Active Management Framework for Distributed Multimedia Systems

Ehab Al-Shaer

Multimedia Networking Research Laboratory
School of Computer Science, Telecommunications and Information Systems
DePaul University
Chicago, IL 60604
ehab@cs.depaul.edu

Abstract

The successful deployment of next-generation distributed multimedia systems is significantly dependent on the efficient management support that improves the performance and the reliability of such applications at run-time. In this paper, we describe an active management framework based on programmable monitoring agents and event-filter-action recursive model. Active management enables users to define reconfigurable and self-directed monitoring tasks that can be automatically customized at run-time in order to track the system behavior. Using this active management framework, the monitoring agents can be programmed to modify their monitoring tasks dynamically based on observed events information, and initiate the appropriate management actions accordingly. This paper also emphasizes the importance of active management for supporting a scalable, highly-responsive and non-intrusive distributed management infrastructure. The presented framework, which is referred to as HiFi monitoring system, supports a comprehensive environment including code instrumentation, user subscription, agents administration, event filtering and action service. Examples of using HiFi in managing large-scale distributed multimedia systems are also shown.

Key words: Distributed systems management, active networks, multimedia systems, event correlation, application steering.

1 Introduction

The next-generation distributed multimedia systems are large-scale, resource intensive and highly dynamic. With the increasing demands of deploying large-scale distributed multimedia (LDM) systems, an efficient on-line monitoring and control has become an essential service for improving the performance and reliability of such complex applications. Examples of LDM systems include large-scale collaborative distance learning, video teleconferencing, distributed interactive simulation, and reliable multi-point applications. In an LDM environment, large numbers of events are generated by system components during their execution and interaction with external objects (e.g. users or processes). These events must be monitored to accurately determine the run-time behavior of an LDM system and to obtain status information that is required for management operations such as steering applications or performing a corrective action [7, 16]. However, the manner in which events are generated in an LDM system is complex and represents a number of challenges for an on-line monitoring system. Correlated events are generated concurrently and can occur from multiple locations distributed throughout the environment. This makes monitoring an intricate task and complicates the management decision process. Furthermore, the large number of entities and the geographical distribution inherent with LDM systems increases the challenge of addressing important issues, such as performance bottlenecks, scalability, and application perturbation.

HiFi active monitoring system is an attempt to deliver a management architecture that explicitly addresses the challenges and requirements associated with managing large-scale distributed multimedia systems. HiFi active monitoring system supports dynamic and automatic customization of the management operations as a response to changes in LDM systems behavior [19, 24]. This is achieved through programmable monitoring agents that re-direct their monitoring activities on-the-fly upon users' requests and based on the information (events) collected during the monitoring operation. For instance, instead of monitoring all events and processes in the system, the agents monitor a subset of events/processes and the monitoring activities expand based on the information of the generated events. Therefore, active monitoring reduces the monitoring space significantly and offers a scalable management architecture. The monitoring intrusiveness is also minimized because this architecture enables initiating few monitoring tasks (targets) at the proper time. In addition, HiFi active monitoring enables the agents to react spontaneously (e.g., corrective actions) which improves the management operations response time compared with human-in-the-loop model [21].

A number of monitoring approaches and systems for monitoring distributed systems have been proposed (e.g., [5, 11, 15, 17, 21, 22, 23]). Although some of these systems provide mechanisms for modifying the monitoring requests dynamically, these mechanisms are man-

ual (i.e., they require human intervention) and insufficient to support a programmable or self-directed monitoring tasks (actions) as described in this paper. In addition, they do not support a scalable and fine grain event filtering mechanism which is significant for monitoring "large-scale" distributed systems such as Internet-based applications.

This paper is organized as follows: Section 2 explains the monitoring model and language specifications, Section 3 gives an overview of HiFi monitoring architecture and process, Section 4 describes our active monitoring approach and its impact on the management infrastructure, Section 5 provides an overview of the monitoring system implementation, Section 6 illustrates, by example, the effectiveness of using HiFi for steering distributed multi-point applications, Section 7 shows the evaluation of HiFi performance, scalability and system perturbation, and Section 8 presents the summary and concluding remarks.

2 Monitoring Model

In order to present a complete abstraction of the active monitoring architecture, our work must include modeling the *application behavior*, the *monitoring demands*, and the *monitoring mechanism* with considering the design objectives presented in Section 1. The program behavior can be expressed in a set of events revealed by the application during execution. In our monitoring model, we call the monitored programs *event producers* which continuously emit *events* that express the execution status. An *event* is a significant occurrence in a system that is represented by a *notification message*. A notification message typically contains information that captures event characteristics such as event type, event values, event generation time, event source, and state changes. *Event signaling* is the process of generating and reporting an event notification. We classify two types of events used in our model: *primitive events* which are based on a single notification message, and *composite events* which depend on more than one notification message. In the monitoring language, the event format (notification) is a variable sequence of *event attributes* determined by the user but it has a fixed header used in the monitoring process. An event attribute is a predicate that contains the *attribute name* which typically represents a variable in the producer (i.e, program) and a value. The event format also determines the type of event signaling: *Immediate* to forward the generated event immediately, or *Delayed* to allow buffering or batching events in the producer before sending them. Table I shows the High-level Event Specification Language (HESL) in BNF. This event abstraction enables consumers (1) to specify any arbitrary event format in a declarative way, and (2) to construct a complex (multi-level) abstraction of a program behavior using composite events. In addition, the event abstraction enables the consumers/users to assign values to the event attributes and does not require specifying attribute type (e.g., `int, float or string`). This provides a simpler interface than CORBA IDL [11].

We call the monitoring objects (e.g., human or software programs) *event consumers* since they receive and present the forwarded monitoring information. The consumers specify their monitoring demands via sending a *filter* program via the *subscription process* which configures the monitoring system accordingly (see Section 3). The monitoring operation is an *event-demand-driven* model. In other words, the producer behavior is observed based on the event generated (event-based) and on the monitoring requests (subscription-based). Therefore, as illustrated in Figure 1, events received in the monitoring system are classified based on exiting filters. If an event is detected, the action specified in the filter is performed such as forwarding the monitoring information to the corresponding consumers. The filter and action specifications are described in Section 4. We show in Figure 1 three management applications of this model: *manual fault recovery* that requires forwarding the monitoring information, *automatic fault recovery* that permit the monitoring system to perform actions on the event producers, and *application steering* in which the application acts as an event producer and consumer in order to tune its parameters.

3 Overview of HiFi Monitoring Architecture

HiFi employs a hierarchical event filtering-based monitoring architecture to distribute the monitoring load in application environment. Based on a user's monitoring requests, the monitoring system determines the appropriate agent or set of agents within the hierarchy to be tasked with inspection and evaluation of application events. The monitoring system uses fine grain decomposition and allocation mechanisms to ensure that filtering tasks are efficiently distributed among the monitoring agents and prevent events propagation in the network. Since our focus in this paper is on the programmable monitoring environment in HiFi, we give a brief overview of the HiFi system and refer to [1, 3, 4] for more details.

