HiFi+: A Monitoring Virtual Machine for Autonomic Distributed Management

Ehab Al-Shaer, Bin Zhang

School of Computer Science, Telecommunications and Information Systems
DePaul University, USA
{ehab, bzhang}@cs.depaul.edu

Abstract. Autonomic distributed management enables for deploying self-directed monitoring and control tasks that track dynamic network problems such as performance degradation and security threats. In this paper, we present a monitoring virtual machine interface (HiFi+) that enables users to define and deploy distributed autonomic management tasks using simple Java programs. HiFi+ provides a generic expressive and flexible language to define distributed event monitoring and correlation tasks in large-scale networks.

1 Introduction

The continuing increase in size and complexity and dynamic state changing properties of modern enterprise network increases the challenges on network monitoring and management system. Next-generation distributed management systems need not only monitor the network events but also dynamically track the network behaviors and update the monitoring tasks accordingly at run-time. This is important to keep up with significant changes in the network and perform recovery/protection actions appropriately in order to maintain the reliability and the integrity of the network services. Traditional network monitoring and management systems lack expressive language interfaces that enable distributed monitoring, correlation and control (actions). In addition, many of the existing management systems are static and lack the ability to dynamically update the monitoring tasks based on analyzed events. In request-based monitoring systems, the managers have to initiate large number of monitoring tasks in order to track events that might overload the monitoring agents and cause events delay or dropping. Next-generation monitoring systems must allow for defining complex monitoring actions or programs, instead of monitoring requests, in order to analyze the received events and initiate customized monitoring or management programs dynamically. For example, it will be more efficient to use a general traffic monitoring task for detecting network security vulnerability and initiate customized/specialized monitoring tasks when misbehaving (suspicions) traffic exits in order to closely track particular clients.

In this paper we present a monitoring virtual machine HiFi+ that explicitly addresses these challenges and provides a generic interface for multi-purpose monitoring applications. HiFi+ system supports dynamic and automatic customization of monitoring and management operations as a response to the change in the network behavior. This is achieved though programmable monitoring interfaces (agents) that can reconfigure their monitoring tasks and execute appropriate actions on the fly based on the use's request and the information collected from the network. HiFi+ employs a hierarchical event filtering approach that distributes the monitoring load and limits event propagation. The main contribution of this work is providing a Java-based monitoring language that can be used to define a dynamic monitoring and control tasks for any distributed management application. It also incorporates many advanced monitoring techniques such as hierarchical filtering and correlation, programmable actions and imperative and declarative interfaces.

This paper is organized as follow. In section 2, we introduce our expressive monitoring language. Section 3 gives application example. In section 4, we compare our work with related works. Section 5 gives the conclusion and identifies the future work.

2 HiFi+ Expressive Language Components

In this section, we present the three components of HiFi+ monitoring language: (1) Event Interface that describes the network or system behavior, (2) Filter Interface that describes monitoring and correlation tasks, and (3) Action Interface that describes the control tasks[2, 3]. HiFi+ is an object-oriented language implemented in Java. Users can use the event and filters interfaces to define the network behavior pattern to be detected and the action interface to perform the appropriate operation.

2.1 Event Definition Interface

An event is a significant occurrence in the system or network that is represented by a notification message. A notification message typically contains information that captures event characteristics such as event type, event source, event values, event generation time, and state changes. Event signaling is the process of generating and reporting an event notification. The HiFi+ Event interface allows users to use standard events like SNMP traps as well as defining a customized event specification. Figure 1 shows the class hierarchy of Event Definition Interface.

In HiFi+, although events can be in different formats, all types of events share the same interface and can be accessed and manipulated in the same way. For example, the special *SNMPTrap* event with fixed format encapsulates all the information in an SNMP trap message and these information can still be accessed by the general event function like *getAttributeValue()*. The *HiFiEvent* event format is the general event type which can be used to construct customized events. The *HiFievent* event can be divided into two parts: the event body and event id. The event id can be a string or an integer which stand for the event name or type. The event body is the container of the real information.

