNLOAD_ERR2",
               "LWAPP-3-IMAGE_DOWNLOAD_ERR6",
               "LWAPP-3-IMAGE_DOWNLOAD_ERR7",
               "LWAPP-3-IMAGE_INVALID_RADIO",
               "LWAPP-3-IMAGE_RADIO_CRASH",
              ]
    "trap" : ["reboot"]
```

```
},

"interface" : {
 "syslog": [
  "LINEPROTO-5-UPDOWN",
  "LINK-3-UPDOWN",
  "LINK-5-CHANGED"
 ],
 "trap": [
  "linkDown",
  "linkUp",
  "cpeExtPolicingNotif"
 ],
 "mibs": [
  "QoSPolicyCounter",
  "ACLCounter"
 ]
},

"fabric-edge":{
  "syslog" : [
          "map_resolver_reachability_change",
          "MAP_SERVER_REACHABILITY_CHANGE",
          "LISP-4-CEF_DISABLED",
          "LISP-4-ASSERT",
          "PROXY_ETR_REACHABILITY_CHANGE",
          "LISP-5-LISP_TCP_SESSION_STATE_CHANGE"],
  "trap" : [
         "clispExtUseMapResolverStateChange",
         "clispExtUseMapServerStateChange",
         "clispExtReliableTransportStateChange",
```

```
            "clispExtUseProxyEtrStateChange"
          ],
  "mibs" : []
},
"fabric-border" : {
  "syslog" : [
            "LISP-4-CEF_DISABLED",
            "LISP-4-ASSERT",
            "LISP-5-LISP_TCP_SESSION_STATE_CHANGE"
          ],
  "trap" : [
            "clispExtUseMapResolverStateChange",
            "clispExtUseMapServerStateChange",
            "clispExtReliableTransportStateChange"
  ],
  "mibs" : []
},

"fabric-map-server" : {
  "syslog" : [
            "LISP-4-CEF_DISABLED",
            "MAP_CACHE_WARNING_THRESHOLD_REACHED",
            "MAP_SERVER_SITE_EID_PREFIX_LIMIT",
            "SITE_REGISTRATION_LIMIT_EXCEEDED",
            "LISP-4-ASSERT",
            "LISP-5-LISP_TCP_SESSION_STATE_CHANGE"
          ],
  "trap:" : [
            "clispExtUseMapResolverStateChange",
            "clispExtReliableTransportStateChange"
```

```
            ],
   "mibs" : []
},

"client" : {
  "syslog" : [],
  "trap" : [],
  "mibs" : []
},

"ospf" : {
    "syslog" : [
              "OSPF-3-INTERNALERR",
              "OSPF-3-UNKNOWNSTATE",
              "OSPF-6-PROC_REM_FROM_INT",
              "OSPF-5-ADJCHG"
             ],
    "trap" : [],
    "mibs": []
},

"isis":  {
     "syslog" : [
               "CLNS-5-ADJCLEAR",
               "CLNS-4-AUTH_FAIL",
               "CLNS-5-ADJCHANGE"
              ],
 "trap" : [
          "ciiAdjacencyChange",
           "ciiAreaMismatch",
           "ciiAuthenticationFailure"
```

```
        ],
 "mibs": []
},

"acl": {
  "mibs" : [
        "dropCountHigh"
       ]
},

"service-http" : {
   "syslog" : [
         "serviceUnreachable"
        ]
}
```

An example of entity class event dependencies is provided as follows.

```
"EntityClass-EntityClass-Dependency-Map": {
  "interface" : {
    "LINK-3-UPDOWN" : {
      "local" : {
         "ospf": {
           "syslog" : ["OSPF-5-ADJCHG"]
         },
         "isis": {
           "syslog" : ["CLNS-5-ADJCHANGE"]
         }
      },
      "remote" : {
         "ap": {
           "tbd" : ["some reboot indicator"]
```

```
        },
        "interface" : {
          "syslog": ["LINK-3-UPDOWN"]
        }
     }
  }
},

"isis" : {
  "CLNS-5-ADJCHANGE": {
    "local" : {
       "fabric-edge" : {
          "syslog" : [
                    "MAP_RESOLVER_REACHABILITY_CHANGE",
                    "MAP_SERVER_REACHABILITY_CHANGE"
                   ]
       },
       "fabric-border" : {
          "syslog" : [
                    "MAP_RESOLVER_REACHABILITY_CHANGE",
                    "MAP_SERVER_REACHABILITY_CHANGE"
                   ]
       }
    }
  }
  "remote" : {
      "isis" : {
          "syslog" : ["CLNS-5-ADJCHANGE"]
      }
  }
},
```