Hierarchical Monitoring Agents: In HiFi monitoring system, the task of detecting primitive and composite events is distributed among dedicated monitoring programs called *monitoring agents* (MA). MA is an application-level monitoring program that runs independently of other applications in the system and that communicates with the outside world (including producers and consumers) via message-passing. HiFi has two types of MAs: *local monitoring agents (LMA)*, and *domain monitoring agents (DMA)* (see Figure 2). The former is responsible of detecting primitive events generated by local applications in the same machine while the latter is responsible of detecting composite events which are beyond the LMA scope of knowledge. One or more producer entities (i.e., processes) are connected to a local LMA in the same machine. Every group of LMAs related to one domain (geographical or logical domain) is attached to one DMA. These DMAs are also connected to higher DMAs to form a hierarchical structure for exchanging the monitoring information. Because

of the different roles of LMA and DMA, LMAs use Direct Acyclic Graph (DAG) [6], however, DMAs use Petri Nets (PN) in order to record and track the event history [9].

Subscription Process: The monitoring process starts by a consumer sending a *filter program* that describes the monitoring request to the local MA. The filter is validated and decomposed into subfilters (e.g. F1, F2,..,Fn) using the *decomposition algorithms* in such a manner that each one represents a primitive event [1]. Then, each decomposed subfilter is assigned to one or more LMAs using the *allocation algorithms* based on the event sources and application distribution. The decomposition and allocation algorithms are described in [1]. The monitoring system also determines the proper DMAs for evaluating the event and the filter expression of the filter program. When MAs receive delegated monitoring tasks (subfilters) [10], they configure themselves accordingly by inserting this subfilter in the filtering internal representation which is a direct acyclic graph (DAG) for LMAs and Petri Nets (PN) for DMAs [4]. This architecture alleviates any performance bottlenecks or scalability problems by distributing the monitoring load among MAs and limiting the events' propagation to the originating sources [1].

4 Techniques for Active Monitoring

The main goal of active monitoring is to offer dynamically customizable monitoring tasks. This provides a flexible management infrastructure that scales well with number of producers and causes a minimal overhead in the application environment as shown in Section 7. In addition, active monitoring, in HiFi, reduces the monitoring latency because monitoring agents can customize their monitoring tasks automatically without users involvement. In this section, we describe the HiFi active monitoring architecture and its impact on improving scalability and reducing the perturbation of the monitoring system.

4.1 Filter-based Programmable Agents

In HiFi, users (or event consumers) describe their monitoring demands via programs called *filters* submitted to the monitoring system at run-time. A filter is a set of *predicates* where each predicate is defined as a boolean-valued expression that returns *true* or *false*. Predicates may be joined by *logical operators* (such as `AND` and `OR`) to form an expression. In our model, the filter consists of three major components: the *event expression* which specifies the relation between the interesting event, *filter expression* which specifies the attributes value or the relation between the attributes of different events, and the *action* to be performed if both event and filters expressions are *true*. Table II shows the High-level Filter Specification Language (HFSL) in BNF.

Consumers may add, modify or delete filters on-the-fly through the subscription component interface described in Section 5. When a consumer performs filter subscription, the monitoring agents reconfigure itself accordingly by updating their internal filtering representation [4]. Different consumers can potentially send different filters simultaneously. However, since the dynamic subscription may create an inconsistency in the monitoring environment, HiFi uses the *subscription protocol*, described in [3], to ensure an atomic state update and synchronization among the monitoring agents.

As the event abstraction emphasizes the declarative aspect of the model, the filter-based programming abstraction enables consumers to describe the relation expression between different events and their attributes as well, which improves the expressive power and the usability of the monitoring language. This feature permits the agent to perform fine-grain filtering based on regular expressions. For example, assume agents have been configured through filter to detect warning events, `AudioWarning` and `VidWarning`, generated from Audio and Video processes, respectively. Consumers can limit the monitoring granularity by re-programming the MAs at run-time to detect only the *event correlation* between these two events such that they are both generated by the same machines via sending the following filter.

FILTER= $[(AudioWarning \wedge VidWarning)]$;
$[(AudioWarning.Machine{=}VidWarning.Machine)]$;
[FORWARD];`Warnings_Correlation_Filter`.

After decomposing and allocating this filter, the DMA will only forward *Audio* and *Video* warning events that are generated in the same machine. Consumers can also deactivate or modify an existing filter using *DEL* and *MOD* in the action part (see Table III). In addition, the programming environment permits consumers to overload the attributes values in the events in order to create a different event instance in the filter program.

4.2 Event Incarnation

Actions in the monitoring model can be simply *executing a program* (local or remote) or *forwarding* the detected event to the corresponding consumers which are both necessary for automatic fault recovery and application steering, respectively. In order to improve the dynamism and the expressive power of the monitoring system, the model provides more complex actions: event and filter incarnation. In HiFi, generating new events as an action is called *event incarnation* in HiFi. This feature improves the performance, expressive power, and usability of the active management system as follows:

- The event incarnation enables the consumer to activate a sequence of monitoring operations automatically using event-filter-action programming model (see Figure 1) without having the user to intervene in the monitoring process. This new event may trigger other filters which in turn causes performing further actions such as checking the status of other running processes. For example, a failure may occur in a producer (process) as a result of abnormal close of a communication channel (primitive event). In this case, the management operation involves *failure recovery* as well as *sending* an event that triggers a new filter to further diagnose the process that closed the connection.

- An action could generate a "summary" event which summarizes the information of detecting a composite event (e.g., the event expression consists of multiple events). This enables suppressing the information of multiple events into one event (summary event), thereby avoiding event report implosion and reducing the event traffic. For example, a monitoring agent may be requested to generate a summary event that conveys the *drop rate average* of a set of receivers. We will present an application example of summary events in Section 6.

- Performing an action such as executing a program may change the state of a running program. Therefore, sending an event that reveals the state change to the monitoring system is important to allow re-observing the behavior, and enabling automatic application steering. This significantly important for attaining system stability during the steering process as shown in Section 6.

4.3 Filter Incarnation

In addition to the manual reconfiguration via dynamic users subscription described in Section 3, HiFi active monitoring also supports programmable agents that reconfigure themselves automatically based on events occurrences. A filter action can be a filter manipulation (typically, adding a new filter, deleting a filter, and modifying a filter). Thus, for example, another new filter can be activated in the monitoring environment as a result of detecting an event. We call this *filter incarnation* and shown in Figure 1. Filter incarnation is formally defined in Table III of the monitoring language. Adding a new filter means activating a pre-defined filter that has not been submitted to the system. This is specified in the monitoring language using a special reserved word (`ADD`) with the pre-defined filter name. On the other hand, deletion or modification must be performed on an existing filter that consumers subscribed to. This is specified using the reserved words, `MOD` and `DEL`, with an active filter name. When modifying an active filter, consumers must specify which parts to modify: event expression (`EX`), filter expression (`FX`), or both. This can be designated by

appending the filter name as a prefix to `EX` and/or `FX`. The resulting EX and/or FX are the effective filter parts after the subscription is completed.