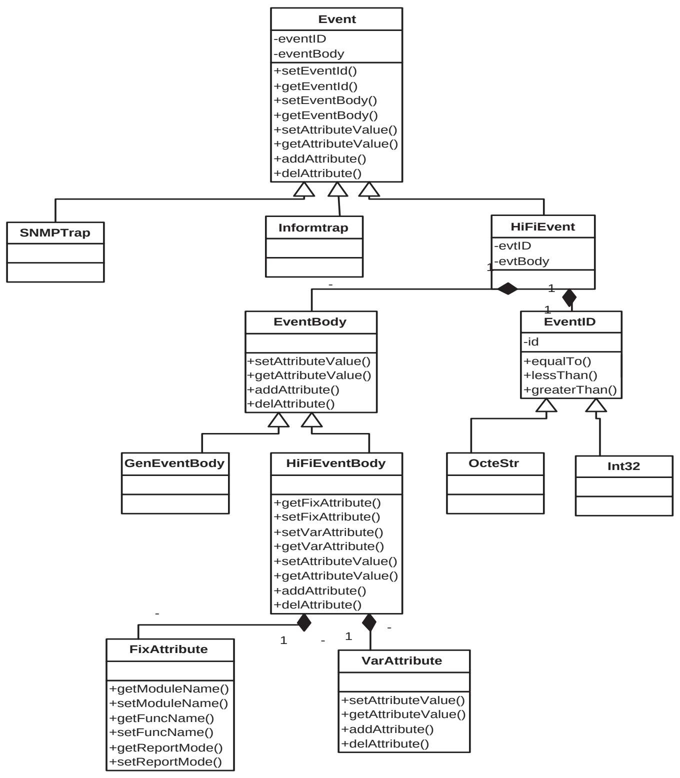

Fig. 1. Event Definition Interface Classes

Two types of event body are extended from the basic event body class: the general event body and *HiFiEvent* body. *HiFiEvent* body mainly has two parts: the fix attributes and the variable attributes. Both of these two parts are composed by a set of predicates. Each predicate has an attribute name, the value of that attribute and the relation between them. For example, $bandwidthUsage > 0.8$ is an event predicate which means the value of $bandwidthUsage$ attribute is larger than 0.8. The fixed attributes define the common attributes shared by all event types. From this part, we can get the information about the event source, generated time and signaling type. When event is created, the system automatically inserts the current time in the *timestamp* attribute of the event. The variable attributes allows user to define any additional general attributes that might reveal more information. For example, suppose we want to monitor the system load of the Web server *neptune*. We can define the format of the event generated by *neptune* as follows:

```
Event systemLoad = new HiFiEvent("systemLoad, "Neptune, loadMonitor,
senderThread", "cpuUsage = Any");
```

