Yemanja – A Layered Event Correlation Engine for Multi-domain Server Farms

K. Appleby, G. Goldszmidt
IBM T. J. Watson Research Center,
30 Saw Mill River Road
Hawthorne, NY 10532
applebyk@us.ibm.com, gsg@us.ibm.com

M. Steinder *
Computer and Information Sciences,
University of Delaware, Newark, DE 19716
steinder@cis.udel.edu

Abstract

Yemanja is a model-based event correlation engine for multi-layer fault diagnosis. It targets complex propagating fault scenarios, and can smoothly correlate low-level network events with high-level application performance alerts related to quality of service violations. Entity models that represent devices or abstract components encapsulate entity behavior. Distantly associated entities are not explicitly aware of each other, and communicate through event propagation chains. Yemanja's state-based engine supports generic scenario definitions, prioritization of alternate solutions, integrated problem-state and device testing, and simultaneous analysis overlapping problem analysis.

The system of correlation rules was developed based on device, layer, and dependency analysis, and reveals the layered structure of computer networks. The primary objectives of this research include the development of reusable, configuration independent, correlation scenarios; adaptability and the extensibility of the engine to match the constantly changing topology of a multi-domain server farm; and the development of a concise specification language that is relatively simple yet powerful.

Keywords
Fault and Performance Management, Management of Service Level Agreements

Research Paper

1 Introduction

Event correlation [13] is a commonly used technique for isolating the root cause of a problem from a stream of reported symptoms. Efficient and precise event correlation is essential for reducing network maintenance costs and improving the availability and performance of network services. Commercial correlation engines on the market today suffer from a number of drawbacks including:

*M. Steinder worked on the Océano team as a summer intern with IBM T. J. Watson Research Center, Hawthorne, NY, 2000

- Hard-coded network connectivity information. This causes an explosion in the number of defined scenarios to be maintained, additionally scenarios must be rewritten whenever the network changes.
- Explicit representation of network entity dependencies within the scenarios. This is expensive to maintain, and complicates each scenario. Adding a new model can require the rewriting of multiple pre-existing models.
- Initiating the network model traversal from all reported event sources. This causes unnecessary duplication of work.
- Inability to integrate actions that collect state and system and device attributes with the event correlation process. This makes it difficult to pinpoint the root cause of the problem.
- Fault diagnosis based on analyzing a sequence of events observed over a fixed time window. This provides only a subset of the functionality needed to perform problem identification, and does not allow for the different time frames of alternate problem solutions.

The system presented here (Yemanja) is model-based, and uses a backward chaining state engine. For each entity (a device or conceptual component) a problem behavior model is developed. Entity-models contain a set of problem scenarios, a set of input events that they consume, and a set of output events that they publish. Published events can be consumed by another higher-level entity. Each entity keeps tract of the entities it depends on. Since the publisher does not know who its consumers are, adding a new entity-model does not require the modification of any existing entity-models.

On every conceptual level scenarios correlate events that pertain to failure conditions on that level and generalize them into events with a higher degree of conceptuality using the concept of composite events [18, 19]. This layered correlation model is in compliance with recent work on modeling networks for supporting fault isolation and recovery [8, 23].

We believe that the layered approach to scenario development has many advantages over other approaches, these include:

- Support for the development of reusable correlation scenarios. In most cases system reconfiguration does not necessitate scenario modification.
- Rule development is simplified by limiting the number of events to be taken into account at any given time.
- The number of rules in the system is reduced.
- Efficiency is improved by limiting the number of duplicate condition computations.
- Unwanted rule interactions are kept under control.

The Yemanja correlation engine is being used to perform network problem determination, and Service Level Agreement (SLA) violation detection and enforcement [10] in a World Wide Web server farm environment. The farm is called Océano [2], and in addition to the normal farm management functions it supports dynamic resource reallocation based on SLA commitments.

The paper is organized as follows. In Section 2 we describe the run-time processing environment of our engine and the interactions between the entities that constitute the network model. Section 4 describes one of the correlation scenarios implemented for the Océano server farm. Section 3 describes the test bed environment. The current status of our system and future work are presented in Section 5. Section 6 contains the comparison of our system with other approaches to event correlation.

2 Correlation Engine

2.1 Entity Models

Yemanja works with a set of entity definitions that model each device and conceptual layer. Each entity has a set of scenarios that embody its problem behavior. Scenarios contain a set of rules that describe a specific type of behavior. Rules in the same scenario will consume some of the same events, and represent causal alternatives to the observed events.

Fault determination is performed independently for each entity and is based on incoming events and the current state of the entities it depends on. On each conceptual level scenarios correlate events that pertain to failure conditions on that level and generalize them into events with a higher degree of conceptuality. These generated events are published for consumption by higher-level entities (in the form of internal events). An entity that determines that its corresponding device or layer failed may initiate automated recovery, notify the system administrator, and/or open a problem record.

Communication systems are frequently modeled using a dependency graph whose directed edges indicate that the entity at the tail of an edge depends on the entity at the head of the edge [8, 15]. The network impact graph created by reversing the edges of the dependency graph shows which entities may fail because of the failure of another entity [8]. It may be observed that failure symptoms propagate along both dependency and impact graph edges. For example, layer-two's ability to form a spanning tree (Spanning Tree Protocol [22]) depends on the quality of the Ethernet links connecting the bridges. Failure of one of these links necessitates the rebuilding of the spanning tree, i.e., the failure at the *Ethernet Link* layer caused the failure at the *Spanning Tree* layer. On the other hand, a spanning tree loop can increase the number of packets dropped by Ethernet links. In this case the higher-level entity (*Spanning Tree*) failure causes the lower level entity (*Ethernet Link*) failure.