Filter incarnation enables users to define "general" monitoring tasks that can be automatically customized by the agents at run-time in order to track and diagnose specific system behavior such as failures or performance bottlenecks. This avoids overwhelming the system by a large number of "static" (hardwired) monitoring tasks to observe all or most system activities. Consequently, consumers can initiate filters that can modify (change or expand) their monitoring scope in order to include other monitoring targets (events and processes) whenever certain events are detected. In the following, we describe various applications of filter incarnation in active monitoring.

Adding/Deleting Filters for controlling monitor timing: Consumers can specify start and end times for any given monitoring activity based on events. In other words, consumers can specify to start/stop a monitoring activity when a certain event (primitive or composite) is detected. This minimizes the monitoring overhead and produces concise event traces. For example, assume a consumer wants to monitor the *drop rate* in the "receiving" events (*RecvEvent*) of *bar* program only when the *transmission rate* in the "transmission" event (*TransEvent*) of *foo* program drops below a certain threshold (*STHRESHOLD*). In this case, the consumer can specify a filter (*MonitorSender*) that monitors the "transmission" events of *foo* which will trigger another filter (*MonitorReceiver*) that monitors the "receiving" events of *bar* if the transmission rate drops below the threshold. The filters example is shown below.

FILTER= $[(TransEvent)]$;
$[(TransEvent.ModuleName =$ "foo" $\wedge\ TransEvent.transrate < STHRESHOLD)]$;
$[ADD\ \ MonitorReceiver]$;`MonintorSender`.

FILTER= $[(RecvEvent)]$;
$[(RecvEvent.ModuleName =$ "bar" $\wedge\ RecvEvent.droprate > RTHRESHOLD)]$;
$[FORWARD]$;`MonintorReceiver`.

The *MonitorReceiver* filter monitors "receiving" events (*RecvEvent*) from *foo* and forwards them to consumers if their *drop rate* exceeds RTHRESHOLD. Similarly, the *MonitorReceiver* filter can be deactivated (deleted) based on the *transrate* value changes in the *TransEvent*. This permits activating *MonitorReceiver* filter at the proper time automatically and minimizing the monitoring perturbation in the application environment.

Modifying Filters Specifications: Usually, monitoring tasks are static and defined prior to any monitoring operation. However, using filter incarnation, the filter information (e.g.,

attribute values) can be determined during the monitoring process itself based on the contents of the detected events. For this purpose, the HASL provides a set of virtual registers called *filter registers* that consumers can use for loading/restoring variables in/from monitoring agents. These registers are used by MAs to restore attribute values of received events. Consumers can simply assign the attribute value of an event used in EX or FX to a filter register and vice versa for this purpose. For example, consumers may want to monitor all events (trace) of the processes that have generated a security warning event (`WarningEvent`). In this case, the monitoring agents can not identify the *module name* information from the monitoring request (i.e., filter subscription). However, the monitoring agents can determine the module name during the monitoring operations. This is achieved by using the filer registers to save and restore the event information. In the following, the `DynamicErrorTrace` is a filter specification example that uses filter registers and filter incarnation to program the agent operations dynamically. In the following example, *ThisMod* is a filter register that restores the module name (*ModuleName*) after the occurrence of `WarningEvent` of type *SECURITY*. Then, the filer incarnation is used to modify the filter expression of *TraceProcess* filter in order to monitor/trace all events from this particular process/module only.

FILTER= $[WarningEvent]$;
$[(WarningEvent.ModuleName =$ "ANY" $\wedge WarningEvent.Type =$ "$SECURITY$")];
$[ThisMod = WarningEvent.ModuleName;$
$MOD\ \ TraceProcess.FX = (AnyEvent.ModuleName = ThisMod)]$; `DynamicErrorTrace`.

FILTER= $[AnyEvent]$;
$[(AnyEvent.ModuleName =$ "ANY")];
[FORWARD]; `TraceProcess`.

The *TraceProcess* is a "generic" filter that monitors and forwards all events from all modules to the corresponding consumers (Notice that "ANY" is a language keyword that indicates any string value). However, this general filter is customize by *DynamicErrorTrace* filter in order to perform a specific monitoring operation. As a result, this technique enables activating/deactivating the appropriate monitoring operations (or filters) at the right time (event), and avoiding the overhead of launching multiple filters or monitoring requests simultaneously. It also reduces the monitoring latency since monitoring agents can be programmed to react spontaneously without consumers intervention. Moreover, the filter incarnation feature provides an extendible programming environment utilizing the power of the event-filter-action recursive model as shown in Section 6.

5 HiFi Monitoring System Implementation

HiFi consists of four main components: *Instrumentation, Subscription Service, Event Filtering* and *Control*. Figure 3 shows the design of the monitoring system in component level and their interactions. Here, we provide an overview of the HiFi design components but detailed description is presented in [4] and [1].

Instrumentation Component: The process of inserting monitoring instructions inside the code of observed programs is called the *instrumentation process*. The instrumentation component utilizes the event information (HESL) supplied by the subscription component to construct low-level event formats called Event Reporting Criteria (ERC) that contains events information such as event location, reporting mode, event attributes, and the type of each attribute. The ERC is then used by the Event Reporting Stub (ERS) for constructing and reporting the event notification. The ERS is a HiFi library linked with the monitored application during compilation to facilitate event reporting. The main function of the instrumentation component is to facilitate the process of inserting the monitoring instruction (sensors) inside the program code. In many monitoring systems [17, 20], programmers write a considerable size of code for each generated event. This makes the instrumentation task tedious and error-prone. In HiFi, users only insert the *user sense* that specifies just the event name (e.g., `ReportEvent`(*WarningEvent*)). The instrumentation component then pre-processes the instrumented code and replaces user sensors with extended *system sensors* that contain all events related information such as machine name, process name, report mode and the attributes types and values. When the program starts executing, the control is transferred to the ERS which creates (i.e., forks) the LMA and initiates the agents organization protocol (described in [3]) in order to establish the agents' hierarchy described in Section 3.

Subscription Service Component: The monitoring subscription is the process by which the consumers would express their monitoring demands represented in *filter programs* using HFSL. There are two major subcomponents in the subscription service: (1) *Monitoring Language Processor* which is used for validating, parsing, and constructing the monitoring-knowledge necessary for distributing events and filtering tasks, and (2) *Monitoring Information Processor* which is used for decomposing and allocating such information to generate monitoring tasks executable by the monitoring agents, and packaging and disseminating the produced information such as event and environment information to the agent networks using reliable multicasting. The subscription component is also responsible for receiving and presenting the event notifications forwarded from the MAs.

