HEWLETT PACKARD

A novel approach to event correlation

Keith Harrison
Telecoms System Department
HP Laboratories Bristol
HPL-94-68
July, 1994

events, network management, TMN, correlation

Hewlett Packard Laboratories is responsible for long-range research into new technologies for use by the Company. The mission of the Intelligent Network Computing Laboratory (INCL) based in Bristol, is to create key computing and system technologies to enable HP to be a leading player in telecommunications industry. We will present an overview of the research being undertaken by INCL in the area of the management of telecommunication networks and will pay particular attention to our research into Event Management and Alarm Correlation.

1. Introduction

The purpose of this paper is to give an overview into some of the research being carried out by HPLabs in the area of event correlation as part of a larger investigation into the open management of large-scale heterogeneous telecommunication networks.

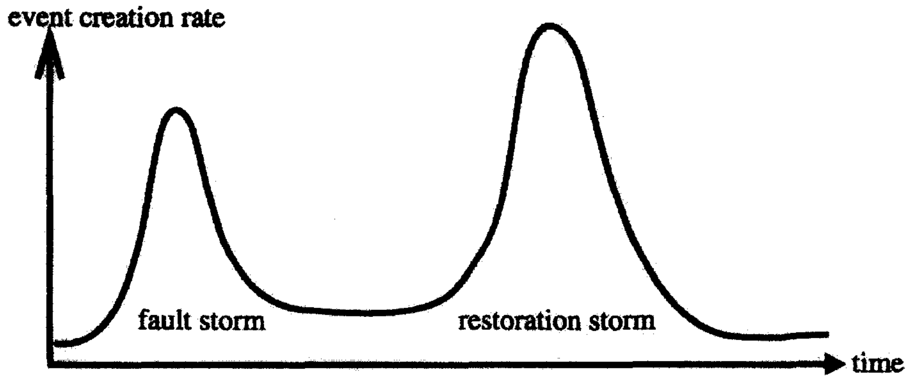
We define **Event Correlation** to be processing that involves multiple events - as opposed to **Event Filtering** that treats each event independently.

2. Background

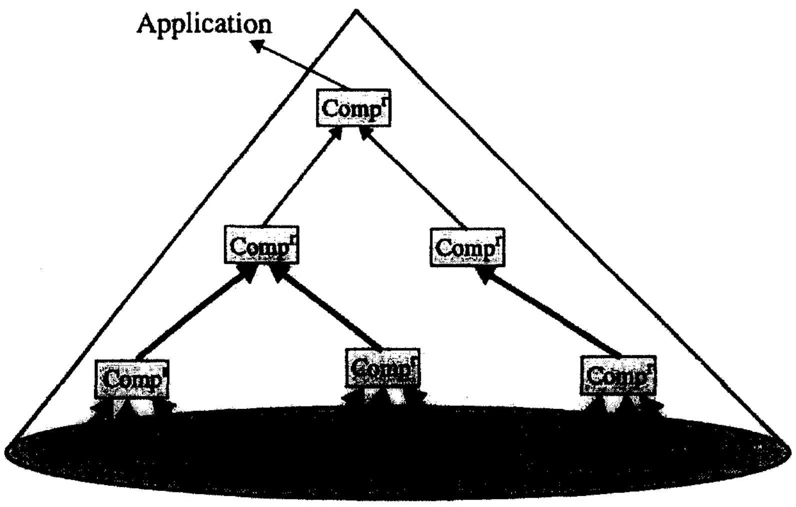
This work on event correlation was motivated by two real-world concerns: the handling of large volumes of events from a single fault, and the determination of the fault that underlies the observed events.

2.1 Volume of Traffic

The first concern is that of volume. When a fault occurs on a network, the network emits a short-lived storm of events. For example, a break in a SONET cable may result in a storm of many thousands of events. Similarly, when the cable is restored, a restoration storm of similar size may occur.



Each event is typically shipped across a data network to one or more network management applications. Somewhere along the way it will be logged for audit purposes. As higher bandwidth transmission standards are adopted, such as OC192, this event storm will reach epic proportions - it is not infeasible for an OC192 break to result in excess of one million events. Ideally, we should think about reducing this traffic by performing some form of *data compression*, and by performing this as early as possible. Obviously, this data compression should be done in such a way that no useful information is lost. An architecture for achieving this might look like:



The events emitted by the network are sent to local ***data compressors*** where their bulk is reduced. This reduced flow of events is then sent to regional ***data compressors*** and so on, up to the central network management application.