Following the above argument we classify events published by a Yemanja entity as either *cause* or *impact*, similar to the classification in [23]. In Yemanja an entity publishes a *cause* event to indicate that a failure has been detected, and that its dependant entities should assess how they are affected. If the *cause* event recipient determines that the service or function provided by the corresponding device or layer has been affected by the lower-layer failure, it publishes a new *cause* event. The new *cause* event contains information about the change in the operational state of the entity's corresponding device or layer incurred by the lower-layer failure. Bottom-up propagation of *cause* events and gradual translation of detailed failure information into more abstract terms allows us to correlate events from various layers without performing a time-consuming network dependency graph traversal. When an entity determines that the events it received does not indicate a failure in its corresponding device or layer, it publishes an *impact* event. Impact events are viewed as symptoms by their recipients.

In the process of event correlation Yemanja entities may initiate database queries (in SQL [5]) and/or issue SNMP [4] commands to obtain configuration information. If an entity fault is isolated recovery actions may be taken. Recovery actions can include system reconfiguration which is performed by a separate *Resource Manager* component, or directly by Yemanja through SNMP. In addition, after isolating the problem the entity stores information about the state of the corresponding device or layer in the open-problem database. The database provides information about the status of entities to the fault isolation process after the events used to isolate the problem are removed from the engine. The open-problem database may contain information about any problem, not only root failures. However, only root problems are reported to the system administrator. Figure 1 presents interactions among Yemanja entities and other Océano modules (Section 3).

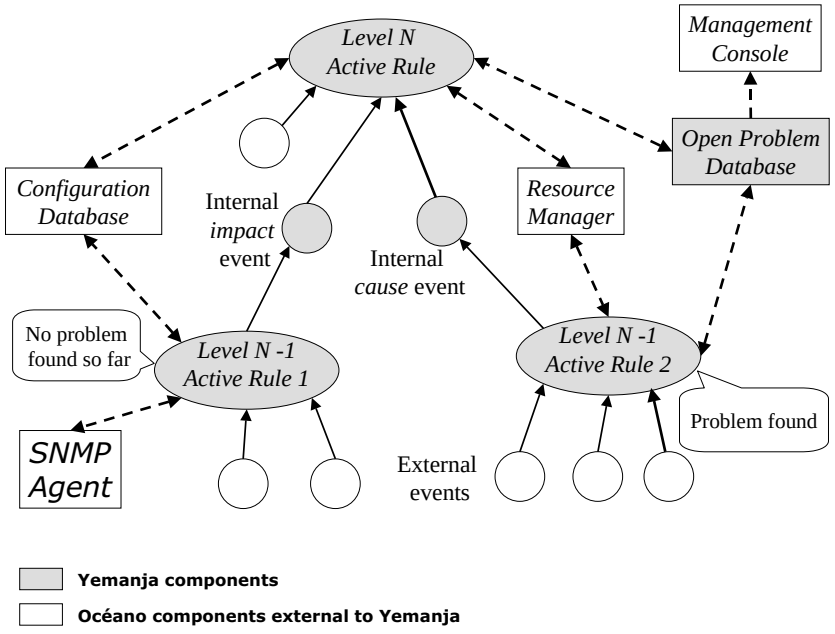

Figure 1: Interactions among Yemanja entities and other Océano modules.

2.2 Runtime Processing

When solving scenarios in an event-driven system there are advantages to using forward-chaining techniques over backward-chaining ones. The main reason for this is that it prevents repetitive rule firing caused by attempting to solve a rule multiple times for which all the required events have not yet arrived. For this reason Yemanja uses forward-chaining, this is done by maintaining the partially solved state of all active rules in the system. When a new event arrives, the states of the active rules with which the event correlates are updated. Using this technique we do not try to prove hypotheses for with the supporting events have not yet arrived. If the event does not correlate with an active instance of a rule within which the event type appears, a new instance of the rule is created.

When a rule is satisfied it is moved to the satisfied list, and may or may not be executed. Whether or not a satisfied rule is fired depends on the state of the other rules the rule events appear in. If there are additional active rules with the same scenario name the satisfied rule must wait for all higher priority rules in this scenario to time out or fail before it can be executed. If any rule event appears in more than one scenario, the higher priority scenario takes precedence.

After a scenario is executed the events correlated in it are canceled. This can be done implicitly or explicitly. If events are canceled implicitly they will be removed only from the scenario that was invoked. If done explicitly they are removed from all scenarios they are found in. Rules that have events removed must have their state rolled back, if they were satisfied and become unsatisfied they must be moved to the `active_rules` set and may cause lower priority rules to be reactivated. Depending on the desired behavior one or the other method will be selected. In the implicit case the event can be used to trigger the firing of an additional rule in a separate scenario. This can be useful when multiple independent recovery operations are required.

The basic correlation algorithm is depicted in Figure 2.

Let **R** be the set of rules that accept the event **E**
Let $\mathbf{R_A} \subseteq \mathbf{R}$ be the set of active rules that accept the event **E**
Let $\mathbf{R_{A\ not\ correlated}} \subseteq \mathbf{R_A}$ be the set of active rules that did not correlate with **E**
Let $\mathbf{R_T}$ be the set of inactive rule templates plus the rule templates of the active rules
that did not correlate with **E**, i.e., $\mathbf{R_T} = (\mathbf{R} - \mathbf{R_A}) \cup \mathbf{R_{A\ not\ correlated}}$
Let $\mathbf{R_S}$ be the set of satisfied rules that **E** was correlated with