Event Filtering Component: The event filtering component is the core component of the monitoring system and it constitutes the internal architecture of the monitoring agents, LMA or DMA. Its main functionality is (1) receiving and processing filtering tasks delegated from the subscription component after decomposition and allocation, and (2) inspecting incoming events based on the event attributes and the filters information (i.e., filter internal representation) to determine if this event is interesting (*detected*) or irrelevant (*rejected*) [4]. This component operates on the *event filtering internal representation* that represent the monitoring information such as consumers' subscriptions and event specifications. The internal filtering representation can be Direct Acyclic Graph (DAG) or Petri Nets (PN). In particular, because of the different roles of LMA and DMA, LMAs use DAG, however, DMAs use PN in order to keep track of the event history. In other words, each filter is implemented in the monitoring agents as sequence of DAG and PN nodes.

Control Component: The control component is provided to support reactive control management applications. The main function of this component is to perform the actions specified in a filter program. There are four types of actions supported by the monitoring architecture: program execution, information dissemination, event generation (incarnation) and filter incarnation (see Table III). As shown in Figure 3, the control component has two major subcomponents: *dissemination service* which uses multicasts information to the corresponding consumers, and *action service* which is used to execute local or remote programs, send a new event and/or perform filter incarnation (i.e., adding, modifying or deleting an existing filter).

6 Active Management for Steering Distributed Multi-point Applications

The HiFi monitoring system is being used in a number of management applications such as application steering, fault recovery and debugging of distributed multimedia systems. In this section, we present an example of using HiFi for monitoring and steering Reliable Multicast Server (RMS) [2]. One of the known problems in some reliable multicasting, is the effect of slow members (e.g., machines) in group communication. A machine is described as a slow machine if its receiving rate is "much" less than the other members in the group. In this case, a slow machine could typically slow down the communication to the entire group because the sender transmission rate, in RMS, eventually adapts to the rate of the slowest receiver. Developing a solution for slow members in multicast groups is beyond the scope of this paper. However, this application example is to show the effectiveness of HiFi active monitoring approach in supporting a *dynamic discovery* mechanism of slow members (or machines)

during a multicast session, and providing an *automatic feedback* to the RMS senders which make the proper steering management decision accordingly. The criteria of slow members are defined based on the user specifications. For example, the user (or manager) may define a slow member whose performance is below a certain threshold. In our example below, the RMS sender acts as a manager and sends the threshold information. Figure 4 shows the event (HESL) and the filter (HFSL) specifications used to discover slow members in multicast groups, and Figure 5 shows the Petri Nets of these filters as constructed in HiFi agents. Each RMS receiver is instrumented using HiFi instrumentation component to send *McastRec* event that contains the machine name, the domain name, the multicast group name, total bytes received (KBrec) so far, and number of NACKs scheduled (NackSch) which equals to the number of Nacks sent plus number of Nacks cancelled. Because of NACK suppression mechanism [8], the number of *NackSch* gives more accurate estimation of the *drop rate* than number of Nacks sent. The `McastRec` event is sent periodically based on time limit or maximum number of bytes received. And the RMS senders send `McastSend` events to indicate two things: the transmission rate (*TransRate*), and the drop rate threshold (*threshold*) for receivers in the group. Thus, the `McastSend` and `McastRecieved` events convey the status of the senders and the receivers, respectively. In order to activate this management operation at the proper time and avoid unnecessary use of resources, the slow members discovery process should be initiated only when the sender suffers some performance degradation (i.e., low transmission rate) due to the existence of slow members. This sender is called the *unhappy sender* and represented with *US* event in Figure 5. For this reason, the *TransRate* (in `McastSend` event) is first checked by the `MonMcastSender` filter (T1 transition in Figure 5) in order to identify the unhappy senders and their group names, and consequently activate other slow members discovery filters that monitor the receivers and update the threshold. If *TransRate* is found below the STHRESHOLD, then the `Slow_Members` filter is modified to get the *GrpName* value stored in the filter register as described in Section 4, and both `Slow_Members` and `Update_Threshold` filter are activated (T1 is fired in Figure 5). The activated `Slow_Members` filter compares the *NackSch* in `McastRec` and *threshold* in `McastSend` to identify slow members. However, because the threshold value is dynamic and may be determined from the overall performance of the participants, another filter (`Update_Threshold`) is used to provide a feedback on the overall drop rate average to senders which consequently re-adjust the threshold value accordingly. Each LMA forwards `McastSend` and `McastRec` primitive events to its DMA that evaluates the filter expression upon receiving both events. The second filter, `Slow_Members`, waits to receive one `McastSend` and `McastRec` events from *all* LMAs in the domain before the filter expression is evaluated. The `_ctr` and `_LMAs` are HiFi reserved key words and used to denote the number of the event occurrences and the number of LMAs in the domain, respectively. The filter expression evaluates to *true* if all RMS receivers in the domain send `McastRec` event from the indicated group name (*GrpName*) and the *NackSch* of one receiver or more

is higher than the threshold. If the filter expression becomes true, then T2 (in Figure 5) fires and three actions are performed: (1) the average of scheduled Nacks for receivers in same domain is calculated by CalcAVG action, which fires T3 transition, (2) the `DomAVG`, which represents a summary event, is sent to the containing DMA to reveal the domain average, and (3) the `McastRec` event that matches the slow member criteria represented in the filter expression (i.e., NackSch < threshold) is forwarded to the manager (RMS sender). The third filter, `Update_Threshold`, receives the `DomAVG` events from the DMAs and then calculates the total NackSch average, updates the threshold and sends the `McastSend` with new threshold to the LMAs/DMAs again. This causes T4 and T5 transitions to fire and an event S (in Figure 5) to be sent with the new threshold. This filter can be a DMA task, instead of RMS senders. However, users must provide the action *UpdateThreashold* to the DMA which then can update the threshold and notify the RMS senders automatically. Since senders or receivers could be members in various multicast groups, the group name (GrpName) is used to limit this management activities (filters) on the multicast groups suffering from slow members. In addition, another filter can be used to de-activate `Slow_Members` filter when the *TransRate* becomes less then STHRESHOLD. This way the slow members filter can activated and deactivated dynamically based on the condition of the multicast group. So in summary, three filters are used in this application: `MonMcastSender` that discovers unhappy senders and initiate the discovery process, `Slow_Members` that identifies receivers below threshold, and `Update_Threshold` that update the threshold value continuously base on the overall session performance.

The slow members and NackSch average information are collected from each receiver via LMAs and then combined and propagated in hierarchical fashion via DMAs to the RMS of the sender. This mechanism is scalable because it avoids the notifications implosion that may occur when `McastRec` are forwarded to one RMS sender from group of receivers. Furthermore, distributing the processing load such as calculating the average drop rate contributes to the monitoring performance.

7 Performance Evaluation

This section describes a performance evaluation study of the HiFi monitoring system. We conducted number of benchmarking and simulation experiments to assess the application *perturbation*, *scalability* and *latency* of HiFi. This section presents the numerical results as well as the conclusions obtained from this performance study.