```
  "acl" : {
    "dropCountHigh" : {
        "local" : {
        },
        "remote" : {
          "service-http" : {
              "tbd": "serviceUnreachable"
          }
        }
    }
  }
}
```

Figure 3 below illustrates an example network topology.

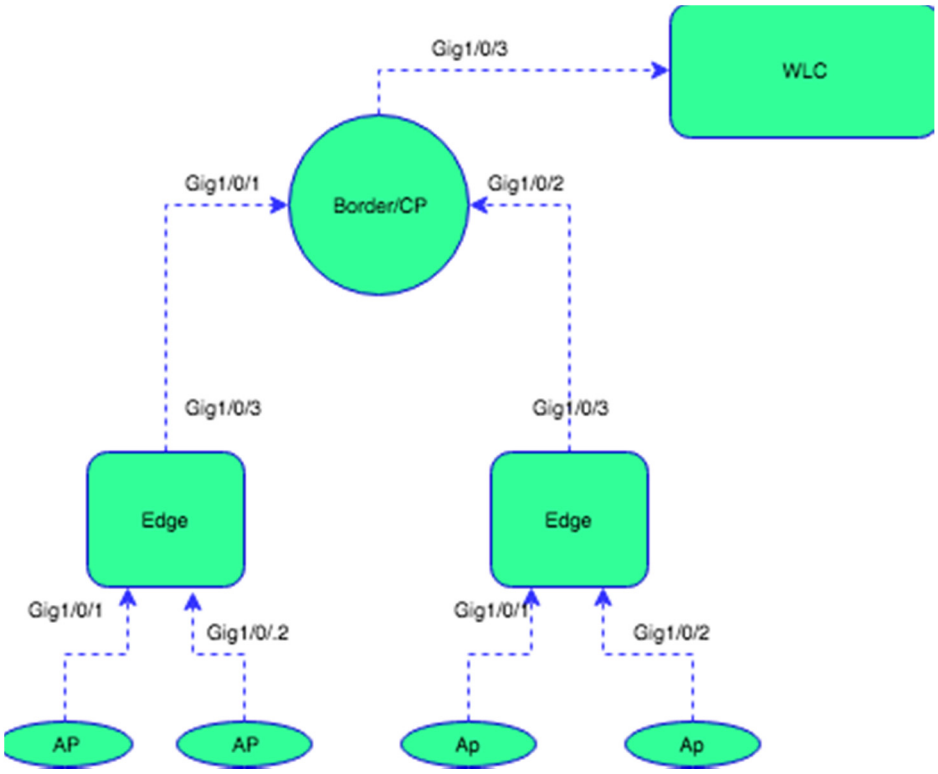

Figure 3

Figure 4 below illustrates the resulting event correlation graph.

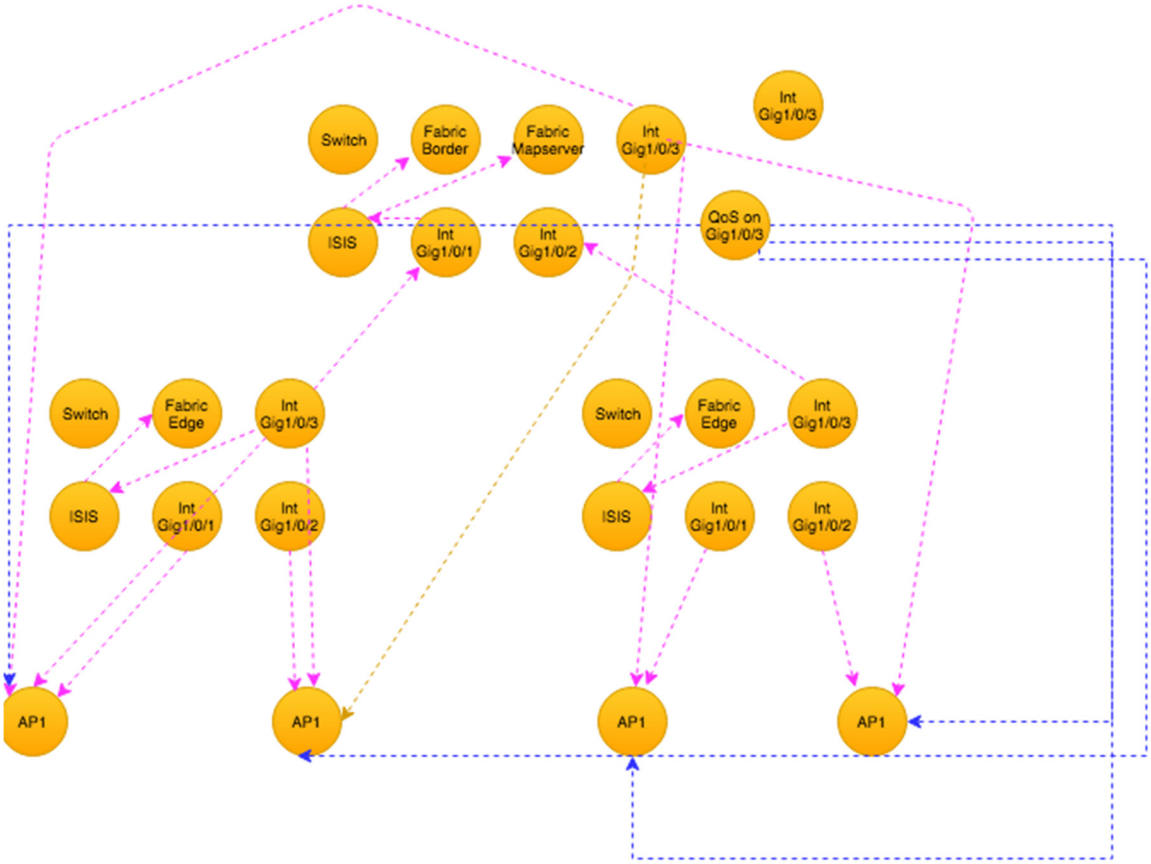

Figure 4

Figure 5 below illustrates an example zoomed in event correlation graph to illustrate event dependencies between nodes (entity class instances).

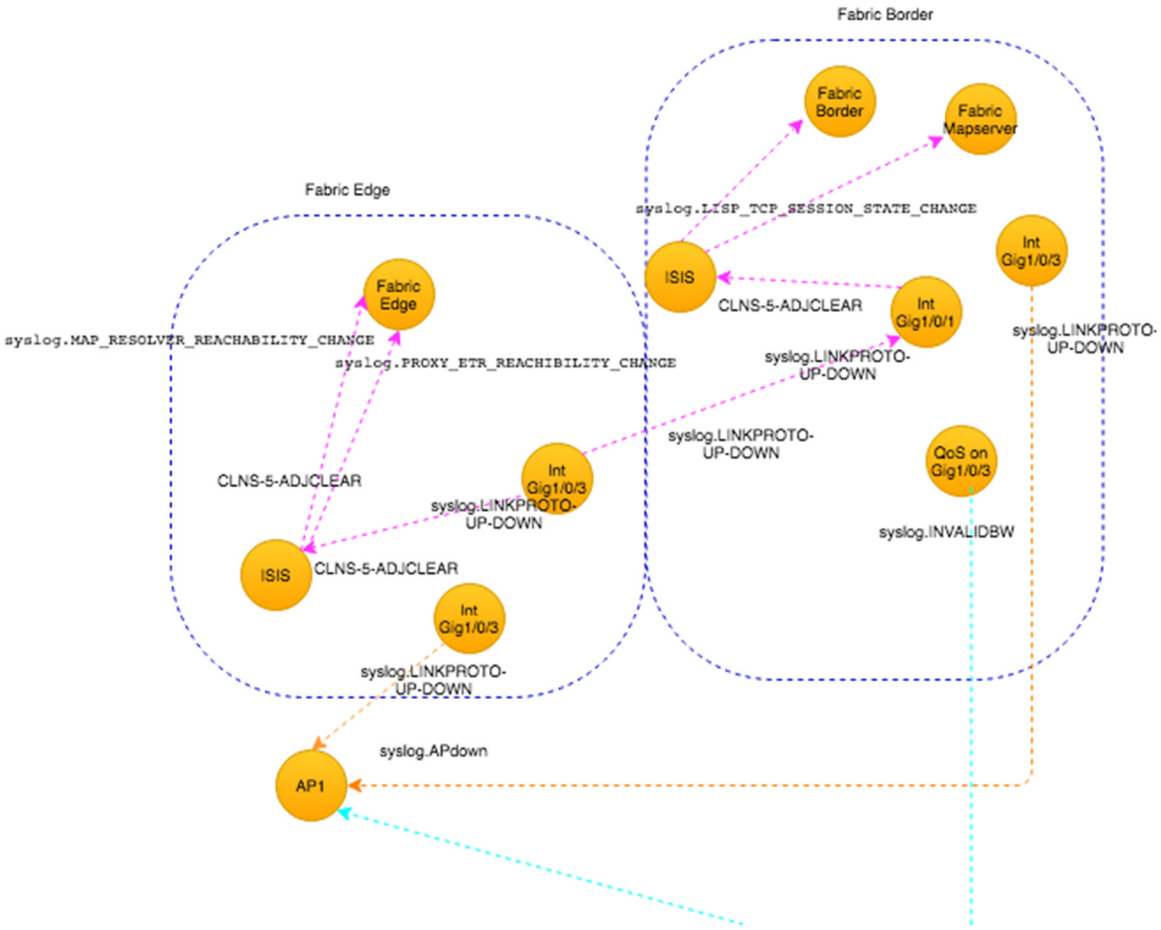

Figure 5

Such a graph may create thousands of nodes, and may be dense. There may be optimizations to reduce the memory footprint. For example, similar nodes may be collapsed into one (e.g., all interfaces with same configuration may be collapsed into one). An adjacency matrix versus linked list may evaluate the graph based on density. A reduced graph may store only hot spots of the graph. It may be adaptive by computing different parts of the graph based on event density with time.

It is impractical to have all network event knowledge hand coded into the above dependency graph. There may be over 18,000 syslogs and thousands of traps and MIBs. A more practical approach may be to learn the links between network artifact nodes based on a network event. As network events are processed, they are used to build more links

between these network entity nodes. A clustering based approach may be used to learn links.

Static links may be obtained from networking knowledge. However, there may be a need to learn thousands of links (not scalable manually), and some links may be network specific with a given traffic pattern and feature tuning. New links and update graph, time lag between nodes, and link weights may be learned.