It would defeat the purpose of the architecture if all events had to be sent to the application for logging. There is an assumption that logging will be performed locally, and if a consolidated log is required, then this can be performed off-line during the quiet periods.

2.2 The Cause Not the Symptoms ...

The operator is not interested in the actual events. They are more interested in knowing what the underlying fault is that triggered the events. This task has been traditionally attacked using expert systems technology and involves a sophisticated form of pattern matching. It is assumed that a fault produces a standard pattern of events - detect the pattern and you can determine the fault.

Both problems, described in 2.1 and 2.2, may be solved by using a pattern matching technology. In the first case we have heavy traffic but the patterns being detected are typically very simple. In the second case, we can take advantage of the reduced event traffic, but the patterns being detected are more sophisticated.

This gives us a design objective:

Objective:

Develop a single technology that can tackle both the data compression of large volumes of events AND the determination of the underlying pathology.

In practice, expert systems can be developed that are considerably more powerful than pattern-matchers. The assumption is that the 80-20 rule applies. That is, a large percentage of the problems that arise in a network have been planned for and their event patterns predetermined. It may well be necessary to use Expert System technologies to handle the remaining problems however this can take advantage of the much reduced, and enriched, stream of events emitted by the *data compressors*.

3. The Use of Expert Systems Technologies

Event correlation has, to date, been tackled in a variety of ways:

* Using specially written application code
* Using table driven pattern matching code
* Using Expert Systems Shells, such as ART or G2

Writing specialised code is expensive and time consuming although the results can run impressively quickly. The disadvantage being that the code is inflexible and difficult to enhance and maintain.

Using expert system shells permits the designer to concentrate on the problem of specifying the required correlation rules without having to worry about other housekeeping tasks at the possible cost of efficiency - and, in some cases, unpredictability of response times.

Event Correlators typically lose interest in events once the events become old. For example, it is unlikely that any correlator would be interested in an event that is more than 24 hours old. Because we cannot, and do not want to, store events within the correlator indefinitely, we are required to find a strategy for releasing out-of-date events. "Events don't die they just fade away."

This gives us a second design objective:

Objective:

Develop a technology that can offer efficient and continuous operation over long timescales.

4. What is an Event?

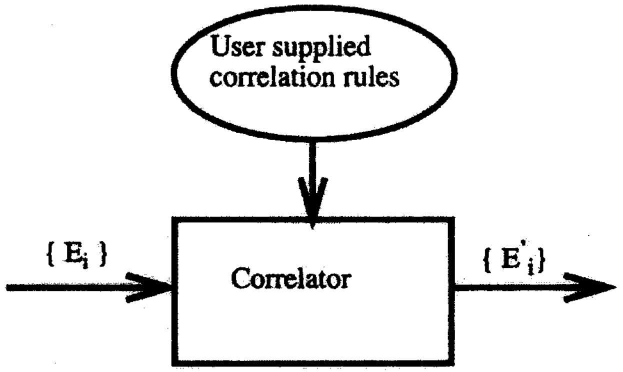
In order to obtain a solution that is as general as possible, and thus more amenable to reuse, it is necessary to determine the essential characteristics of events. The chosen set is as follows:

* An event has a type
* Each event is timestamped upon creation
* Events have to be sent from the issuing device to the correlator. This transit delay will take a random but bounded length of time.
* Events may arrive out of order.

Note that in order to correlate we are going to have to be able to say whether one event was created before or after another and what is the time difference. This requires the events to be timestamped at the time of creation - using a global time reference. In practice it is sufficient to know which device issued the event and how far adrift its clock is. A second problem occurs because the clock resolution is not always fine enough. For example, TL1 typically has a one second clock resolution. In this case, unambiguous ordering is going to necessitate the event emittor to tag the event with an incrementing counter.

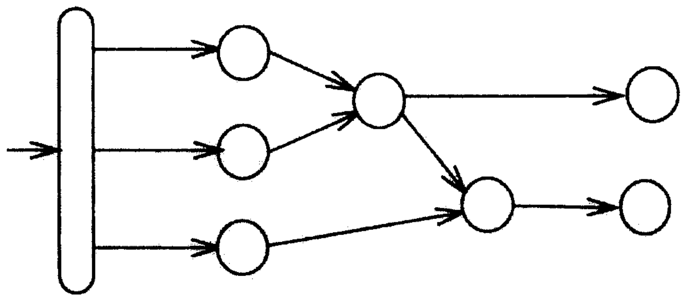
5. The Basic Idea

The event handling architecture described in section 2.1 assumes the existence of a *data compressor* that is capable of accepting multiple streams of events, correlating the events, and then creating one or more streams of events for use elsewhere.