Application Perturbation Measurements: The application perturbation can be measured by the execution time overhead caused by the monitoring operations including the

ERS event reporting process, and monitoring agents (LMAs and DMAs) operations. The first experiment is performed to evaluate the ERS overhead with respect to the event length. In order to show the effect of `EventReport()` function in ERS, used for constructing and reporting events, we compare its overhead with the traditional C `printf()` function which is frequently used by programmers as a simplest way for debugging and inspecting the program state and behavior. Figure 6 shows that the overhead of `EventReport()` is very comparable with `printf()` and ranges between 100 to 200 microseconds based on the event length. This figure also shows that ERS overhead grows slowly (about 2% per event attribute) when the event length is increased.

We then measured the actual overhead caused by the monitoring system including reporting time (ERS processing), primitive event filtering time (LMAs Processing), event correlation time (DMAs Processing), UNIX socket communication and the RMS communication. In order to measure this experimentally, we use a filter correlation example called *Hello World* filter. In this experiment, two event emulation processes called Random Event Generator (REG) are used to generate up to 5000 events randomly using Bernoulli distribution. Both REG processes are located in different hosts (Sun Sparc 5 running Solaris 2.5) in the same 10 Mbps Ethernet LAN. At each event time, REG process may choose to generate or not generate *Hello* or *World* events, which are 8 attributes long, with a probability of P. Each REG program is connected to an LMA residing in the same machine. The DMA sends a notification message to the manager if both *Hello* event and *World* event have the same sequence number (*TStamp*) but generated from two different REG processes (i.e., machines).

In these experiments, REG programs are fist run without instrumentation or monitoring. Then programs are instrumented and run in HiFi environment to detect the event of interest. Figure 7-A depicts the results of this experiment with various event generation probabilities (P). Several techniques are developed to minimize the application perturbation including *dynamic signaling* and *events batching*. Using dynamic signaling, ERS only generates the events which are contained in the expressions of any existing filter. Other events which are defined but not used in any filter program are suppressed by ERS, even if they are invoked by event producers. The event batching is a mechanism to buffer more events before sending them and it is used if *Delayed* mode is selected in HESL as described in Section 2.

To measure the effect of dynamic signaling, the REG programs were changed such that 50% of the generated events are filtered out by ERS. And to measure the impact of both dynamic signaling and event batching with maximum of 5 events, REG and ERS programs has been changed to reflect this effect. Figure 7-A shows a substantial improvement in reducing the application perturbation. These experiments also show a low perturbation figure, less than 8%, when P as low as 0.1 which what we believe a typical program generates in distributed environment. It is important to mention that agents communication primitives are the primary source of overhead in the monitoring operation [1].

Scalability Analysis: Our approach for evaluating the scalability of HiFi is by measuring the impact of increasing the event frequency and number of event producers on the mean response time or the monitoring latency. In this paper, we present the later case since the results are similar. The *monitoring latency* is the elapsed time between the event occurrence and the manager notification. In these experiments, we developed simulation routines to compare the Mean Response Time (MRT) [14] (monitoring latency) of hierarchical filtering approach with the centralized and decentralized monitoring approaches [5, 12, 13, 15, 17, 18, 22, 25]. The simulation routines assume that event arrivals (λ) are exponentially distributed, and the average monitoring/filtering service rate (μ) of an agent is 8000 events/second which was experimentally derived from a benchmarking experiments of LMA filtering. Therefore, the mean response time (MRT) for the centralized architecture is as follows:

$MRT_{centralized} = (1/\mu)/(1-(f*N)/\mu)$ such that $\lambda = f*N$ where f is the event frequency and N the number of event producers. In decentralized monitoring architecture, there are two levels of processing/filtering: in the producer agent, and in the centralized monitoring node. Thus,

$MRT_{decentralized} = (1/\mu)/(1-(f/\mu)) + (1/\mu)/(1-((f*0.5*N)/\mu))$

However, in HiFi which is hierarchical filtering-based monitoring, the latency is:

$Latecny_{hierarchical} = EGT + LFT + \sum_{i=1}^{h-1}(DFT_i + C)$ where hierarchy height: $h = \lceil log_x(N) \rceil$ such that x: branching factor and N: number of producers. The EGT and C can be neglected since it is a comparison study. Thus, the mean response time can be expressed as follows:

$MRT_{hierarchical} = (1/\mu)/(1-((0.5*f)/M)) + \sum_{i=1}^{h-1}((1/\mu_i)/(1-((f*0.5*x)/\mu_i)))$
The first factor represents the LMA filtering and the second represent the DMA filtering. Notice that the LMA receives only 0.5*f of the events since 50% of such event on average are filtered by ERS. Since not every event is necessarily forwarded up all the way in the DMA hierarchy, we calculate the MRT considering three different probabilities for forwarding an event: $10\%, 50\%$ and 90% of the events are forwarded. We also assume that there are 10 LMAs maximum are connected to one DMA in any domain (i.e., $x = 10$).

The simulation results of MRT verses number of producers of three approaches are depicted in and Figure 7-B. The event frequency (f) is considered 20 events per second. The figure shows the high superiority of the response time (latency) of the hierarchical architecture over the centralized and decentralized ones. The saturation points in the figures indicates a buffer overflow and indefinite response time since $\rho > 1$ in this case. The hierarchical architecture

with probability 0.9 is still superior over the other architecture because of the use of dynamic signaling. This figure also shows that the MRT of hierarchical architecture grow slowly with respect of number of producers. In fact, the "jump points" in hierarchical graphs in Figure 7-B represent creating a new level in the hierarchy to accommodate additional producers. These figures also show that the centralized and decentralized approaches have a better MRT than the hierarchical approach when event frequency is low and very small number of event producers exist in the system.

8 Conclusion and Future Work

This paper describes a novel active monitoring architecture (called HiFi) for distributed multimedia systems. The presented monitoring system organizes the monitoring agents in a hierarchical structure that distributes the monitoring load and limits the event propagation. The monitoring agents are programmable and can be reconfigured manually through users' interactions, or automatically by the agents based on the detected events. Users utilize a simple language interface, called filter, to define their monitoring demands and associated actions. Users can specify "general" monitoring tasks that can be customized dynamically by the monitoring agents in order to perform special monitoring tasks. We developed several techniques to support a programmable agents environment for active monitoring. This includes *event incarnation* to enable event-filter-action programming model, *filter incarnation* and *filter registers* to enable self-configurable monitoring operations, and *dynamic subscription* that enables users to add, delete or modify their requests at run-time. In this paper, we also demonstrate number of examples for using HiFi in monitoring and steering large-scale distributed multimedia systems.

The active monitoring architecture offers significant advantages in the scalability and performance of the monitoring systems. It also enables the consumer to control the monitoring granularity, and thereby minimizing its intrusiveness. The scalability testing results shows an improvement of 37% and 29% (on average) of HiFi mean response time over centralized and decentralized architecture, respectively, with the increase of event producers. The primary source for these improvements is attributable to the programmable agents environment that permits distributing of monitoring load among the LMA and DMA groups and increasing concurrency of monitoring tasks. The HiFi active architecture also enables the localization of event filtering and classification in the area from which the events originate, which is necessary for reducing monitoring intrusiveness. HiFi's dynamic signaling inherent with the active monitoring, and event batching techniques significantly minimize the application perturbation to about 50%.