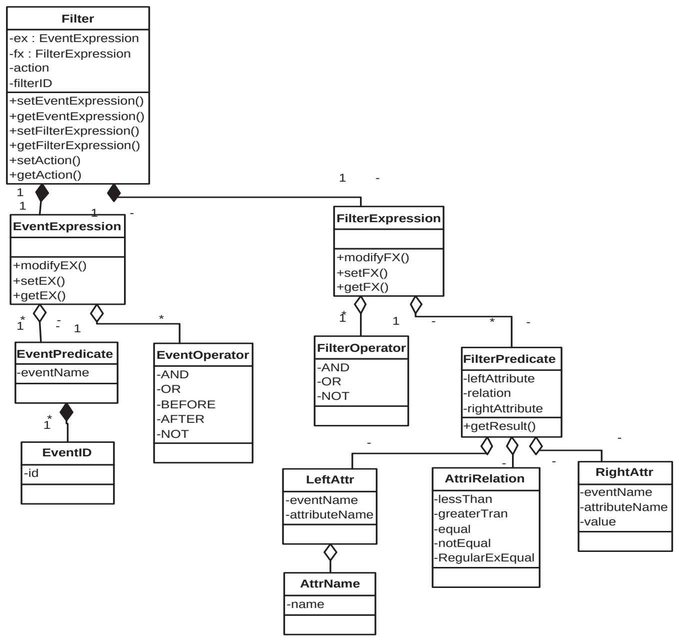

Fig. 2. Filter Definition Interface Classes

2.2 Filter Definition Interface

In HiFi+ monitoring virtual machine, users describe their monitoring demands by defining "filters" and submit them to the monitoring system at run time[2]. Figure 2 shows the filter classes hierarchy. A filter is a set of predicates where each predicate is defined as a Boolean valued expression that returns true or false. Predicates are joined by logical operators such as "AND" and "OR" to form an expression[3]. In HiFi+ language, the filter is composed by four components: filter ID, event expression which specifies the relation between the interesting events, filter expression which specifies the relations or the matching values of different attributes, and action object name. The action object will be loaded and executed by the monitoring agent if both the event and filter expressions are true. The event and filter expressions define the correlation pattern requested by consumers. Consumers may add, modify or delete filters on the fly. The filters are inserted into the monitoring system through filter subscription procedure[2]. The monitoring agent can reconfigure itself by updating their internal filtering representation. This feature is highly significant to provide dynamic features of programmable monitoring virtual machine.

Every filter has a filter id that is unique in the monitoring system. To illustrate the expressive power of the filter abstraction to define monitoring tasks, we will next show some examples. Assume we want to monitor the performance of our web server *neptune* and accept new connections only if the service time of the existing clients is acceptable. Therefore, if the simultaneous connections exceed a certain threshold and the connected clients experience unacceptable performance drop, then we need *neptune* to refuse more service requests. In this example, we need to monitor not only the system load of *neptune*, but also the performance drop in the client side. Assume *neptune* will send out *systemLoad* event periodically or when load is significantly increased as defined in the previous example. We are interested in events that reflect high increase of CPU Load (assuming 80%). The filter for this requirement can be defined like this:

```
Filter systemLoadFilter = new Filter("systemLoadFilter", "systemLoad",
"systemLoad.cpuUsage > 0.8 AND systemLoad.machine = neptune",
"systemLoadAction" )
```

The four parameters transferred to the filter constructor are filter ID, event expression, filter expression, and action class name. Suppose the web client will send out *performaceDrop* event if it experience long response time from a web server. The performaceDrop event format can be defined like this:

```
Event performaceDrop = new HiFiEvent("performaceDrop", "Any, webClient ,
senderThread", "responseTime = Any, server = Any")
```

The long response time experienced by web client can be caused by network congestion and packets drop or server load exceed its capacity. If only one web client complains about the long response time, it's hard to decide it's the server or the network causes this problem. If we receive multiple *performanceDrop* events come from different web clients, we have more confidence to suspect that the long response time may be caused by server overload. So the filter should keep a counter for how many clients have sent out *performanceDrop* events. When the counter value is larger than threshold (assuming 5), the filter will send out *serverOverloadAlert* event. We can define event and filter for this task like this:

```
Event serverOverloadAlert = new HiFiEvent("serverOverloadAlert", "Any,
performanceDropFilter, Any", "server = Any")

Filter performceDropFilter = new Filter("performaceDropFilter",
"performanceDrop", performanceDrop.responseTime > 120 AND
performanceDrop.server = neptune", "performanceDropAction")
```

The counter updating and the event creation are implemented in the action and not shown in the filter definition. Suppose if we receive the *serverOverloadAlert* event in ten seconds after receive *systemLoad* event, we can get the conclusion that the server has been overloaded. The filter for this task can be defined as follow:

```
Filter serverOverloadFilter = new Filter("serverOverloadFilter",
"serverOverloadAlert AND systemLoad", "systemLoad.timeStamp -
serverOverloadAlert.timeStamp < 10000", "serverOverloadAction")
```

2.3 Action Definition Interface

Actions describe the tasks to be performed when the desired event pattern (correlation or composition) is detected. In this part, we support programmable management interface. Users can write a Java program to perform any action to respond to detected network conditions. If the event and filter expressions in filter evaluate true, the monitoring agents load and execute the corresponding action programs.