Receive event **E**,
For each rule instance $\mathbf{r_i} \in \mathbf{R_A}$ {
Try to correlate **E** with $\mathbf{r_i}$
If correlation succeeds update the state of rule $\mathbf{r_i}$
If $\mathbf{r_i}$ is satisfied move $\mathbf{r_i}$ to $\mathbf{R_S}$, and remove lower priority rules from $\mathbf{R_A}$
}
For each rule template $\mathbf{r_t} \in \mathbf{R_T}$ {
Try to correlate **E** with $\mathbf{r_t}$
If correlation succeeds create a new active instance of $\mathbf{r_t}$, $\mathbf{r_i}$, and add it to $\mathbf{R_A}$
If $\mathbf{r_i}$ is satisfied move $\mathbf{r_i}$ to $\mathbf{R_S}$, and remove lower priority rules from $\mathbf{R_A}$
}
If $\mathbf{R_A}$ is empty (i.e., there are no higher-priority rules waiting to be solved) {
Select $\mathbf{r_i} \in \mathbf{R_S}$ with the highest priority in terms of rule and scenario priority
Execute actions found in $\mathbf{r_i}$'s action list
Delete event **E** and roll-back rules in $\mathbf{R_A} \cup \mathbf{R_S}$ as determined by event cancel mode
If $\mathbf{R_S}$ is not empty continue from the beginning
}

Figure 2: Correlation engine algorithm

2.3 Problem Scenario Definitions

Problem scenarios for a given entity contain any number of rules of the following format.

rule_spec(**Scenario-Name, Priority, Event-Predicate-List, Timing, Action-List**)

- **Scenario-Name** – Name of a scenario that the rule belongs to.
  There can be several rules in a single scenario. Each rule is considered as an alternative solution to the same basic problem.
- **Priority** – Priority of this rule within the scenario.
  The rule with the lowest index has the highest priority.
- **Event-Predicate-List** – Set of events and predicates that make up the conditions under which the rule is satisfied.
  Predicates are arbitrary Prolog predicates or Java methods. Events are of the form:
  `event(`**Event_ID, Event_Def, Resource, TimeStamp, Data, Count**`)`
  or
  `event(`**Event_ID, Event_Def, Resource, [Resource_Set_Generator(...), Set],**
  %Of_Set_Required, TimeStamp, Data, Count`)`
  Arguments of an `event` term are defined as follows.

Event_ID	Identifier of the event
Event_Def	Event type, e.g., `Interface.Link_Down`
Resource	Identifier of the resource the event refers to, which may be an integer, a string, or a list of the above depending on the type of the resource
TimeStamp	Time when the event was sent
Data	List of arbitrary event data, consisting of (name, value) pairs
Count	Number of times the event should be received before it is considered satisfied
Resource_Set_Generator(...)	Predicate that generates the set of resource identifiers from which we will wait for events of the type specified in **Event_Def**
Set	Set returned from **Resource_Set_Generator(...)**
%Of_Set_Required	Percent of elements in **Set** that the event must be received from for the event to be considered satisfied

- **Timing** – Term of the form `timing(Solve_time, Rearm_time)` where `Solve_time` is the number of seconds within which the rule must be solved for it to apply, and `Rearm_time` is the amount of time the scenario is inactivated before it is applicable again.
- **Action-List** – A list of predicate calls, many of which are built-in.

The following two rules belong to the scenario (`Customer.BRT.high`) that is activated when a `BRT.high` event is received. The `BRT.high` event indicates that web server, `WebServerId`, has noticed excessive response times for requests directed to the back-end server (this is usually a database). The first rule checks to see if there is an open problem involving the web server's connection to the back-end. If so, the event is discarded since the problem is already being processed, and back-end response time cannot be improved. The second rule uses the multi-event form of an event, and waits to see if the event is received from 70 percent of the web servers assigned to the customer who owns the server the initial event was sent from. If the event is satisfied we check to see if there is an open problem involving the back-end server's reachability, (i.e., it has a problem with its network connection to the front-end servers or is down). If so, we escalate the problem and take no further action as the problem is already open.

```
rule_spec('Customer.BRT.high', 1,
          [ event(E1, 'BRT.high', WebServerId, Time, Data, 1),
            find_problem(ProblemID_, 'WebServer.backend-connection-problem',
                         WebServerId, _, _, _, _, _, _)
          ],
          timing(1, 1),
          [ cancel_events([E1])]
).

rule_spec('Customer.BRT.high', 2,
          [ event(E1, 'BRT.high', WebServerId,
                  [ CustomerServers(WebServerId, AllWebServers), AllWebServers],
                  0.7, _, [['BE SERVER ID', BEServerId]], 1),
            find_problem(ProblemId, 'BEServer.frontend-problem',
                         BEServerId, _, _, Data, _, _, _)
          ],
          timing(100, 600),
          [ escalate_problem(ProblemId, 'critical'),
            cancel_events([E1])
          ]).
```

2.4 Built-in Predicates

Yemanja provides a number of built-in predicates that facilitate the process of rule development. While the predicate library is still being extended, the functionality currently available in Yemanja may be classified according to the following list:

1. Messaging
   - Sending messages to the management console.
   - Initiating auto-recovery using an external module.
   - Publishing events for consumption by higher level entities. Forwarded events are inserted at the beginning of the input event stream. These events contain a reference to the associated open problem (if one exists), which gives event recipients access to the root cause of the forwarded event.
2. Problem record management
   - Opening new problems. For every problem we specify at least its type (*Bad Cable*), identifier of the failed resource, and severity. If the problem is not a root cause and it is opened as a result of a correlation involving an internal *cause* event, the problem record will also contain the identifier of the parent problem. Optionally, a template list of events that will be used to cancel the problem can be specified.
   - Increasing the severity level of a problem.
   - Searching for problems matching a specified description.
   - Clearing problems. Cleared problems are removed from the database and cancellation of resultant problems is initiated as described above.

   Note: Problem management will be described in more detail in Section 2.5.
3. Retrieving MIB object values from SNMP agents [4].
4. Retrieving Configuration data from an SQL database.
5. Canceling events from scenarios. Events can be removed from the current scenario only or from all defined scenarios.
6. Scheduling a predicate for execution at a later time.