The resulting stream of events may be a subset of the original stream, or comprise events of a new type. This correlator should be driven, or controlled, by a user supplied specification. It should also be capable of being efficient and scalable, both in terms of the arrival rate of events and the complexity of the correlations to be performed.

The key insight was to realize that this macro-level architecture remains valid when used at the micro-level. Thus, the internals of the correlator might have the following structure:



The question then becomes - can we design a small number of simple, efficient and easily understood nodes that may be combined together to provide processors with desired behaviour?

It is easy to see that this architecture is suitable for parallel processing.

This concept is not dissimilar to that found within asynchronous logic design. Digital circuits are constructed by combining NAND, NOR and NOT gates in the appropriate manner. In order to ease this task, standard combinations of these primitives have been packaged as multiplexors, half-adders, flip-flops or even microprocessors.

It is important to recognise that our "hardware" components are implemented in software - and thus may be easily customised.

6. An Example

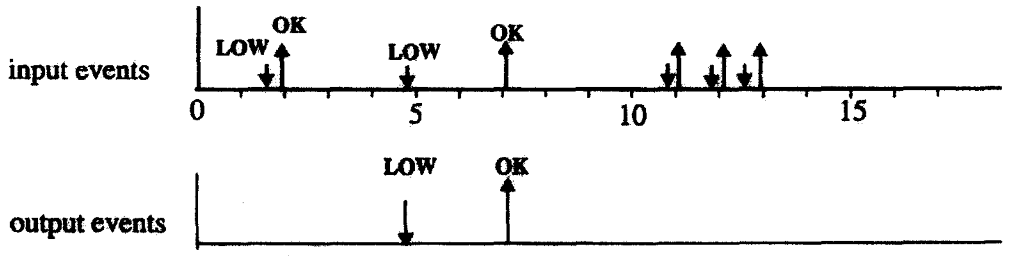
A detailed description of all of the standard components is beyond the scope of this paper. To give a flavour we will consider a very simple problem:

Imagine we have a hardware device that emits two types of events -

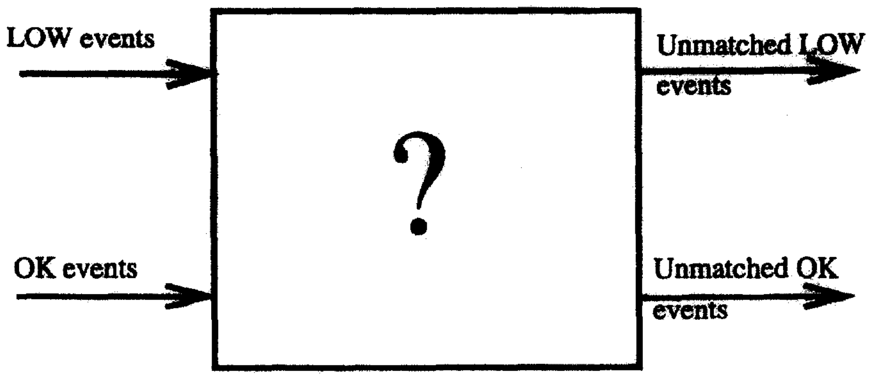
> LOW -- the power level is getting dangerously low
> OK -- the power level is acceptable again.

We might require only those LOW events which are not associated with a power loss transient. That is, the LOW power event is not immediately followed by a power OK event.

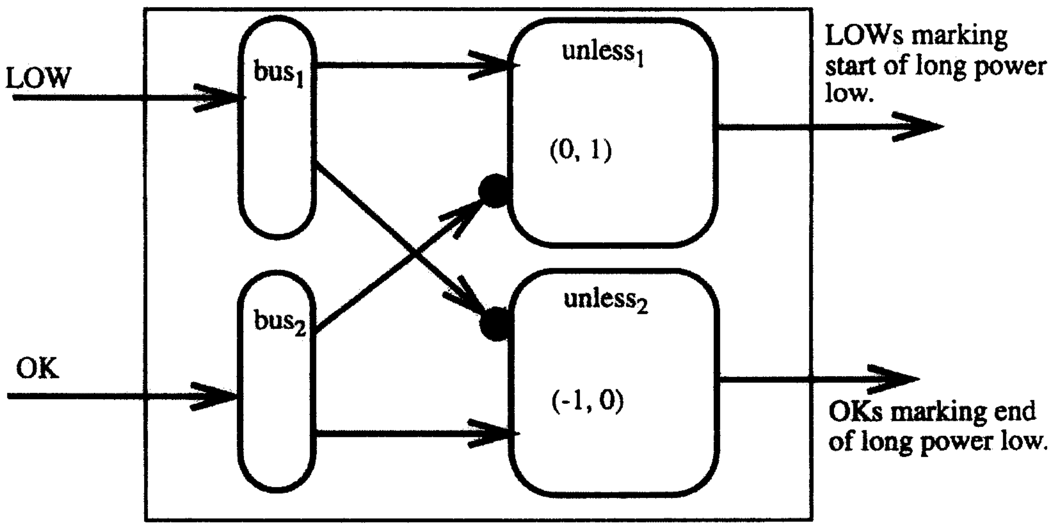
Similarly we want to see those power OK events that mark the end of such a long power shortage. Thus