Supporting customized action is one of the major objectives of HiFi+ monitoring virtual machine. The action class allows users to define monitoring task that can dynamically be updated. It provides a set of API to allow users to create their own action implementation which extended from action abstract class. The users can also execute scripts or binary files that will be loaded on-demand into the monitoring agents. The action class supports five different action types: (1) activating/adding a new filter to the monitoring system or deactivating/removing an existing filter from the system, (2) modifying the filter expression of an existing filter to accommodate changes in the monitoring environment, (3) forwarding the receiving event to agents, (4) creating new events as a summary of previous event reports, and (5) executing a shell or binary program. The action class is actually a Java program extends "Action" abstract class, and thereby all standard Java as well as HiFi+ API can be used in an extended action class. This offers great flexibility to customize the monitoring system.

The action interface also provides "virtual registers" that the action developer can use to store event information history. The user can dynamically create and update registers in the action program, and these registers will be used locally and globally by the monitoring agents during the monitoring operations. Figure 3 shows the classes hierarchy of the action interface. To implement an action program in HiFi+ system, user defined actions must extend the action class and override the *performAction()* method to specify his action implementation. When action class is loaded and executed by the monitoring agent, the *performAction()* method will be invoked with three arguments: *EventManager, FilterManager, and ActionManager*. The *EventManager* has the methods by which the user can access and analyze the received events, create and forward events. The *FilterManager* lets the user activate (*addFilter()*) and deactivate (*delFilter()*) filters in the system or update the filter expression (*modifyFX()*). The *actionManager* allows users to execute script or binary file and create or update virtual register (*create/get/check/deleteRegister()*).

The action class provides rich event management functions that have a significant impact on the language expressiveness. Events can be retrieved based on its time-order, event type, event name, value of event attribute and so on. For Example, users can get all events sent by host *neptune* by invoking the following function: *getAnyEventQueue("machine=Neptune")*. On the other hand, we can use *getEventQueue("systemLoad", "machine=Neptune")* method to find all the *systemLoad* events sent by *neptune*. In addition, event queues can be sorted based on a specific attribute value.

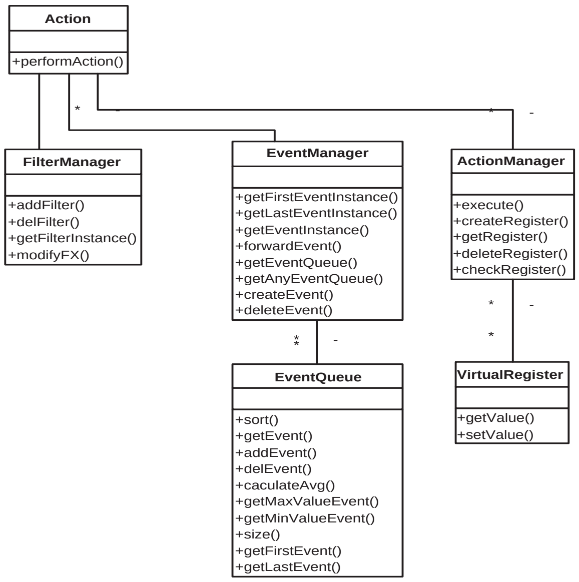

Fig. 3. Action Definition Interface Classes

Now, let's us show example of action programs using HiFi+ virtual machine language. In the web server performance monitoring example discussed in section 2.2, we didn't define the action programs for those filters. The action program for *performanceDropFilter* filter is a good example to show how to use virtual register.

```
public class performanceDropAction extend Action{
  public void performAction(EventManager EM, FilterManager FM,
  ActionaManager AM){
  Register reg;
  if(reg = AM.createRegister("hostCounter") !=void)
    reg.setValue(1);
  else{
    reg = AM.getRegister("hostCounter");
    String host =
    EM.getLastEvent("performanceDrop").getEventAttribute("machine");
    if (EM.getEventQueue("systemLoad", "machine ="+host).size()==1){
    reg.increaseValue();
    if(reg.getValue() >=5){
     AM.deleteRegister("hostCounter");
     Event evt =
     EM.createEvent("serverOverloadAlert", "server =neptune");
     EM.forwardEvent(evt);}}}
   }
}
```