   Predicate `delayed_method(Delay_Time, Method)` delays the invocation of predicate `Method` for `Delay_Time`. In the following example, the `process_sustained_event` predicate will be delayed for five minutes. The `process_sustained_event` predicate will escalate the CISCO switch event `Chassis_Major_Alarm` to a `Switch_Down`, unless a `Chassis_Major_Alarm` cancellation event was received during the delay period.

```
delayed_method(5min, process_sustained_event(Switch_Id,
                                             ['Chassis_Major_Alarm', Result] )).
```

2.5 Managing Problem Records

As mentioned in Section 2.1, every correlation entity can open problem records. A problem describes the current operational state of the device or layer corresponding to the entity. A problem open in a higher layer *A* resulting from a problem open in a lower layer *B* hides the problem detail used by layer *B* from layer *A*. High-level events are correlated with high-level open problems only. Thus, we do not need to traverse the network model looking for low-level root causes that might correlate with a newly received high-level event every time a high-level event is received.

Within the open-problem database cause-effect relationships between problems are represented using a doubly-linked list. Thus, given a particular problem, the correlation engine always has access to its root cause and its resultant problems. When a lower-level problem is cleared, either manually by the operator or automatically by Yemanja, the resultant problems can also be cleared. However, it is not necessarily the case that a higher-level problem *A* recorded as an effect of a lower-level problem *B* ceases to exist when *B* is canceled. For example, a new lower-level problem *C* may occur while *B* is open, which even after *B* is resolved inhibits automatic correction of problem *A*. If this happens, when *B* is resolved, *A* should be bound to *C* rather than canceled. Recognizing that *C* prevents *A* from being cleared is difficult. There are three cases of this situation to consider.

In the first case events received by the *entity* that opened problem *A*, are received multiple times. This occurs when the source sends the event at regular intervals as long as the condition persists. In this case, event repetitions may be used to re-open problem *A* after it is canceled along with problem *B*. When the problem is reopened it will be bound to the currently existing problem *C*. However, many network monitoring tools implement a hysteresis mechanism that inhibits repetitive notifications of the same condition. Typically, one notification is sent to indicate occurrence of the abnormal condition (*bad event*) and another one to indicate that the condition no longer exists (*good event*). In this second case when the bad events related to problem *A* are not repetitive, but a cancel problem event for *A* has been sent, we need to make sure that the corresponding good events are received before we cancel problem *A*.

Yemanja implements a generic event cancellation rule that handles both of the above cases. When a problem is open the rule builder has an opportunity to specify templates of events that have to be received before the problem may be cleared. When the root problem is closed using `clear_problem` predicate, internal event `Oceano.Yemanja.problem.cleared` is automatically sent for every problem directly resulting from the root problem, with `ProblemId` identifying the resultant problem. The event is consumed by the rule presented below, which is assigned the highest possible priority to guarantee its execution before any user-defined rules consuming the `Oceano.Yemanja.problem.cleared` event.

```
rule_spec ('clear_problem', 1,
           [ event (E, 'Oceano.Yemanja.problem.cleared', ProblemId, _, [], 1),
             open_problem:get_clear_events(ProblemId, EventList, EventIds, TimingPred)
             | EventList ],
           TimingPred,
           [],
           [ clear_problem(ProblemId),
             cancel_events([E | EventIds])
           ]).
```

The predicate `open_problem:get_clear_events` in the rule above reads the event templates that were specified when the problem `ProblemId` was opened. This list, `EventList`, is then unified with the tail of the rule's `Event-Predicate-List` described in Section 2.3. Adding these events to the tail of the `Event-Predicate-List` forces the Yemanja engine to block this rule until the events in `EventList` arrive. Similarly, the timing can be defined when the problem is opened and instantiated when problem cancellation in initiated. Using this method every non-root problem automatically builds its cancellation rule, which will be satisfied only if the events specified

when the problem was open are received in the given time window. After all events have been satisfied, the rule uses the `clear_problem` predicate to delete the problem `ProblemId` and recursively clear resultant problems.

If the above rule times out or fails, we look to see if there are any custom `rule_spec`s defined for the `ProblemId`. If there are none the default action, which cancels the event `Oceano.Yemanja.problem.cleared` and leaves the problem open, is taken. If problem cancellation is to be handled explicitly the template event list passed to `open_problem` is set to `[fail]`.

3 Test bed environment: The Océano Server Farm

The Océano project [3] is developing a scalable infrastructure that enables multi-enterprise hosting on a virtualized collection of hardware resources. Hosted customers increasingly require support for peak loads that are orders of magnitude larger than what they experience in their normal steady state, particularly for commercial Web workloads. Océano provides a hosting environment that can rapidly adjust the resources (bandwidth, servers, and storage), assigned to each hosted customer, based on the dynamically fluctuating workload. It constructs dynamically created domain clusters for each hosted enterprise, and introduces high levels of automation to resource management. The main objective is to investigate hosting architectures that enable high availability and dynamic scalability, for delivering networked applications and services that are capable of handling large surges in traffic. We have implemented a pilot over a set of 50 Linux and 6 AIX servers. The hosting infrastructure expands the service levels provided to customers, without compromising their security requirements. Hosted customers increasingly require support for peak loads that are orders of magnitude larger than what they experience in their normal steady state. The ”collocation” hosting model, however, uses dedicated servers for each customer. In this model, enabling peak-load scale on demand would require large investments in standby, non-shared resources, which would be mostly under utilized and would occupy large amounts of physical space. Such a model does not efficiently mitigate the differences between average and peak load.