Although HiFi was developed and used to monitor existing applications such as RMS, it remains a prototype monitoring system. Many open issues remain to be addressed by our

research plan. These include: the integration of fault tolerance to make HiFi resilient to network and application failures, improved filter incarnation mechanisms to provide higher abstraction such that users can specify the ultimate monitoring target without having to specify intermediate monitoring tasks, and extended monitoring language and architecture that addresses temporal events.

References

[1] Ehab Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. HiFi: A New Monitoring Architecture for Distributed System Management. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS'99)*, pages 171–178, Austin, TX, May 1990.

[2] Ehab Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. Application-Layer Group Communication Server for Extending Reliable Multicast Protocols Services. In *IEEE Int. Conference on Network Protocols*, pages 267–274, Atlanta, GA, October 1997.

[3] Ehab Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. Dynamic Monitoring Approach for Multi-point Multimedia Systems. *International Journal of Networking and Information Systems*, pages 75–88, June 1999.

[4] Ehab Al-Shaer, Mohamed Fayad, Hussein Abdel-Wahab, and Kurt Maly. Adaptive Object-Oriented Filtering Framework for Event Management Applications. *To Appear in ACM Computing Surveys.*

[5] 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.

[6] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry Peterson, and Prasenjit Sarkar. PathFinder: A Pattern-Based Packet Classifier. In *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*, pages 24–42. USENIX Association, November 1994.

[7] M.C. Chan, G. Pacifici, and R. Stadler. Managing Multimedia Network Services. *Journal of Network and Systems Management (JNSM)*, 5(3), September 1997.

[8] S. Floyd, V. Jacobson, S. McCanne, C-G. Liu, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. pages 342–356, October 1995.

[9] Stella Gatziu and Klaus R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the* 4^{th} *International Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, February 1994.

[10] German Goldszmidt, Shaula Yemini, and Yachiam Yemini. Network Management by Delegation - the MAD approach. In *Proceedings of the 1991 CAS Conference*, pages 347–359, 1991.

[11] Object Management Group. *The Common Object Request Broker: Event Service Specification.* Tech. Rep. CCITT X.734/ISO 10164-5, 1993.

[12] W. Gu, G. Eisenhauer, E. Kraemer, K Schwan J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. In *Proceedings of FRONTIERS'95*, pages 11–19, February 1995.

[13] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–597, June 1994.

[14] Raj Jain. *The Arts of Computer Systems Performance Analysis.* Addison-Wesley, Reading, Massachusetts, 1991.

[15] J. Joyce, G. Lomow, K. Slind, and Unger B. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–50, 1987.

[16] S. Kaetker and K. Geihs. A Generic Model for Fault Isolation in Integrated Management System. *Journal of Network and Systems Management (JNSM)*, 5(2), June 1997.

[17] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman. Tools for distributed Application Management. *IEEE Computer*, 24(8):42–51, August 1991.

[18] Charles E. McDowell and David D. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[19] J.V.D. Merwe, S. Rooney, I. Leslie, and S. Crosby. The tempest: A practical framework for network programmability. *IEEE Network Magazine*, June 1998.

[20] D. Olge, K. Schwan, and R. Snodgrass. Application-Dependent Dynamic Monitoring of Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):593–622, December 1989.

[21] Guru Parulkar, Douglas C. Schmidt, Eileen Kraemer, Jon Turner, and Anshul Kantawala. An architecture for monitoring, visualization, and control and gigabit networks,. *IEEE Network*, 11(5):32–38, October 1997.

[22] Beth Schroeder. On-line Monitoring: A Tutorial. *IEEE Computer*, 28:72–78, June 1995.

[23] Morris Sloman, editor. *Network and Distributed System Management*. Addison-Wesley, Reading, Massachusetts, 1994.

[24] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, , and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, Jan 1997.

[25] Ouri Wolfson, Soumitra Sengupta, and Yechiam Yemini. Managing Communication Networks by Monitoring Databases. pages 944–953, September 1991.

Ehab Al-Shaer is an assistant professor of computer science and the director of the Multimedia Networking Research Lab in School of Computer Science, Telecommunications and Information Systems at DePaul University. He received an MSc in computer science from Northeastern University, Boston, Massachusetts and a Ph.D. in computer science from Old Dominion University in Norfolk, Virginia, in 1994 and 1998, respectively. He worked as a senior networking engineer for DataGeneral and Tellabs Corporations from 1990 to 1993. During his research career, he was awarded 14 professional certificates in networking technology. Al-Shaer also received a NASA fellowship in 1997. His research interests include network and distributed systems management, reliable multicast protocols, active networking and multimedia communications.

<Event> ::= **EVENT =** <Event_Body> .
<Event_Body> ::= <Prim_Event> | <Comp_Event>
<Prim_Event> ::= **{**<Fix_Att> **;** <Var_Att>**}** <Event_Name>
<Comp_Event> ::= (<Prim_Event> <Event_Op> <Comp_Event>) |
(<Prim_Event> <Event_Op> <Prim_Event>)
<Fix_Att> ::= **ModuleName** = <String> ,
FuncName = <String> , <Report_Mode>
<Report_Mode> ::= **Immediate** | **Delayed**
<Var_Att> ::= <Predicate> , <Var_Att> | <Predicate>
<Predicate> ::= <Att_Name> <Relation> <Value>
<Event_Op> ::= $\wedge$ | $\vee$ | $\sim$
<Relation> ::= < | > | = | $\neq$ | $\leq$ | $\geq$
<Value> ::= <Number> | <String>
<Event_Name> ::= <Att_Name> ::= <String>

Table I. High-level Event Specification Language (HESL)

```
<Filter> ::= FILTER = <Filter_Body>
<Filter_Body> ::= [<Event_Expr>]; [<Filter_Expr>]; [<Actions>];
                  <Filter_Name> .
<Event_Expr> ::= ( <Event_Name> <Event_Op> <Event_Expr> )
                 | <Event_Name>
<Filter_Expr> ::= ( <Predicate> <Filter_Op> <Filter_Expr> )
                  | <Predicate> | TRUE
<Predicate> ::= ( <Pred_Att> <Relation> <Pred_Att> )
                | ( <Pred_Att> <Relation> <Value> )
<Pred_Att> ::= <Event_Name>.<Att_Name>
<Filter_Op> ::= <Event_Op>
<Actions> ::= <Action> ; <Actions> | <Action>
<Filter_Name> ::= <Program_Name> ::= <String>
```

Table II. High-level Filter Specification Language (HFSL)

