A Survey of Event Filtering Mechanisms for Dynamic Multi-Point Applications

Ehab S. Al-Shaer
Department of Computer Science
Washington University
St. Louis, MO, USA

Douglas C. Schmidt
Department of Computer Science
Washington University
St. Louis, MO, USA

Abstract

High-performance event filtering is an essential service in a distributed processing environment. We are developing an object-oriented event filtering framework to efficiently process the large volume of event traffic generated by dynamic multi-point (DMP) applications (such as automated fault management in telecommunication systems). Our framework, which is based on CORBA, supports the automated generation and optimization of event filters in a distributed system.

Our work represents a major contribution by (1) devising an integrated model of event filtering that spans several application domains and (2) improving the functionality, performance, scalability and usability of event filtering.

This paper describes the primary characteristics and challenges of developing high-performance event filtering for dynamic multi-point applications. We survey existing event filtering mechanisms and explain key characteristics for each technique. In addition, we discuss limitations with existing event filtering mechanisms. and outline how our framework will improve key aspects of event filtering.

1 Introduction

The demand for dynamic multi-point (DMP) applications is increasing. DMP applications deliver messages from one or more producers to one or more consumers. Examples of DMP applications include decision support in information systems, wide-area information servers (such as the WWW or archie [27]), distributed systems monitoring, performance and fault management in large-scale network management systems, real-time market data analysis systems, and on-line news clipping services.

DMP applications exhibit two primary features:

- They are *multi-point* – consumers subscribe to subsets of events that are generated in the system. Consumers may have different subscription demands and any event that matches a subscription is delivered to the corresponding set of consumers;
- They are *dynamic* – since each event may potentially be delivered to a different subset of consumers. Consumers are capable of updating their subscriptions at run-time in response to changes in their environment.

Figure 1: General Structure of a Dynamic Multi-point Application

In contrast, conventional multi-point applications (such as teleconferencing) exhibit less dynamism. For instance, once a consumer has joined a multi-point group in an ATM switch, all data sent by a supplier is received by all other consumers that are members of the group. Figure 1 shows the general structure of a DMP application.

In an enterprise-wide DMP application (such as an automated fault management system for global satellite-based personal communications system), a large volume of events can be generated by suppliers. However, consumers may need to be notified about only a small portion of the events in the system. Therefore, *event filtering* mechanisms may be required to reduce the volume of notifications and efficiently deliver events to the appropriate consumers.

Event filtering serves as an efficient mechanism for monitoring and detecting events and delivering them to the interested consumers. It also serves as a data reduction mechanism that eliminates unnecessary network traffic and unnecessary processing by consumers. Not all consumers in DMP applications are interested in every event. Therefore, aggregate system performance can be enhanced significantly through the use of event filtering.

This paper presents an integrated model of event filtering that encompasses the features of a wide range of filtering mechanisms and satisfies the requirements of DMP applications, as well. We identify the key criteria that are useful for evaluating event filtering design alternatives.

Event filtering spans several application domains. However, existing techniques for event filtering may be insufficient for DMP applications due to four primary limitations: *functionality, performance, scalability and usability*. This paper motivates and describes the object-oriented framework and optimization techniques we are developing to improve the functionality, performance, scalability and usability of event filtering for DMP applications.

This paper is organized as follows: Section 2 gives an overview of event filtering and describes the primary characteristics and requirements of dynamic multi-point applications; Section 3 classifies existing event filtering mechanisms according to several criteria; Section 4 surveys existing event filtering mechanisms; Section 5 illustrates limitations with existing event filtering mechanisms; and Section 6 presents concluding remarks.

2 High-Performance Event Filtering for Dynamic Multi-point Applications

As background for our discussion of event filtering for DMP applications, this section introduces basic terminology and explains the impact of DMP application requirements on the design of event filtering.

2.1 Terminology

An *event* is a significant occurrence in a system that is reported by a *notification message*. The notification message may contain information that captures event characteristics such as event type, event values, event generation time, event source, and event state changes.