Océano provides automation tools to reduce the costs of setting up and operating the hosting farm, and to offer a variety of SLAs to the hosted customers. While the hosting infrastructure uses shared servers and storage, at any point in time, each resource is dedicated to a single customer. Océano will quickly (minutes) move customers to (larger/smaller) servers in response to changing workloads or failures. These changes require networking reconfigurations, which are accomplished without service interruption. Customers will be offered a tiered service bounded by discrete bars, where the lowest bar represents the minimum guarantee that must be maintained, e.g., at least S servers and B external bandwidth. Application models determine the logical configuration (single-tier or multi-tier) of the server nodes, and range from simple Web serving to complex multi-tier e-commerce applications. An Océano farm consists of 3 tiers: front-end IP sprayers for load balancers (e.g., Network Dispatchers [7]), large pools of front-end servers, and static back-end servers, all interconnected by switches. A ”freepool” of servers is available to be ”leased” to customers. When the load on a customer drops below some level, one of its servers is quiesced, ”scrubbed” of any residual customer data, and assigned to the freepool. Later, when the load on another customer exceeds some trigger level, a freepool server is primed with the necessary OS, applications, and data to acquire the personality of that customer. Monitoring and dynamic resource allocations are used to shift resources between customers based on Service Level Agreement [10], (SLA) commitments, and penalties.

A small set of dedicated nodes implement the management functions. For the purposes of this paper we will combine a number of relevant subcomponents under the common name the Océano `Resource Manager`. The *Resource Manager* determines how to reallocate resources when a service level threshold is violated; automates network reconfiguration and node priming (loading customer data); and ensures that discovered physical and logical configuration are consistent with the configuration database.

4 Example Correlation Scenarios

In this section we present one of the fault scenarios developed for the Océano server farm. Consider the configuration of network devices shown in the small customer-segment depicted in Figure 3. This segment contains one back-end database server (number 47), and three front-end web servers (numbers 11, 12, and 13). The front-end servers and back-end server are connected through VLAN [22] 462 using network adapters 36, 39, 42, and 144 respectively. Front-end servers receive requests from a load balancer (Network Dispatcher [7]) via VLAN 461 using network adapters 35, 38, and 41.

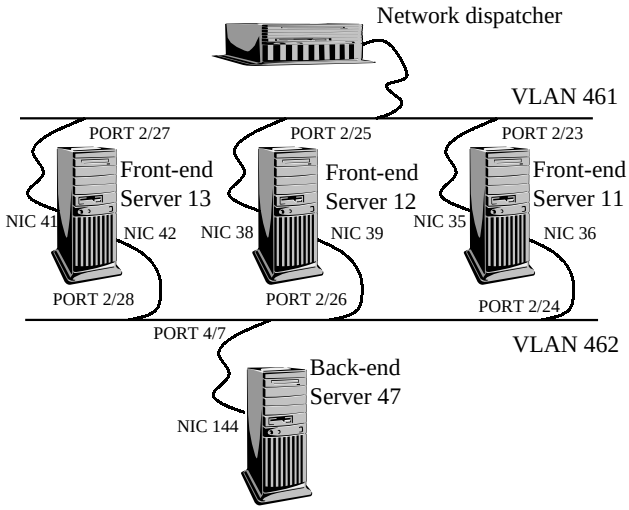

Figure 3: Sample network configuration

Consider the following fault scenario that may occur in this network. Suppose that a broken cable that is causing random bit errors to be introduced into the Ethernet frames transmitted between network adapter 144 and port 7 on switch blade 4. As a result, the SNMP agent on both back-end server 47 and the switch detect an increased number of bit errors in their input packets [20, 25]. When the *rising thresholds* [25] are exceeded, both agents send rising threshold alarms that are normalized into `ethernet_interface.input_errors-High` events with resource identifiers set to 144 and ['1.2.3.4', 4, 7] respectively.

We have defined an *Ethernet Link* entity that models the Ethernet link which is composed of a network adapter, switch port, and 10BaseTX cable connector (Figure 4). The scenarios in this model correlate all events related to the operation of the corresponding network adapter, switch port, and cable connector. One of the *Ethernet Link* scenarios models the behavior that caused the `ethernet_interface.input_errors-High` events. This scenario will check the configuration database to see if network adapter 144 and port 4/7 on the switch with administrative IP address '1.2.3.4' are connected. Since they are, the scenario correlates the two received events.

When bit errors are detected in an increased number of Ethernet frames, and this occurs on both ends of an Ethernet link the most likely cause is a bad cable. Therefore, the satisfied rule opens a problem indicating that the cable has to be replaced (`ethernet_link.bad_cable`). It also publishes an internal Yemanja event (Figure 4) indicating that Ethernet frames are lost on the Ethernet link connected to network adapter 144. When performing correlation at higher-levels we do not need to know the exact events or problems that were reported at the *Ethernet Link* level. Thus bit errors, lack of buffers, or collisions could have all been generalized into the `loss_and_delay` event that is published at this level.