In this action, we extract the IP addresses of the web clients and compare it with other events and update the counter. The counter should be kept outside the action program in virtual register so it can be referenced by next round execution. When receive first *performanceDrop* event, we use the action manager to create a virtual register to store the counter (line 4). Then we initialize the counter (line 5). For the following events, we use the action manager get the register (line 7). Then in line 8, we get the IP address for the web client who sends the *performanceDrop* event. We search all the received events to find the events come from the same host with the last received event and put these events in an event queue (line 9). If the event queue size equal one, that mean this is the first time that web client send out *performanceDrop* event. Then we increase the counter (line 10). When the register value is equal or larger than 5, we delete the register and create and forward the *serverOverloadAlert* event (line 11-14).

3 Application of HiFi+ in Distributed Intrusion Detection

In this section, we will show an example of how HiFi+ can be used in intrusion detection systems (IDS) to detect DDoS attack. DDoS attacks usually launch number of aggressive traffic streams (e.g., UDP, ICMP and TCP-SYN) from different sources to a particular server. This example shows how HiFi+ can be used to support IDS devices in deploying security (signature-based or anomaly-based) monitoring tasks efficiently. In [10], a proposal for an attack signature was presented to detect DDoS by observing the number of new source IP addresses (not seen during a time window) in the traffic going to a particular server. We here will implement a variation of this technique using HiFi+ interfaces.

We will, first, monitor the load of the target servers using *systemLoadFilter* filter. If any of these filters indicates that the system load of a server goes beyond a specific threshold, then the *diffSrcCheckFilter* filter will be activated in order to monitor all new tcp connections initiated to this target server. The *diffSrcCheckFilter* filter receives and filters all *tcpSyn* events that represent TCP-SYN packets destined (i.e., destination IP) to the target server. The *tcpSyn* events can be generated by network-based intrusion detection system (IDS). As the *diffSrcCheckFilter* filter keeps track of *tcpSyn* events, it calculates the number of different IP sources seen within a one-second time windows. If the number of different IP sources is larger than a specified threshold, then the *diffSrcChcekAction* will create *diffSrcExceedThr* event. Here we need point out the difference between our approach and the approach proposed in[10] is that we don't need keep the history of IP source for every server which makes our approach suitable for large-scale network with many target servers. Finally, we use the *DDosFilter* filter to correlate the *systemLoad* and *diffSrcExceedThr* events. Only when these two events occur within a close time window from each other and they are both related to the same server, then we can conclude the server is under DDOS attack. Let us assume that the DDOS signature will be defined like this: if the CPU usage on a server increases beyond the 0.6 and within one second we detect that there are more than 100 different IP source addresses starting tcp connections to that server, we will report DDOS attack. The events and filters used in this monitoring task can be defined as follows:

```
Event tcpSyn = new HiFiEvent("tcpSyn", "Any, Any, Any",
"sourceIP = Any, destinationIP = Any")

Event diffSrcExceedThr = new HiFiEvent("diffSrcExceedThr", " Any,
diffSrcCheckFilter, Any", "targetIP = Any")

Event systemLoad = new HiFiEvent("systemLoad", "Any, loadMonitor,
senderThread", "cpuUsage = Any, diffSrcThr = 100");

Filter systemLoadFilter = new Filter("systemLoadFilter", "systemLoad",
"systemLoad.cpuUsage > 0.6 OR systemLoad.cpuUsage <0.3",
"systemLoadAction" )

Filter diffSrcCheckFilter = new Filter("diffSrcCheckFilter", "tcpSyn",
"tcpSyn.destinationIP =Any", "diffSrcChcekAction")

Filter DDosFilter = new Filter("DDosFiler", "diffSrcExceedThr AND
systemLoad",  "diffSrcExceedThr.targetIP = systemLoad.machine AND
systemLoad.timeStamp - diffSrcExceedThr.timeStamp < 1000","DDosAction")
```