For simplicity, we use "event" and "notification" interchangeably in this paper. That is, we consider a notification to represent an event. An event is called a *primitive event* if it is based on a single notification message in the system. For example, an event representing all notifications with destination address "foo" and application name "ftp" is a primitive event since detecting this event only requires checking the fields of a single notification.

An event that depends on more than one notification is called a *composite event*. For example, assume a filter is defined to detect an event represented by notifications from source address "foo" with subject "overloaded" and all notifications of destination address "foo." This represents a composite event since detecting this event requires the recognition of multiple primitive events. We use the term *event pattern* to refer to the definition of a primitive or a composite event.

An *event filter* is a set of *predicates*. Each predicate is defined as a boolean-valued expression that returns *true* or *false*. Predicates may be joined by *operators* (such as `AND`, `OR` and `NOT`) that enable the composition of arbitrarily complex filter expressions. Thus a *filter* or *event expression* is a set of predicates joined by operators. Filters can be joined together to form an optimized filter by a process called *filter composition*.

2.2 An Example DMP Application

This section describes an example DMP application to illustrate the requirements that will be discussed in Section 2.3. The example also illustrates how event filtering helps to satisfy the requirements of DMP applications. This example is taken from the domain of network management (other examples are discussed in Section 3.1).

In an enterprise-wide network management environment, the management entities (*i.e.,* managers and agents) are distributed throughout the network. Using DMP terminology, managers are event *consumers* that have different demands on agents. For example, managers remotely monitor agents and gather network statistics related to performance and reliability. Managed objects in the network (such as routers, bridges, switches, and workstations) are equipped with agents that serve as event *suppliers*. Agents can either generate notifications that report the status of managed objects periodically (traps) or generate notifications according to explicit requests from managers (polling).

In an enterprise-wide network, the managers may be unable to handle a *high-volume* of notification messages generated by suppliers. Filtering agents can be used to alleviate these bottlenecks. These agents continually receive subscription requests from managers and classify events based on filters installed by managers. Managers can add, delete or modify their subscription requests at run-time. To reduce network traffic, one or more network managers can be notified about an event through dynamic multicast operations supported by the filtering agent. Filtering agents may reside in hosts other than suppliers and consumers.

Filtering agents can detect primitive events by efficiently classifying notifications they receive. For example, a filtering agent can detect a notification of an inoperable network link using the following filter expression:

Link_Down: (`msg_type` = error `and` `error_code` = link_down)

Another example, the filtering agent can detect if the utilization of a link is 90 percent or above:

Utilization: (`msg_type` = stat `and` `util_stat` $\geq$ 90)

In addition, managers can subscribe to composite events that consist of other events. For example, a network manager may want to be notified if a *Link* is down and the utilization of any other links is 90 percent or more:

Link_Down `and` Utilization.

2.3 Requirements of Dynamic Multi-point Applications

The primary goal of our work is to design and develop an event filtering framework that supports DMP application requirements. Therefore, we first describe the requirements of DMP applications and explain how these requirements affect the design of an event filtering framework:

• **A high volume of event messages are generated continuously in real-time by one or more suppliers:** For example, in an enterprise network management system, a large number of events originating from suppliers (such as network managed objects like routers, bridges and hosts) are sent to manages for immediate response. This shows the need for *high-performance* event filtering to classify and evaluate the large volume of events. Moreover, event filtering mechanism

Table 1: DMP Applications Requirements

must be *scalable* to handle large numbers of suppliers and consumers.

• **The message formats of events are potentially complex:** message formats contain both header fields (*e.g.,* timestamps, source/destination addresses, routing ids, priority levels) and data payloads (*e.g.,* telemetry measurand values). This implies the need to provide high-level programming tools that allow developers to define filters in a DMP application without being concerned with the low-level details of event formats.

• **Zero or more consumers subscribe to a subset of the total events generated by the supplier(s):** Unlike traditional static multi-point applications (such as teleconferencing), each generated event may be received by a different subset of consumers. Therefore, an event filtering system should support a *dynamic multicast* operation where members in the multicast groups can change dynamically at run-time.