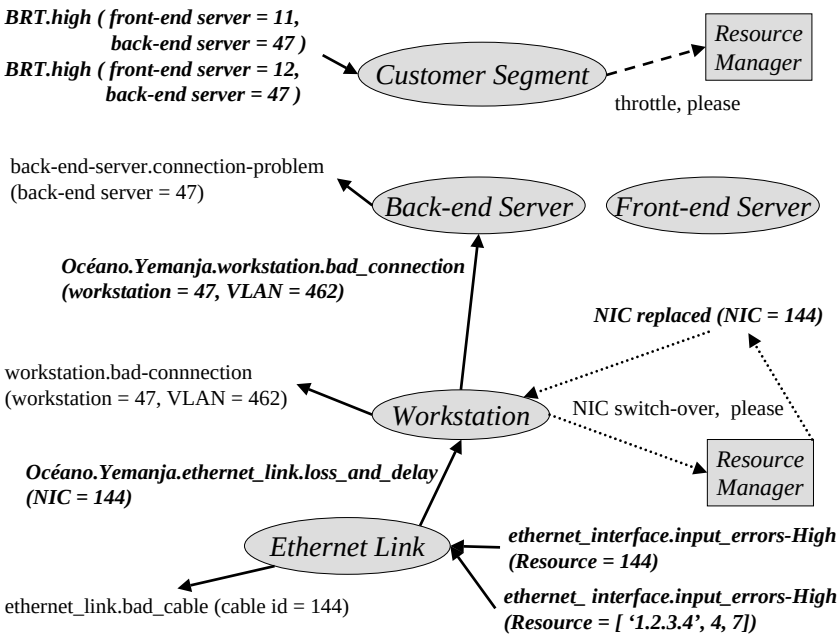

Figure 4: A correlation scenario for `bad_cable` condition

The event published by the *Ethernet Link* entity activates scenarios belonging to the *Workstation* entity-model. This scenario translates the received event into higher-level semantics again and generates a (`bad_connection` event and problem, see Figure 4). This operation eliminates duplicate computations of predicates involved in the translation process from scenarios representing workstations of particular type, such as *Front-end Server* and *Back-end Server*. Since workstation 47 is a back-end server, the translated event activates scenarios belonging to the *Back-end Server* entity, whose actions depend on the VLAN whose identifier was specified as the event attribute. VLAN 462 is used for communication between front-end servers and back-end servers in this customer-segment. Therefore, *Back-end Server* opens a new problem `back-end-server.connection-problem` for back-end server 47. The `bad_connection` condition is ignored when the VLAN for which it is reported is used only for administrative purposes (VLAN not shown in Figure 3).

After the back-end server problem has been opened suppose an application-level monitor on front-end server 11 detects that the request latency between itself and back-end server 47 is excessive. In this case it generates a `BRT.high` alarm. For a subset of the back-end processes supported by the Océano server farm this condition indicates that the back-end server is overloaded. For this subset of customer-segments we look to see if a significant number of its servers report the alarm, if they do Océano alleviates the problem by decreasing the admission rate to the customer-segment. However, in the presence of network faults, decreasing the admission rate may be the wrong solution. In the presented example increasing the throttling rate would not significantly improve the performance of requests involving access to back-end server 47, and would negatively affect other requests in this customer-segment.

Yemanja's *Customer Segment* entity handles this situation by correlating the alarms related to quality of service with information corresponding to the operational state of the network infrastructure. When the *Customer Segment* entity is activated by the `BRT.high` alarm it checks if there are any open problems associated with the front-end server's back-end connection, back-end server, or the entire sub-network connecting the front-end servers to the back-end servers (e.g., resulting from a switch failure). Since a problem associated with the back-end server is found, the *Customer Segment* entity ignores the `BRT.high` event.

Later when the faulty cable is replaced, the problem `ethernet_link.bad_cable` is closed, and all resultant higher-level problems are recursively closed as well. After the problem `back-end-server.connection-problem` is closed, arriving `BRT.high` alarms are processed by a different rule within the *Customer Segment* entity, this rule will submit the request throttling request to the *Resource Manager* (see dashed arrow in Figure 4).

4.1 High-availability

Let us now imagine that back-end server 47 has two network adapters in VLAN 462. Upon reception of the `Oceano.Yemanja.ethernet_link.loss_and_delay` event the *Workstation* entity checks if the backup network adapter is fully operational. If so the *Workstation* entity sends a message to the *Resource Manager* requesting that the network adapters be exchanged (see three dotted arrows in Figure 4).

Regardless of whether the Océano *Resource Manager* has been notified, the *Workstation* entity opens the `workstation.bad_connection` problem and publishes the event `Oceano.Yemanja.workstation.bad_connection`. This insures that higher level alarms continue to be explained by this condition until the switch-over actually occurs. After notification is received that the switch-over has been completed the `workstation.bad_connection` problem and all resultant problems are closed. Notice that the `bad_cable` problem remains open until the cable problem is resolved.

4.2 Problem escalation and management console reporting

In Océano as with any management system we are concerned with limiting the number of notifications sent to the management console. However, network administrators have different preferences regarding what type and severity of problems they want to be informed about. Some network management products may be customized with respect to when and how problem notifications are sent to fit an individual administrator's needs. For this reason problem entity, resource type, and severity are sent along with each administrator message. The management console software is then responsible for determining which messages each administrator will see, and how they should be presented.

Some problems are assigned a low severity when they are first opened, e.g., the bad cable problem described above. If in a higher layer we determine that a low severity problem has an impact on the deterioration of service offered by the higher layer, e.g., `BRT.high` events have been received from a number of front-end servers, the severity of a low level problem can be escalated. Problem escalation will in many cases trigger new administrator notifications. To support problem escalation the unique identifier of root cause problems are forwarded along with published events.

4.3 Configuration problems

One of the most common causes of network faults is misconfiguration. In Océano this problem is even more important as configuration changes are frequent. Moving a network adapter from one VLAN to another, adding a new node, or adding a new network adapter are very common. Thus, automating the detection of configuration inconsistencies between the database and the current network topology is very important in the Océano environment.

The management agents on network devices provide some indications of configuration changes, e.g., *linkUp/Down* [4] and *topologyChange* [6] traps. In Yemanja these Yemanja was able to use traps from the management agents on network devices to provide some indications of configuration changes. In Yemanja these traps are used to isolate faults, and trigger database consistency checks. When the *linkDown* trap is received we check the administrative status of the interface for which the down status was reported and if the administrative status is also

down, we verify if the database contains accurate information about the state of the network adapter connected to this interface. Any inconsistency detected may be either reported to the manager or automatically corrected.

5 Current Status and Future Work

Our work to date has concentrated on the implementation and testing of the correlation engine and development of an initial set of entity-models and associated correlation scenarios. The engine was written in an efficient version of Prolog, SICStus [12], and in Java. The SNMP functions were implemented using AdventNet's SNMP API 3.0 [1].

All of the event scenarios we have written are generic and topology independent. We would like to develop a large set of pre-defined scenarios, component-models, and problem identification heuristics to improve the out-of-box experience. In the next few months Yemanja's scenarios will be tested in the Océano server farm. A performance evaluation of the engine will also be conducted at that time.