Next, let us look at the action programs for *systemLoadFilter* filter. This action has three tasks: (1) activating the *diffSrcCheckFilter* filter to detect DDoS attack if the system load is beyond a threshold, (2) forward the *systemLoad* event, and (3) deactivate (or deleting) the *diffSrcChekcFilter* filter if the system load drops below the threshold because the DDoS investigation is not needed any more. In this action program we dynamically change filter expression of *diffSrcCheckFilter* filter. The original filter expression will check *tcpSyn* event for any server to find if there are more than threshold different IP want to connect to that server. After the filter expression is updated, the filter checks only the *tcpSyn* events with the destination IP of the suspect target server.

```
public class systemLoadAction extend Action {
 public void performAction(EventManager EM, FilterManager FM,
 ActionaManager AM){
   Event evt = EM.getLastEvent("systemLoad");
   float load = evt.getAttributeValue("cpuUsage");
   if (load > 0.8){
     String target = evt.getAttributeValue("machine");
     FM.modifyFX("diffSrcCheckFilter", "tcpSyn.destionationIP =" +
     target);
     FM.addFilter("diffSrcCheckFilter");
     EM.forward(ent);}
   if(load < 0.3)FM.delFilter("diffSrcCheckFilter");}
 }
```

Next, let us look at the action program of the *diffSrcCheckFilter* filter below. In line 4, we get the time stamp for last *tcpSyn* event. This event triggers the execution of action program, so the destination IP must equal the target server. We get the time t2 which is 1 second before last event in line 5. Then we delete the outdated or irrelevant events whose time stamps are less than t2 or whose IP destinations are different than the target server (line 7). We then get the rest of *tcpSyn* events and put them in an event queue (line 8) and create a set to store the source IP addresses (line 9). In lines 10-13, we go through every event in the queue and putting the source IP in source IP set. Then we get the threshold for different source IP in line 14. In lines 15-18, we check the size of the source IP set. If the size is larger than threshold, we create and forward the *diffSrcExceedThr* event.

```
 1  public class diffSrcCheckFilter extend Action {
 2    public void performAction(EventManager EM, FilterManager FM,
      ActionaManager AM){
 3      Event evt = = EM.getLastEvent("tcpSyn");
 4      int t1 = evt.getEventAttribute("timeStamp");
 5      int t2 = t1 - 1000;
 6      String targetIP = evt.getEventAttribute("destinationIP");
 7      EM.deleteEvent("tcpSyn", "tcpSyn.timeStamp <  " + t2  + " OR
        tcpSyn.destinationIP != " + targetIP);
 8      EventQueue queue= EM.getEventQueue("tcpSyn");
 9      Set IpSource = new HashSet();
10      for (int i=0; i < = queue.size(); i++){
11        String sourceIP =
          (queue.getEvent(i)).getAttributeValue("sourceIP");
12        IpSource.add(sourceIP);
13      }
14     int threshold =
       EM.getLastEvent("systemLoad").getEventAttribute("diffSrcThr");
15      if (IpSource.size()> threshold){
16       Event evt = EM.createEvent("diffSrcExceedThr,", "targetIP
         =’’+  targetIp );
17       EM.forwardEvent(evt);}
18  }
```

Finally, let's look at the action program for DDosFilter. It just create and forward the *DDosAlert* event.

```
public class DDosAction extend Action {
  public void performAction(EventManager EM, FilterManager FM,
  ActionaManager AM){
    EM.createEvent("DDosAlert");}
}
```

4 Related Works

Numbers of monitoring and management approaches based on event filtering have been proposed in [1, 6–8]. Many of these approaches focus on event filtering techniques such as performance and scalability. But less attention was given to provide flexible programming interfaces as described in this paper.

Hierarchy filtering-based monitoring and management system (HiFi) was introduced in [2, 3]. HiFi employs an active management framework based on programmable monitoring agents and event-filtter-action recursive model. This work is an extension of HiFi system to provide an expressive and imperative language based on Java. The user can get benefit from the new API by implementing really complex action programs using known programming language.