• **Consumers may add, delete, or modify their subscriptions dynamically:** The event filtering system must provide a high level of *dynamism*. This is important to ensure adequate response to selectively monitor, detect, and recover when resource failures occur during system operation. Moreover, event filtering should be highly re-configurable to respond to dynamic consumer requests (add/delete/modify) with minimal performance overhead.

• **Event consumers typically reside on different hosts than the suppliers:** The suppliers and the consumers are connected via local or wide area networks. In some cases, consumers and suppliers may be separated by high-latency communication links that possess high delay and jitter. Reducing the volume of events is important to eliminate unnecessary network traffic and minimize the use of network bandwidth. In many cases, therefore, event filtering should not take place in the consumer even if the consumer is powerful enough to handle the filtering because this consumes excessive network bandwidth.

• **Consumers may need to detect both composite events and primitive events:** Consumers may need to know if certain events occur together under certain conditions. For this reason, the event filtering system not only classify events, but also tracks the event history of the system to detect interesting combinations of events. This feature is required by many DMP applications. Moreover, many DMP applications require an integrated event filtering model that classifies both primitive and composite events. In a filtering agent environment, for example, a high-volume of notification messages from sensors or agents are generated. The event filtering agent tracks the event history flow and classifies individual events. When a specified event is detected by the event filtering agent, it forwards the notification and/or performs the action corresponding to the detected event.

• **Consumers may need to receive events according to priority order and/or time-constraints associated with events:** This implies the need for real-time processing of events. For example, in an automated event monitoring of a large-scale health delivery system, notification messages received from a critical nodes (such as intensive-care equipment) are given high service priority. Some DMP applications (such as wide-area information location services) can tolerate "best-effort" delivery while others (such as real-time fault management) require reliable "real-time" delivery to satisfy the hard time-constraints of these applications. A real-time resource allocation and priority scheduling scheme may be necessary to support DMP applications with stringent performance requirements.

3 A Taxonomy of Event Filtering Criteria

Event filtering is useful in several domains including distributed systems toolkits, network and distributed system management, communication protocols, and active databases. Classifying event filtering systems based on their key design criteria helps identify and evaluate alternatives for designing event filtering mechanisms. In this section, we identify key event filtering criteria and outline different alternatives for each one. Section 3.1 explains the application of event filtering in four domains, emphasizing the domain-specific goals of each one; Section 3.2 describes different internal representations for event filtering; Section 3.3 discusses the programming interface used to define an event filter; and Section 3.4 presents alternative approaches for modeling and defining events.

3.1 Application Domain

Event filtering is used as a classification mechanism in several application domains. Each domain uses filtering for different purposes according to domain-specific goals and requirements. In this section, we list several application domains that use event filtering, explain the purpose of using filtering as a classification mechanism in each domain, and present example applications. Table 2 summarizes the discussion in this section.

Distributed Systems Toolkits: Event filters are used in distributed systems for the following purposes:

• **Validating incoming messages:** Filters are used to distinguish (classify) invalid messages. A message arriving at a consumer is examined by passing it through a series of validation filters. For example, the Isis system provides a facility that protects itself and its clients against errors from peers [2] (*e.g.,* unauthenticated clients, truncated messages

Table 2: Application Domains for Event Filtering

from faulty clients). Isis uses filters as a protection facility to validate authenticated messages in applications such as the Isis distributed news service.

• **Extracting portions of the message:** In distributed systems that support parallel programming (such as Paralex [2]), filters are used to associate data with processors in data-parallel computing. Data may be sent to multiple destinations for parallel processing. Each node uses a filter to extract different subsets of the data to process in parallel [2].