A static graph may be generated as a correlation graph at a given network configuration. An active graph may be a subset of nodes whose events occurred in the time window. There may be multiple disparate active graphs at a time. Paths in the active graph are correlated events that recently happened. Paths in the active graph may be the complete root cause of related events that just occurred. An example path may be: Start node → … Transient nodes … → End node.

Paths in the active graph may also be partial with a missing start (e.g., a still unknown root cause). An example path may be: ? → … Transient nodes … → End node. There may also be one or more missing transient nodes (e.g., root caused, possibly with some probability). An example path may be: Start node → … ? … → End node. There may also be a missing end node (e.g., a new effect of the root cause). An example path may be: Start node → … Transient nodes … → ?. Whether partial or complete, the path helps in debugging issues faster.

For all events happening in a processing window of five minutes, the nodes corresponding to these events may be determined in the dependency graph. A traversal path graph algorithm may be run to find paths between nodes found in this window. A complete path gives a complete chain of events, with the source node being the root cause. In case of lack or loss of events, the path may be incomplete. However, missing nodes may be filled with one or more alternatives. This does not provide an exact root cause, but still narrows the solution space to a very few. This may rely on stream processing technologies for such event co-relation to occur in real time. The processing of events in a window occurs via a stream processing engine by feeding events to an engine processing graph.