<Action> ::= <Exec> | <Event_Name> | <Filter_Rinc> | <Filter_Registers> | **FORWARD**
<Exec> ::= <Dir_Name> / <Exec> | <Program_Name>
<Filter_Register> ::= <Identifier> = <Event_Name>.<Att_Name>
<Filter_Registers> ::= <Filter_Register> | <Filter_Register>;<Filter_Registers>;
<Filter_Reinc> ::= **ADD** <Filter_Name>; <Filter_Reinc> | **ADD** <Filter_Name> |
DEL <Filter_Name>; <Filter_Reinc> | **DEL** <Filter_Name> |
MOD <New Filter>; <Filter_Reinc> | **MOD** <New Filter>
<New Filter> ::= <Filter_Name>.**EX** = <Event_Expr> | <Filter_Name>.**FX**= <Filter_Expr> |
<Filter_Name>.**EX** = <Event_Expr>; <Filter_Name>.**FX**= <Filter_Expr>

Table III. High-level Action Specification Language (HASL)

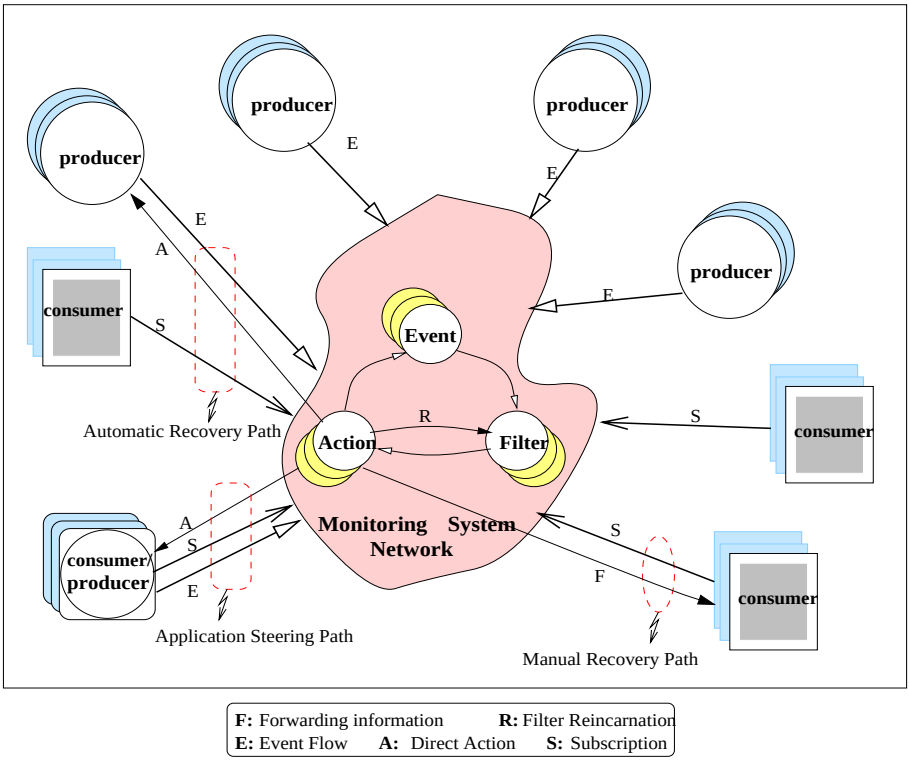


Figure 1: Monitoring Model.

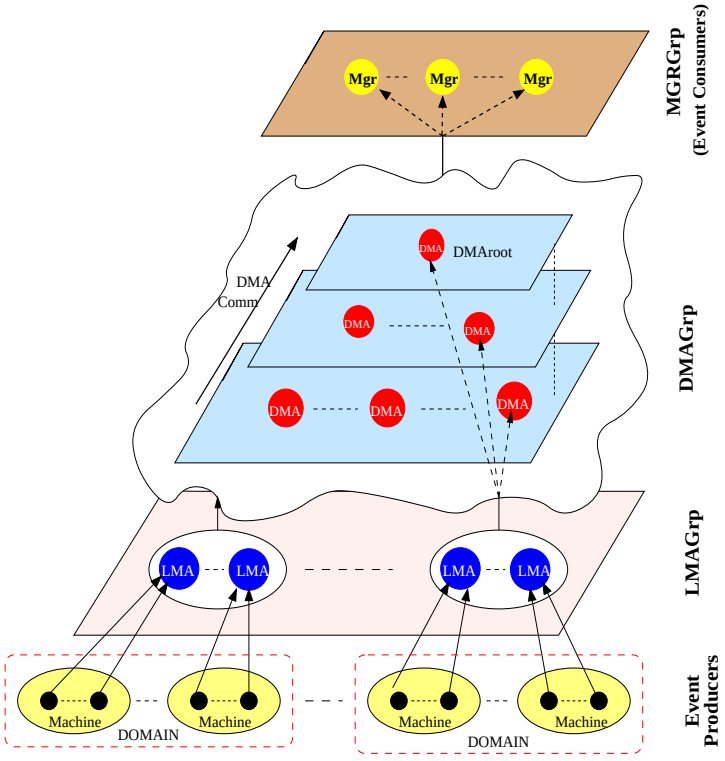


Figure 2: Hierarchical Filtering-based Monitoring Architecture.

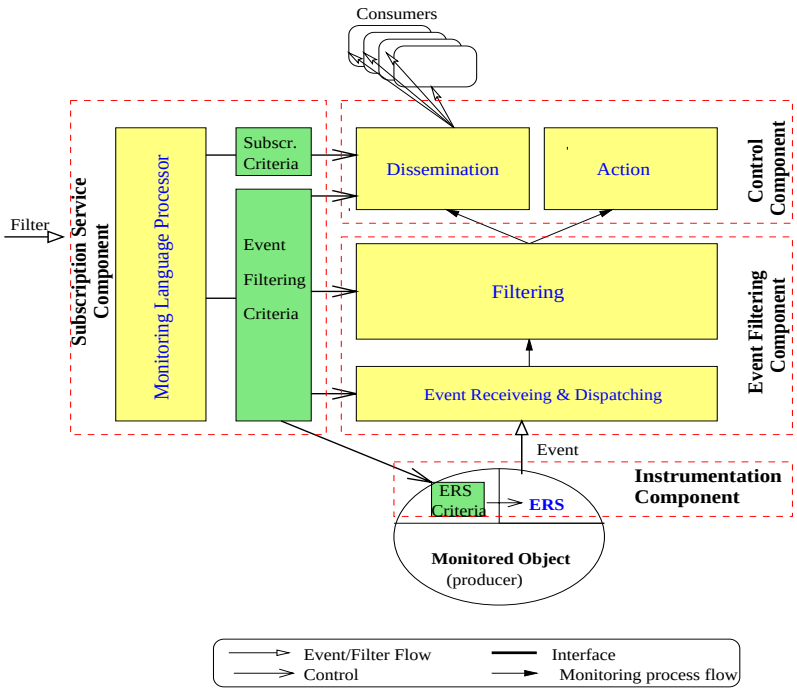


Figure 3: Monitoring System Components.