• **Agent-based distributed applications:** Agent-based distributed applications [19, 7] require efficient managing and coordination of a high volume flow of events. In these applications, event filtering agents are responsible for filtering incoming events. These events are received continuously as notification messages from active sensors or other remote agents. The remote agents may poll devices frequently and generate reports on the health and status of the system. In this environment, the event filtering agents classify individual messages, as well as detect composite events of interest to consumers. Since these systems are distributed by nature, no agent has complete knowledge of the system. Therefore, event filters can be used to detect composite events consisting of events sent from one or more agents or sensors. Some examples of distributed agent applications include the following:

- Event management in large-scale organizations – In large organizations (such as hospitals, airports, and automated factories) event filtering agents can be used to improve and automate decision management capabilities by monitoring, classifying, and coordinating events that flow through the system. If the event filtering agent detects a specified event (such as failure, emergency, or paging), it immediately performs the action correspond to this event (such as forwarding a new event to another agent or consumer).
- *Mission critical applications* – Mission critical military applications involving missile guidance systems require a form of filtering. An automatic missile guidance system is typically equipped with an intelligent filtering agent that continually receives notifications from the radar system (or the agent of the radar system). It fires at the target automatically when the filter conditions are satisfied.
- *Control systems* – Robots systems in a nuclear reactor may be provided with filtering agents that receive events from agents distributed throughout the site. The filtering agents determine the action to be taken if a specified error condition is detected.

Network and System Management: Managing networks and distributed systems is hard since a large number of events occur simultaneously. Event filtering is used to monitor and manage the flow of events in networks and distributed systems. Applications of event filters in network and system management environments include the following:

• **Monitoring and analysis:** Using filters, a network administrator can classify and analyze certain types of events and collect statistical information about different aspects of network operation [3]. In distributed systems, event filters are used to manage distributed applications by monitoring the events either at the communication level or at the application level [24]. Events and information collected through filters may be analyzed further using databases or graphical tools for debugging and troubleshooting [29].

• **Fault management:** Event filtering is a basic component in *fault management systems* [14, 25]. In these systems, network elements/objects (such as hosts or routers) send alarms (also called "traps") to indicate changes in system status. Event filters are used to classify different types of alarm events and forward them to management applications based on prior subscriptions.

• **Security management:** Event filters may be installed at different points in the network to capture events that reveal fraudulent behavior and generate notifications accordingly. The filters classify events based on a particular pattern of values in the packet headers, network traffic, or the content of data in messages.

Examples of events filtering toolkits in the network and system management domain are the Packet Monitoring Program (PMP) [3] and HP OpenView [13].

Communication Protocols: Event filters have been used to support efficient packet demultiplexing. An event filter examines incoming packets and forwards them to one or more end points based on values of fields in the packet [16, 18, 28]. Some applications of packet classification in communication protocols include:

• **Packet demultiplexing:** In operating systems, packet filters are used as an efficient technique to classify packets and determine the path the packet must follow within an end-system to route the packet to its communication endpoint [1].

• **User-level protocol implementation:** In micro-kernel operating systems different protocol stacks may coexist together in user-space. Using event filters in the kernel, packets are demultiplexed and forwarded to the proper user-level protocol stack.

The CSPF [18], BPF [16], MPF [28] and PathFinder [1] are examples of event filtering mechanisms in the communication protocol domain.

Active Databases: Active databases are constructed via *triggers* in a database environment. Triggers are specified as *event-condition-action* tuples. When an *event* occurs and a *condition* is satisfied, the corresponding trigger fires and the *action* is executed. In active databases, events may be classified (or detected) by event filters based on pre-defined definitions. The following examples show the use of event filters in active databases [6]:

• **Financial applications:** Trades can be executed in response to an observed pattern of trading events in a stock market.

• **Fraud detection:** A particular sequence of credit card purchases may point to fraudulent use.

• **Production management and quality assurance:** A particular sequence of defects indicates problems that must be brought to attention of a supervisor.

[6] and [5] present examples in this domain.