Figure 6 below illustrates event processing threads.

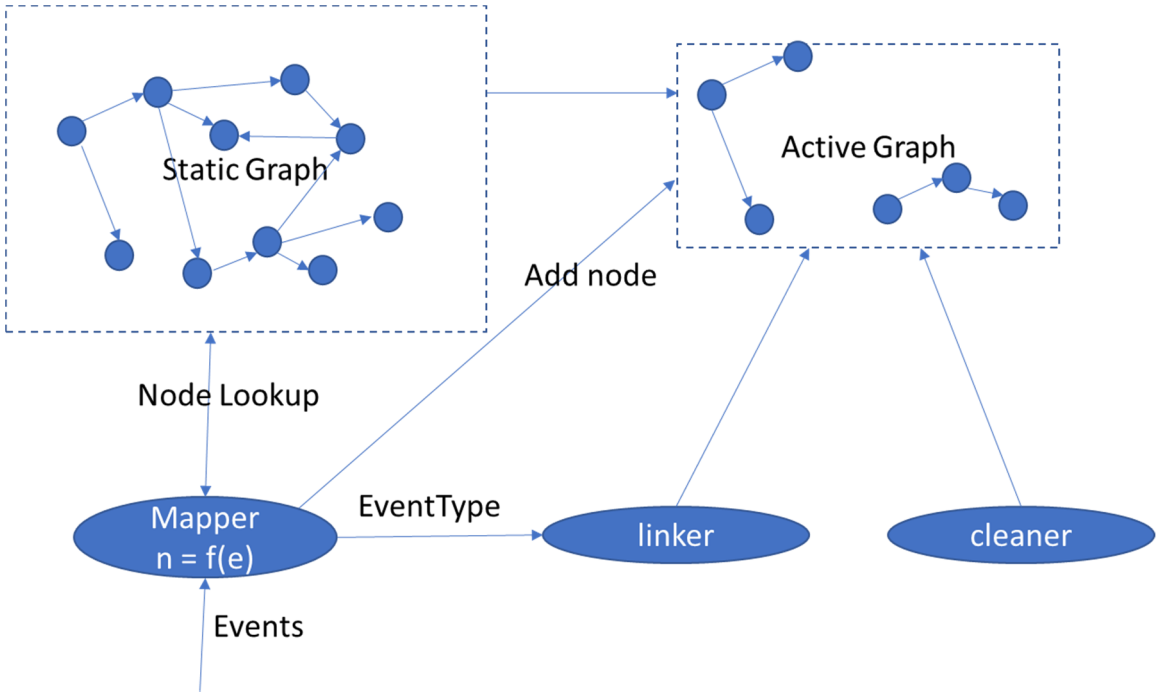

Figure 6

The mapper builds the NodeID from event info and maps it to a static graph node. The mapper also extracts event type(s) and maps it to event(s) in EntityClassEvents. The input may be a location (e.g., switch, AP, client, etc.), sub-location (e.g., interface), and/or event (syslog message type, trap ID, MIB ID, etc.). The output may be the NodeID and/or the EntityClass.

The linker connects new nodes found by the mapper to existing nodes of the active graph. The linker also finds incoming and outgoing links of the node/event pair. The outgoing link may connect to any "Waiting" peer node in the active graph. Otherwise, it may be marked as "Dangling." The incoming link may provide a check of whether the peer is a "Dangling" link in the existing active graph. Otherwise, it may be marked as "Waiting."

The cleaner identifies nodes falling out of the window, and if all nodes of a path fall out of the window, the path is removed. The path window may be a timestamp of any node in the path seen the first time and the Path.Timeout. The cleaner may check whether at least one event in the path is still in the window. If it is, it may wait, and if not, it may clean up nodes from the active graph.

The time window may be fixed at five minutes, and may be based on nodes/links. The time window may be specific to each graph path. The link time lag may be used for links. The time lag may be added up for possible paths between nodes.

Within a time window, events may be clustered based on event type, location in network, timestamp, and data path. Events within a cluster are likely to be related. Within a cluster, events may be arranged in ascending order of the timestamp. A link may be created between a lower timestamp event to a higher one.

There may be several operational requirements. Network configurations may be read for graph generation/updates. Real time event data may be used for learning links in the co-relation graph and processing events as co-relations in the graph. Storage may be used to store static co-relation graph in memory (e.g., on the order of MBs) and to store active co-relation graphs in memory (e.g., on the order of KBs). Compute resources may be used for graph node look and possible traversal, to update the static co-relation graph on a network configuration change, and to update the active graph.

Operational model options may include on-premise or cloud. For on-premise, graphs may be built in clusters from network configuration, a co-relation graph may be stored in memory in the cluster, and events may be processed in the memory in the cluster. For cloud, a graph in the cluster from network configuration may be built, a co-relation graph may be stored in the cloud, and events may be processed in memory in the cloud.

Event sources may be based on a pipeline or cloud-based analytics solutions. Pipeline-based sources may be on-premise only, and may scale issues for handling events for learning links and a processing graph. For on-premise implementations, the cloud-based analytics solution agent may be leveraged for event data. The event data may be processed locally on the cluster. For cloud implementations, the cloud-based analytics solution pushes events (anonymized) to the cloud, and may leverage this for a learning model and event processing.

In summary, techniques are described for a graph based approach to co-relate events in a network. The root cause of any network failure may be determined in real time.