That is, we wish to specify the circuitry needed to complete the following:



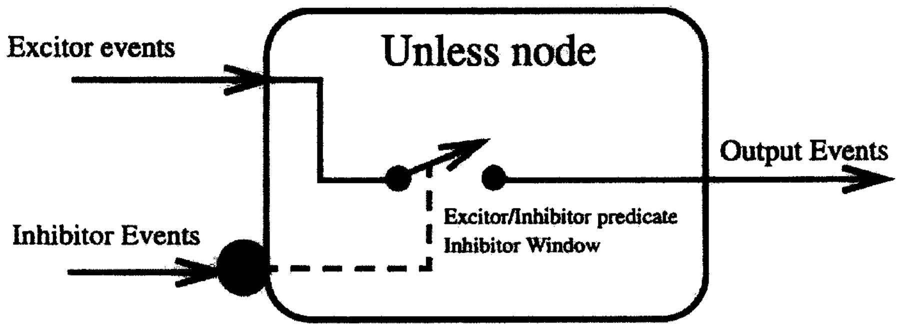
One solution, making use of two types of nodes is as follows:



LOW
OK
bus$_1$
bus$_2$
unless$_1$
(0, 1)
unless$_2$
(-1, 0)
LOWs marking start of long power low.
OKs marking end of long power low.

Any event entering a Bus node is forwarded to each of the successor nodes.

The Unless node has the property that events presented to the excitor input will be outputted unless a matching event arrives at the inhibitor event. We can visualise the Unless node as:



The match criterion contains two pieces of information:

1. What is the required time relation between the inhibitor and the excitor event? In the upper Unless node, unless$_1$, the LOW event is the excitor event and the OK event is the inhibitor event. The upper Unless node requires the OK event to have been created after the LOW event - but by no more than one second. Similarly, in the lower Unless node, unless$_2$, the OK event is the excitor event and the LOW event is the inhibitor event. The LOW event must have been created before the OK event, but by no more than one second.

2. What relation, involving event attributes, must be satisfied between the two events? In this case we require the LOW and OK events to have originated from the same device.

7. Processing Delays

This simple example illustrates an unavoidable fact of life. LOW events will need to be delayed until the system is quite sure whether or not a matching OK event is going to arrive. Even assuming there are no transit delays, this might result in a one second delay before the LOW is sent to its successor.

This gives us a third design objective:

Objective

Unnecessary delays should not be engineered into the system

Note: In the example, if we have no transit delays to worry about then the OK events will not be delayed because the inhibiting LOW event must have already arrived.

8. Memory Management

Once implemented, the event correlator is going to have to run for a long time, in some cases for several years. Consequently it is important that events be released as soon as possible. By examining the Unless node in detail, we will see that its memory requirements are well defined and lead to a simple and efficient memory management scheme.

The Unless node has two input streams of events - the excitor stream and the inhibitor stream. An event from the excitor stream will be outputted if, and only if, there is no matching event in the inhibitor stream such that the inhibitor event was created within a specified time window relative to the excitor event, and the two events satisfy some predefined criteria such as they come from the same device. These semantics are implemented as follows:

On receipt of an excitor event we must:

* check to see if an acceptable inhibitor event has already been received. This is done by searching through a memory containing recently received inhibitor events. If so, then processing is terminated.

Note: Even if we require the inhibitor event to have been created AFTER the excitor event it is still possible that, because of transit delays, the inhibitor event arrived BEFORE the excitor.

* check to see whether there is still time for an inhibitor event to arrive. If not, then the excitor event can be immediately output. If there is still time, then the excitor event is placed into an "excitor memory".