3.2 Event Filter Internal Representation

The internal representation of a filter is a key issue in studying and evaluating event filtering mechanisms. The internal representation determines the *structure* and the *operation* of the filter. A filter consists of the *algorithm* and the *data structure* used to detect and classify events. The internal representation of a filter has a major impact on the following filtering characteristics:

- *Performance* – The algorithm and data structure used by the internal representation determines the number of comparisons needed to process an event. Hence, filtering efficiency can be improved if the internal representation of the filter reduces the number of comparisons required to match an event.
- *Scalability* – The internal representation may provide an efficient filter composition technique that increases the scalability of the filtering mechanisms by minimizing the overhead of adding/deleting/changing filters in the system.
- *Functionality* – The internal representation of the filter determines the scope of the functionality that should be performed by the filter. For instance, some filter representations [1, 16, 18, 28] only classify primitive events since they do not keep track of the event history in the system. Conversely, other filter representations [5, 6] are capable of detecting composite events that are based on extensive history.

In the following, we classify event filtering mechanisms based on the functionality of filter internal representation. For each classification, we present key alternative internal representation models (data structures and algorithms) and optimization techniques used for each filtering mechanism. To focus the discussion, we show a filter example and how it is constructed using each representation. This example captures all packets with an IP source address "foo" and either the IP destination address is "bar" or the TCP destination port is "ftp."

3.2.1 Primitive Events Classifiers

The internal representation of this type of filter focuses on supporting an efficient classification of primitive events. In particular, it does not record any history of events detected in the system. This makes it impractical for use classifying composite events. Conventional packet filters [1, 16, 18, 28] are examples of this type of filter. Packet filters primarily classifies and demultiplexed communication protocol packets to user process endpoints. The remainder of this section presents alternative internal representations of filters for primitive events:

Boolean Expression Tree Representation: A boolean expression tree representation is a binary tree. Each interior node in the tree represents a boolean operation. The leaves represent test predicates (also called *masks* or *cells*) on event fields. Each edge in the tree connects the operator (parent node) with its operand (child node).

The algorithm for manipulating a boolean expression tree is based on a bottom-up parse of the tree. Events are classified by evaluating the tree starting at the leaves (test predicates) and propagating the results up to the binary operator at the root. The event is matched if the root of the tree evaluates to "true." Figure **??** illustrates an example of this representation model.

A tree representation algorithm can be optimized using a *short circuit* expression evaluation technique. In this technique, the tree parsing process terminates with a value of "false" whenever a conjunctive predicate evaluates to false.

Directed Acyclic Graph (DAG) Representation: A DAG representation is implemented as an acyclic graph. Its nodes represent the test predicates and the edges represent the control transfer.

The DAG is parsed top-down such that if the test predicate (also called a *cell*) is true, the right-hand edge is traversed, otherwise the left-hand edge is traversed. Thus, the test predicate (either *true* or *false*) determines the edge to traverse. An event is matched if the terminating node (leaf node) is denoted as *true*. There are two terminal nodes in the graph, the *true* node that denotes the acceptance of the packet and the *false* node that denotes the packet rejection.

A DAG may be optimized [1] by re-arranging the cells so that the longest common prefix between events is the longest *prefix* matched in the DAG. When a new event pattern is added, the DAG is traversed to determine the longest prefix that matches the pattern. A new edge is then inserted to form the suffix for this event. Figure **??** shows an example of an optimized DAG representation.

The DAG representation model is also known as: a *Directed Acyclic Control Flow Graph* (DCFG) [16] and a *Field Parsing Tree (FPT)* [3], which have similar internal representations. Figure **??** shows an example of a DCFG.

3.2.2 Composite Events Classifiers

The internal representation of this type of filter is designed to detect and classify composite events, as well as primitive events. An example of this type of filter is the event filtering in active database systems. The main function of the internal representation of these filters is to track events detected in the system, classify composite events as they are recognized, and trigger actions based on events.

In DMP applications, both primitive and composite event classification functionality may be necessary in filtering systems. Primitive event classification is required since events in DMP applications are mostly represented as notification messages. In addition, composite event classification is required since the event filter may need to track event history in the system. Table 3 compares these two types of event classifiers.