A graphical scenario editor that can simplify the development process and perform consistency checks will also be built. The editor will support the definition of component models and the building of scenarios for these models. Models will define the set input events it accepts, and the set of outputs events and problems it publishes.

Built-in predicates described in Section 2.4 have already been implemented and tested. We are working on improving their efficiency and ease of use. In particular, multiple predicates have been defined to execute the most common SQL queries such as connectivity or containment checks. A cache of query results is provided to reduce the overhead associated with performing database operations. We are also working on improving the organization of the internal open-problem database to guarantee search and insert efficiency.

5.1 Integration with other tools

Currently there are a number of commercial products available that can handle various aspects of fault management. Although none of them can satisfy all the requirements. Some of them may be used to provide certain functions within Yemanja including network monitoring and data collection, problem reporting, fault management of particular devices, or fault isolation within data link and network layers.

We have begun to integrate Yemanja with Tivoli Netview [24]. Letting Netview perform simple alarm correlation, e.g., filtering recurrent alarms, suppressing immediately canceled alarms, etc. The remaining Netview alarms will be passed to Yemanja. Netview will also be used for the collection of device statistics that may not be collected using RMON [25] probes.

Other research directions include:

- Scenario generation based on SLA requirements
- Techniques to tune threshold values
- Threshold violations and system bottleneck prediction
- Support for uncertainty and lost events

6 Related work

In the past various event correlation techniques were proposed including rule-based systems [18, 26], model-based reasoning systems [13, 21], model traversing techniques [14, 15], case-based systems [17], fault propagation models [9, 16], and the code-book approach [27].

Rule-based systems are composed of rules (productions) of the form *conclusion* **if** *condition*. The condition part is a logical combination of propositions about the current set of received alarms and the system state [18, 26]; the conclusion determines the state of correlation process. The operation of the system is controlled by an inference engine, which in fault management applications typically uses a forward-chaining inference mechanism [18, 21]. Rule-based systems are believed to lack scalability, to be difficult to maintain, and to have difficult to predict outcomes due to unforeseen rule interactions. Most of these difficulties are a result of hard-coded network connectivity information within the rules. Yemanja's scenarios contain no hard-coded configuration information. Additionally, By using a layered model we control the number of symptoms considered by any given scenario and eliminate unwanted rule side-effects and decrease the overall number of correlation rules.

Another group of approaches incorporate an explicit representation of the structure and function of the system being diagnosed. The representation provides information about dependencies between network components [11, 13, 14, 15, 16] or about cause-effect relationships between network events [9, 21]. The fault isolation process explores the network model to verify correlation between events. Model-based reasoning systems [13, 21] utilize inference engines controlled by a set of correlation rules, which contain model exploration predicates. Model-traversal techniques recursively search the dependency graph towards the failing object in an event-driven fashion starting from the component that symptoms refer to [11, 14, 15]. Fault propagation models [9, 16] provide heuristic symptom explanation algorithms aimed at satisfying some optimality criteria.

Event correlation systems based on a formal representation of network dependencies and structure represent an improvement over early rule-based systems by having the potential to solve novel problems, and by being more expandable. However, the models that they require are difficult to obtain and keep up-to-date. The computational complexity involved in model traversals limits the scalability of the fault isolation process. In contrast, Yemanja does not maintain an explicit network model. Instead, it provides scenario templates organized on a hierarchically based network structure, which are instantiated with the data obtained from the arriving event attributes or from the configuration database. In addition, Yemanja's internal event publishers need not be aware which components consume the events that they forward; therefore, a change to higher-level scenario does not require changes to any of the lower level scenarios.

The code-book technique [27] uses a network model to derive a code – a set of possible symptom observations for every problem that may occur in the network. This process, called code-book generation, is performed in advance upon the installations particular network topology. Code-book generation eliminates the runtime computational complexity involved in model traversals. Network alarms observed over a certain time window constitute a coded problem to be decoded using the code-book based on the minimum Hamming distance metric. The code-book technique is very efficient and is resilient to the noise in the alarm data. However, it is difficult to apply to the correlation of transport and application layer events since relationship changes between managed objects, which are frequent in higher-layers, require reconfiguration of the code-book. Additionally, multi-layer event correlation using a single correlation window is inadequate as events on different layers may have substantially different temporal relationships. Also, we can not perform tests or access the open problem database during correlation. Yemanja allows the time window to be specified separately for every event layer, which is based on the latency of the monitoring tool active in the particular layer. Testing and database lookup is easily incorporated into Yemanja's correlation scenarios.

Case-based systems [17] try to use experience gathered through past problem solving to find a solution for the new problem. The solution for the new problem is adapted form the solution of the closest matching problem solved in

the past. Case-based systems are able to learn correlation patterns and are resilient to network configuration changes. They do not have a problem with network model maintenance. However, they do not take advantage of the known knowledge regarding entity behavior, nor do they allow fault isolation to be combined with fault detection. The need to build a substantial case library before the system is able to isolate faults makes case-based systems difficult to apply to an evolving architecture, such as Océano. Yemanja correlation scenarios are based on the understanding of the system operation; therefore, they may be applied to a given system without lengthy periods of experience gathering.

Event correlation approaches may be further classified as state-based [11, 21] or stateless [16, 27]. Stateless systems typically are only able to correlate alarms that occur in a certain time-window. State-based systems support the correlation of events in an event-driven fashion at the expense of the additional overhead associated with maintaining the system state. Yemanja keeps two types of state information: (1) the state of the active correlation rules and (2) the state of the system resulting from previous correlation (*the open-problem database*). This second type of information allows Yemanja to correlate events that are distant in time, as well as those that occur within a short time window (typically 20 minuets or less).