EVENT= { `ModuleName`=RMS,`FuncName`=McastSend,`Immediate`;
`Machine`="ANY", `Domain`="ANY", `GrpName`="ANY", `TransRate`=ANY, threshold= ANY } `McastSend.`
EVENT= { `ModuleName`=RMS,`FuncName`=McastRecv,`Immediate`;
`Machine`="ANY", `Domain`="ANY", `GrpName`="ANY", KBrec= ANY, NackSch=ANY } `McastRec.`
EVENT= { `ModuleName`=DMA,`FuncName`=ANY,`Immediate`;
`Machine`="ANY", `Domain`="ANY", KBrec= ANY, NackSch=ANY } `DomAVG.`

FILTER= $[McastSend]$;
$[McastSend.TransRate \ < \ STHRESHOLD]$;
$[ThisGrp = GrpName; MOD \ \ Slow\_Memebrs.FX = (McastRec.GrpName = ThisGrp)$;
$ADD \ \ Update\_Threhold]]$;`MonMcastSender`.

FILTER= $[(McastSend \wedge McastRec)]$;
$[(McastRec.\_ctr{=}\_LMAs \wedge McastRec.GrpName = \text{“}*'') \wedge McastRec.NackSch > McastSend.threshold)]$;
$[CalcAvg; DomAVG; FORWARD]$; `Slow_Memebrs`.

FILTER= $[DomAVG]$;
$[DomAVG.\_ctr = \_DMAs]$;
$[UpdateThrehold; \ McastSend]$; `Update_Threhold`.

Figure 4: Slow Members Discovery in Reliable Multicasting.

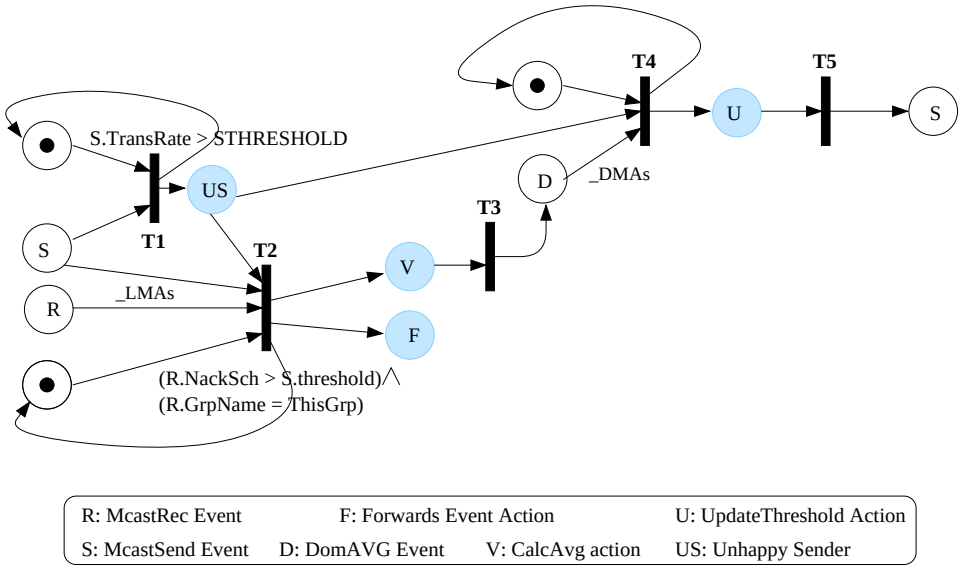


Figure 5: The PN Representation of Slow Members Discovery Filters.

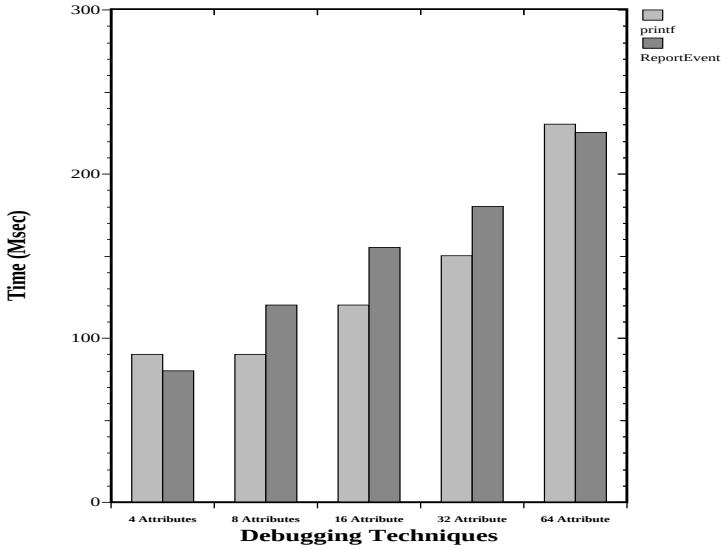
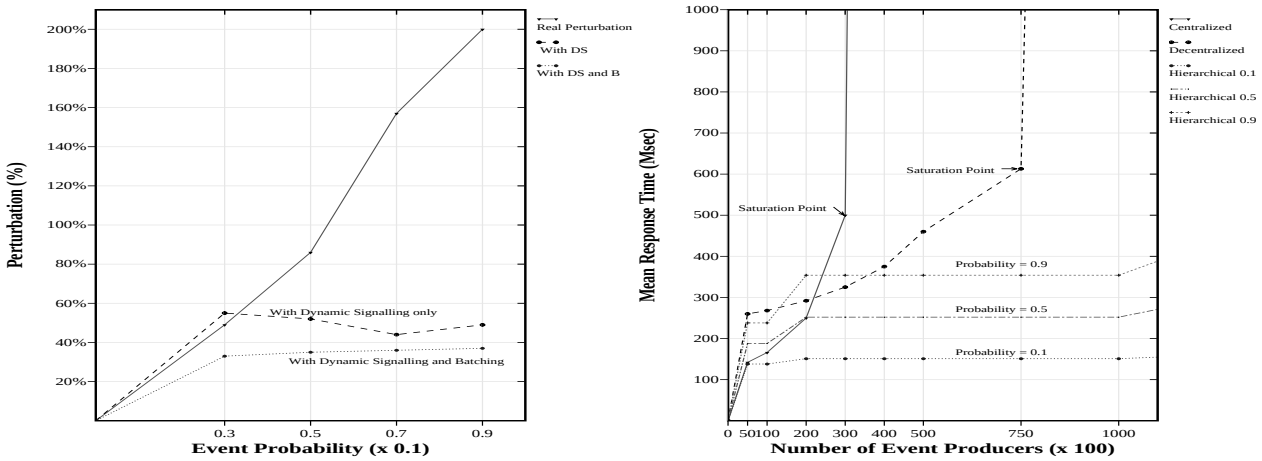


Figure 6: ERS ReportEvent Perturbation



A. B.

Figure 7: A) Application Perturbation Analysis. B) Scalability with Number of Event Producers.