The following examines alternative internal representations for composite event filters.

Deterministic Finite Automata (DFA) Representation: The DFA representation is a finite state machine graph. A transition between two states represents an event occurrence. Each state represents the history of the system environment either before or after the occurrence of an event. For example, if event x occurs a transition on x from one state (h_1) to the next state (h_2) occurs. In this case, h_1 and h_2 represent the environment *before* and *after* x occurs, respectively [6].

In this model, a filter consisting of a single primitive event is represented by a three state automaton consisting of a start state, accept state, and non-acceptance state. From all states, the transition on event x (the event to be detected) triggers a transition to the accept state. Otherwise, on all other events the transition is to the non-acceptance state.

A filter consisting of composite events is constructed by combining the DFAs of primitive events together into one DFA using joining rules of Finite Automaton. By definition, all DFA transitions are deterministic. An example of this representation model is given in Figure **??**.

Reachability and state minimization analysis are used as optimization techniques to eliminate unreachable states and to minimize the DFA, respectively [6].

Petri Nets (PN) Representation: A model of Petri nets called *Coloured Petri Nets* (CPN) has been used in active database systems [5] to define event filtering. The following describes this model:

All predicates are represented as states called *places*. CPN has a number of *tokens* assigned to places that are defined by a *marking*. A *place* is marked when a predicate of that place is matched. Operations between places (predicates) are represented by *guard functions* that are checked after places are marked.

Whenever a predicate match occurs, the input *place* of the CPN is marked with a token. Initially, *tokens* are stored in auxiliary places of the CPN to designate the marking at creation time. Now, if *all* input places of a state are marked, the event *variables* that are denoted as labels in the CPN arc are bound to the value of the appropriate token and the *guard function* is evaluated. If the *guard function* evaluates to true, the state *transition* is fired, the token is transferred to the output place of the transition, and the variable value is propagated to the next state. More details of Petri Net concepts and behavior are found in [5]. Figure 5 **??** illustrates a simple example of an event filter modeled by a CPN.

3.3 Event Filter Programming Interface

This section describes various ways to program event filters, which are then implemented using internal representations discussed in the previous section (Section 3.2). An *event filter programming interface* provides a language for defining filter components (such as filter expressions and actions). An event filter expression describes all *predicates* involved in the event definition (including the message fields and the operators) and the *actions* to take when the desired event is detected. The internal representation then uses this information to construct a corresponding filter definition.

The event filter programming interface is a crucial criteria for event filtering. It represents the access point that a developer or application can use to control the filtering engine (which contains the internal representations of an event filtering system). There are many design alternatives that determine the characteristics of the event filter programming interface. Two key characteristics of event filter programming include the following:

- *Expressiveness* – the expressive power of the event filtering definition depends on the filter operators provided by the filtering programming language. Providing a rich set of expression operators offers flexibility in constructing filtering expressions. Filter programming interfaces that lack this expressiveness limit the capability of the event filtering mechanism and make it more application-dependent since it allows to express filters only within the application domain functions.
- *Ease of use* – some event filter programming interfaces are declarative languages where the filter definition is given as pattern/event match. Thus, no need to specify the program control and the data structure operations as it is the case in imperative languages. This allows users to specify their interest without focusing on low-level programming details. In contrast, other programming interfaces are more like an imperative "assembly language." This requires users to deal with low-level details (such as message format and bit/byte operations).

In the following subsections, we discuss several design trade-offs involved in event filter programming interfaces and then classify existing event filter programming interfaces.

3.3.1 Design Trade-offs Between Event Filter Programming Interfaces

There are a number of design trade-offs that arise when implementing event filter programming interfaces. In the follow-

Table 3: Event filtering Internal Representation

ing, we will briefly discuss the trade-offs between alternative designs for event filter programming interfaces. More details and examples appear in Section 4.3.

Low-level vs. High-level: The advantage of programming filters using low-level interfaces (such as a filter "assembly language") is that they may perform better than the high-level interfaces. However, programming filters using high-level interfaces increases usability (since they are easier to program) and portability (since they are less dependent on the hardware and underlying virtual machine of the filtering engine).