A general event filtering model has been discussed in [5]. But this approach can filter the primitive events based on attribute values only, thereby doesn't support event correlation. SIENA, a distributed event notification service has been described in[4]. The programming interface of SIENA mainly provides functions for the user to subscribe, unsubscribe, publish and advertise events. It doesn't provide functions for the user to aggregate and processing events.

High-level language for event management is described in READY event notification system [6]. In READY, matching expressions are used to define the event pattern. The matching expression and actions in READY have same abstraction level similarity with filter and action in HiFi+. But the action types in READY are limited, only assignment, notify and announce action are supported. HiFi+ approach allows the user define complex action to trace and analyze the event history, modify the monitoring tasks dynamically, aggregate information to generate new meaningful events or even execute scripts and binary files.

Java Management Extensions (JMX)[11] is a framework for instrumentation and management of Java based resources. JMX focuses on providing a universal management standard, so the management application will not rely on fixed information model and communication protocol. HiFi+ focuses on supplying users a flexible and expressive programming interface to define the monitoring tasks and appropriate actions.

The Meta monitoring system[9] is a collection of tools used for constructing distributed application management software. But in Meta, sensors (a function that returns program state and environment values) are static programs that are linked with the monitored application prior to its execution. This reduces the dynamism and the flexibility of the monitoring system. Unlike in HiFi+, the monitoring agent can dynamically be configured and updated.

5 Conclusion and future works

In this paper, we present flexible monitoring programming interfaces for distributed management systems. The presented framework, called HiFi+ virtual monitoring machine, enables users to expressively define events formats, network pattern or behaviors to be monitored and the management actions using simple Java-based filter-action programs. Filters can implement intelligent monitoring tasks that go, beyond just fetching the information, to correlate events, investigate problems, and initiate appropriate management actions. The HiFi+ virtual monitoring machine provides unified interfaces for distributed monitoring regardless of the application domain. We show examples of using HiFi+ in security and performance management applications; however, many other examples can be similarly developed.

Our future research work includes important enhancements in the language interfaces and the system architecture such as integrating more event operators, implanting safe-guard for infinite loops, improving the virtual registers abstraction, developing topology-aware agents' distribution.

References

1. S. Alexander, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie: High Speed and Robust Event Correlation. IEEE Communication Magazine, pages 433-450, May 1996.
2. Ehab Al-Shaer: Active Management Framework for Distributed Multimedia Systems. Journal of Network and Systems Management (JNSM), March 2000.
3. Ehab Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly: HiFi: A New Monitoring Architecture for Distributed System Management. Proceedings of International Conference on Distributed Computing Systems (ICDCS'99), pages 171-178, Austin, TX, May1999.
4. Antonio Carzaniga, David S. Rosenblum Alexander L. Wolf: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems (TOCS), Volume 19, Issue 3, August 2001
5. P. Th. Eugster, P.Felber, R. Guerraoui1, S. B. Handurukande: Event Systems: How to Have Your Cake and Eat It Too. 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), July, 2002.
6. Robert E. Gruber, Balachander Krishnamurthy and Euthimios Panagos: High-level constructs in the READY event notification system. Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications 1998, Sintra, Portugal.
7. Boris Gruschke: A New Approach for Event Correlation based on Dependency Graphs. Proceedings of the 5th Workshop of the OpenView, University Association: OVUA'98, Rennes, France, April 1998.
8. Mads Haahr and Rene Meier and Paddy Nixon and Vinny Cahill: Filtering and Scalability in the ECO Distributed Event Model. International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)
9. K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman: Tools for distributed Application Management. IEEE Computer, vol. 24, August 1991.
10. Peng, C. Leckie and R. Kotagiri: Protection from Distributed Denial of Service Attack Using History-based IP Filtering. Proceedings of ICC 2003, Anchorage, Alaska, USA, May 2003.
11. Sun Microsystems: Java Management Extensions (JMX). http://java.sun.com/products/JavaManagement/index.jsp