7 Conclusions

Yemanja is being used in the Océano test bed to monitor network status and SLA requirements. It was developed with this type of dynamic environment in mind and is well suited for the task. Yemanja is capable of correlating low-level network events with high-level application performance alarms without explicitly having the two entities that receive the external events aware of each other. This is accomplished through the use of a multi-layer model based approach which encapsulates device and abstract component behavior. Since each network event is sent from a single device, building an entity-model that receives and processes all events sent from devices of its type, is a natural way to develop a correlation system. Entities published their state to the world in terms of generated events and opened-problems, and are unaware of who consumes the information. This approach greatly simplifies the process of adding and deleting entity-models, and keeps individual scenarios simple and small.

Yemanja does not hard-code configuration data into its rules. Instead it depends on a high performance cached configuration database. For this reason our scenario-set is independent of the network topology, and does not require modification when the topology changes. This also keeps our scenario set orders of magnitude smaller then the scenario sets used by systems having hard-coded information.

Yemanja scenarios can contain method calls, SNMP and database configuration queries, delayed predicate invocations, etc.. These facilities allow the source of the problem to be pinpointed and tested in a single process. This integrated approach to fault detection, localization and isolation, provides for a maximum amount of flexibility. Scenarios that can not be captured in other more restrictive languages, can be easily represented in Yemanja.

Scenario rules are grouped and prioritized, providing a strait forward means to capture and rank multiple reasons for the same set of reported symptoms. Since the engine keeps track of each scenario an event is in, it knows when all scenarios have failed, been satisfied or timed out. This allows us to automatically remove events from the cache when they are no longer needed.

References

[1] Inc. AdventNet. AdventNet SNMP API 3.0. http://www.adventnet.com.

[2] K. Appleby, S. Fakhouri, J. Fong, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, M. Mei, D. Pazel, J. Pershing, B. Rochwerger, and A. Tal. The Océano White Paper. IBM Internal Article.

[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Océano – SLA-based management of computing utility. (submitted for publication).

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *A Simple Network Management Protovol (SNMP)*. IETF Network Working Group, 1990. RFC 1157.

[5] D. D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann Publishers, 1998.

[6] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie. *Definition of Managed Objects for Bridges*. IETF Network Working Group, 1993. RFC 1493.

[7] G. Goldszmidt and G. Hunt. Scaling Internet services by dynamic allocation of connections. In M. Sloman, S. Mazumdar, and E. Lupu, editors, *Integrated Network Management VI*, pages 171–184, Boston, MA, May 1999.

[8] R. Gopal. Layered model for supporting fault isolation and recovery. In J. W. Hong and R. Weihmayer, editors, *Proceedings of IEEE/IFIP Network Operation and Management Symposium*, Honolulu, Hawaii, Apr. 2000.

[9] M. Hasan, B. Sugla, and R. Viswanathan. A conceptual framework for network management event correlation and filtering systems. In M. Sloman, S. Mazumdar, and E. Lupu, editors, *Integrated Network Management VI*, pages 233–246, Boston, MA, May 1999.

[10] A. Hiles. *Service Level Agreements : Managing Cost and Quality in Service Relationships*. Chapman and Hall, 1993.

[11] K. Houck, S. Calo, and A. Finkel. Towards a practical alarm correlation system. In A. S. Sethi, Y. Reynaud, and F. Faure-Vincent, editors, *Integrated Network Management IV*, pages 226–237, Santa Barbara, CA, May 1995. Chapman and Hall.

[12] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *SICStus Prolog User's Manual*. http://www.sics.se/sicstus.

[13] G. Jakobson and M. D. Weissman. Alarm correlation. *IEEE Network*, pages 52–59, Nov. 1993.

[14] J. F. Jordaan and M. E. Paterok. Event correlation in heterogeneous networks using the OSI management framework. In H. G. Hegering and Y. Yemini, editors, *Integrated Network Management III*, pages 683–695, San Francisco, CA, Apr. 1993. North-Holland.

[15] S. Kätker. A modeling framework for integrated distributed systems fault management. In C. Popien, editor, *Proceeding of the IFIP/IEEE International Conference on Distributed Platforms*, pages 187–198, Dresden, Germany, 1996.

[16] I. Katzela and M. Schwartz. Schemes for fault identification in communication networks. *IEEE Transactions on Networking*, 3(6), 1995.

[17] L. Lewis. A case-based reasoning approach to the resolution of faults in communications networks. In H. G. Hegering and Y. Yemini, editors, *Integrated Network Management III*, pages 671–681, San Francisco, CA, Apr. 1993. North-Holland.

[18] G. Liu, A. K. Mok, and E. J. Yang. Composite events for network event correlation. In M. Sloman, S. Mazumdar, and E. Lupu, editors, *Integrated Network Management VI*, pages 247–260, Boston, MA, May 1999.

[19] M. Mansouri-Samani and M. Sloman. GEM – a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), Jun. 1997.

[20] K. McCloghrie and M. Rose. *Management Information Base for Network Mangement of TCP/IP-based internets: MIB-II*. IETF Network Working Group, 1991. RFC 1213.

[21] Y. A. Nygate. Event correlation using rule and object based techniques. In A. S. Sethi, Y. Reynaud, and F. Faure-Vincent, editors, *Integrated Network Management IV*, pages 278–289, Santa Barbara, CA, May 1995. Chapman and Hall.

[22] R. Perlman. *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*. Addison Wesley, 1999.

[23] S. H. Schwartz and D. Zager. Value-oriented network management. In J. W. Hong and R. Weihmayer, editors, *Proceedings of IEEE/IFIP Network Operation and Management Symposium*, Honolulu, Hawaii, Apr. 2000.

[24] Tivoli. *Netview for Unix: Administrator's Guide, Version 6.0*, Jan. 2000.

[25] S. Waldbusser. *Remote Network Monitoring Management Information Base*. IETF Network Working Group, 1995. RFC 1271.

[26] P. Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, and Z. Shi. Alarm correlation engine (ACE). In *Proceedings of IEEE/IFIP Network Operation and Management Symposium*, pages 733–742, 1998.

[27] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, 1996.