Imperative vs. Declarative: The declarative approach makes filter programs more concise and easier to write compared with the imperative approach. Thus, the declarative approach increases the extensibility and maintainability of the filter programming. However,, the declarative approach imposes limitations on programmer expressiveness. This may decrease the power of event filter programming. For example, some declarative filtering interfaces (such as PathFinder [1]) are customized to work for specific applications (like demultiplexing TCP/IP packets). In contrast, the imperative approach helps to avoid this limitation since the user may write more expressive filter programs.

Basic Operators vs. Advanced Operators: Some filter programming interfaces provide basic operators (such as `AND`, `OR`, and `NOT`). Others may provide more advanced operators such as `Before`, `After`, and `Sequence`. Advanced operators increase the expressive power of event filtering expressions. The disadvantage of using advanced operators is that there may be addition overhead at run-time caused by translating and processing these operators. Since the basic operators usually represent core instructions in the filter expression, they may not incur additional run-time overhead.

Interpreters vs. Compilers: The choice between using interpreters versus compilers leads to several design trade-offs. In the following, we discuss these trade-offs in the context of event filtering design:

- An interpreter is normally used when filters are implemented in the the OS kernel (where compilation may not be feasible due to protection and robustness concerns).
- In some environments, a compiler is the favorable choice to keep the OS protected from any bug generated by ad-hoc programs.
- A compiler is more convenient when the filtering mechanism is implemented in user-level applications since dynamic linking and run-time optimization can be used. Compilation increases the efficiency of event filtering mechanisms.
- Interpretation increases execution overhead since the program code is continuously re-examined. This causes a significant degradation in event filtering performance.
- Interpreters may also increase space overhead compared with compilers. For example, the interpreter and all the supported routines must usually be kept available. In contrast, compilers use dynamic linking to link the target routine at run-time. This feature helps to minimize space utilization.

3.3.2 Classification of Event Filter Programming Interfaces

The filter definition can be programmed at different levels of abstraction. In the following we classify these abstractions. For each abstraction level, we discuss examples of filter programming interfaces used by existing filtering mechanisms. In addition, for each filter programming interface, we show how the filter examples presented in Section 3.2 is implemented. Table 4 summarizes the material discussed below.

Imperative Low-level Programming Interface: Low-level languages/interpreters (such as assembly or micro-code languages) have been used to program filters imperatively. This type of filter programming interface can be combined with other high-level applications (such as user-interface or high level description languages) to provide a high-level interface.[1] The following are examples of event filter mechanisms that use this level of abstraction to define filters:

- *Stack-based Interpreter* – A stack-based interpreter uses *push* and *pop* operations to construct a filter [18]. The *push* operation causes either a word from a received packet or a constant to be pushed onto the stack. The *pop* operation evaluates the top two words on the stack and pushes the result back on the stack.

  A filter boolean expression is a filter composed of logical operations. It is constructed by a sequence of *push* and *pop* stack operations. An example is given in Table 5.
- *Register-based Assembly Language* – Other packet filter mechanisms use register-based languages to define packet filter definitions [16, 28]. By using assembly language instructions (such as *load, store, branch/jump and compare* instructions), event or packet filter expressions are specified as a sequence of load, store and compare instructions. Some packet filter assembly languages support more powerful features. For example, MPF is an extended assembly language that supports filter composition and packet fragmentation [28]. Table 5 illustrates a short example written in a register-based assembly language.

Imperative High-level Programming Interface: A high-level description language (similar to high-level programming languages) has been used to define filters. This programming interface is imperative since the filter definition is

[1] For example, *tcpdump* Unix network utility uses compiler to translate the high level filter description into BPF assembly language programs [16].