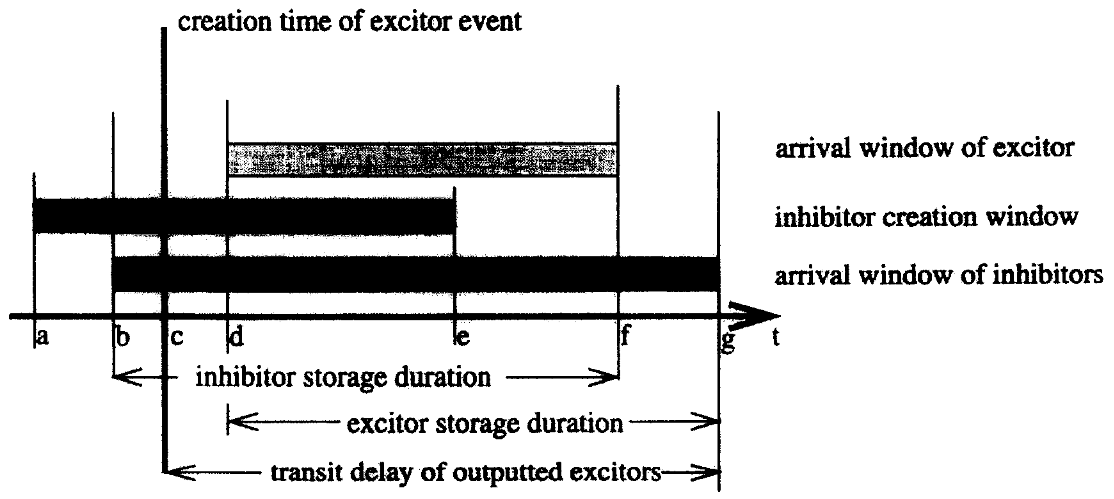
On receipt of an inhibitor event we must:

* Place the inhibitor event into an "inhibitor memory".
* check the store of excitor events. Any stored excitor events that matches may be removed- they have been inhibited.

Note: An inhibitor event may inhibit many excitor events.

Finally, periodically we must search the excitor memory. Any excitor event that is sufficiently old that no matching inhibitor event could possibly arrive may be output and removed from the excitor memory. Similarly, inhibitor events in the inhibitor memory will eventually become sufficiently old that no matching excitor events could arrive, in which case they may be removed.

In order to consider how long events need to stored, consider the following diagram:


creation time of excitor event
arrival window of excitor
inhibitor creation window
arrival window of inhibitors
a
b
c
d
e
f
g
t
inhibitor storage duration
excitor storage duration
transit delay of outputted excitors

The line at time c represents the creation time of an excitor event. Because of minimum and maximum transit delays for an excitor event, we do not know the precise time this event will arrive, but we do have an arrival window - the interval (d, f).

Associated with the Unless node is a time interval. Any inhibitor event that is a match for the excitor event must have been created within this window - the interval (a, e).

Because of the known bounds on the transit delays for the inhibitor events, we have an arrival window for the range of acceptable inhibitor events - the interval (b, g).

From this diagram it is possible to deduce the following:

> The earliest that the excitor event can arrive is d. But the latest that the matching inhibitor can arrive is g. Thus we may have to store the excitor events for (g - d).
>
> The earliest that the inhibitor event can arrive is b. But the latest that an interested excitor can arrive is f. Thus we may have to store the inhibitor events for (f - b).
>
> We cannot release an excitor event until we know the inhibitor event cannot arrive. Thus, we have to wait until time g before the excitor may be emitted. This means the excitor event emitted from the unless node will have a transit delay of (g - c).

An implication of the analysis given above is that although the Unless node has to have memory to store events we can predict, a priori, the maximum length of time we have to store the events. Any events that have passed their "sell-by date" may be released. Similar analysis is possible and required for all node types.

9. Summary

In this paper we have described a technology that satisfies the three design objectives. Namely:

1. develop a single technology that can tackle both the data compression of large volumes of events and the determination of the underlying pathology;

2. develop a technology that can offer efficient and continuous operation over long timescales;

3. unnecessary delays should not be engineered into the system.

The system is based on the novel notion of streams of events that are processed using simple processing elements that are composed to produce sophisticated correlations.

The system has been implemented and is running in the laboratory where the initial results have been found to be extremely promising. Work is in progress to test the system in a real environment to characterise its performance and to determine the level of support needed to aid the circuit designer.

Because an abstract definition of an event was adopted, this technology has found other uses apart from event correlation. For example, it can be used for consolidating billing records into a single billing record for the call.