Table 4: Event Filter Programming Interface Dimensions

Table 5: This example represents an event filter that captures all packets of IP source address= "foo" and either IP destination address= "bar" or the TCP destination port= "ftp"

given as a program that describes the semantics of the filter predicates. The following is an example of this type of filter programming interface:

- *Interpretive Pseudo-Machine (IPM)*– The interpretive pseudo-machine is an interpreter of the packet monitoring program (PMP) [3] that parses incoming messages and analyzes their field values. Message fields are processed by two objects: a *filtering object* and a *recording object*. Filtering objects cause message fields to be checked based on test predicates. Recording objects record the message fields to keep track of statistics. IPM is described further in Section 4. The filter example is shown in Table 5.

Declarative High-level Programming interface: Event filters may be specified in a high-level declarative language. The previous event filter programming interfaces use imperative languages where filter definitions are given as programs. Other event filtering interfaces use a declarative programming interface where filter definitions are given by a pattern/event match [1, 26]. The following are examples of this declarative filter programming interfaces:

- *Rule and Database Languages* – Active Databases have been used to define and implement event filters. In [26] an event filter is specified by a query-based language using SQL embedded with a rule-based language similar to *Prolog*. *Basic events* and *actions*[2] are specified by normal SQL queries:

  `SELECT` X = IP `FROM` Host_table
  `WHERE` Retransmission > 5

  This example defines the following filter: *if a host retransmission counter exceeds 5, then extract and display the IP address.*

  From an end-user perspective, *correlated events*[3] and *inferences*[4] may be described by a ruled-based language such as *Prolog*. Using this notation, the correlated event is placed on the left-hand side of the rule and the component event is placed on the right-hand side, *e.g.*:

  ```
  Overload(X) :- Users_Count(X,C),
  CPU_Response(X,T), C > 100, T > 0.5.
  ```

  These rules specify that the `Overload` event of machine *X* occurs if the number of users *C* is more than 100 and the average CPU response time exceeds 0.5 seconds. Another example of a declarative rule is:

  ```
  Terminate(X) :- Overload(X).
  ```

  This shows an inference constructor example where the action `Terminate` causes the machine *X* to stop accepting any more users when it gets overloaded (*i.e.,* the `Overload` event occurs). Table 5 shows a complete example of this user interface.

- *Declarative Scripting Language* –

  This special-purpose declarative language was developed to describe filters such as the PathFinder packet classifier [1]. In this scripting language, a filter is described by defining the packet headers and a pattern match [1]. Based on the declarative script, a packet classifier can identify all packets that match the pattern. More details on this language appear in Section 4.3. An example of this scripting language is shown in Table 5.

3.4 Models of Event Filtering

Defining the model of an event filtering mechanism determines the structure of event filter components such as predicates, event definition, and filter expression. This model determines the generality of the event filtering mechanism. A powerful filtering model is one that is flexible enough to express any desired filtering functionality. For example, modeling an *event filter expression* to handle composite events, as well as primitive events, helps to support a broad range of applications. Moreover, integrating the notion of time into the filter expression model enables the management of events according to when they occur.

Many existing event filtering mechanisms are application-dependent [1, 3, 18, 16, 28]) because they are modeled based on application domain requirements. This design reduces the generality and flexibility of the event filtering mechanisms. More discussion is presented in Section 5. The existing mechanisms of event filtering use different models. In this section, we present a taxonomy of alternative approaches to model event filters. Table 6 summarizes our discussion.

3.4.1 Event Definitions

Events are defined concisely in a filter program. An event can either be a *primitive* or a *composite* event constructed from one or more primitive events. Hence, modeling the event filter requires defining primitive and composite events definitions.

Primitive Event Definitions: The primitive event is a basic event that does not depend on the occurrence of other events. Composite events are composed from multiple primitive events. Different event filtering mechanisms use different definitions for primitive events based on application requirements. The following is a discussion of different

[2] what is intended to be done if an event occurs.
[3] called also *composite events* – compose more than one event.
[4] function that maps one or more events to a set of actions/test

Table 6: Models of Event Filtering

models of a primitive events that are used in existing filtering mechanisms:

- *Relational Operations*– The primitive event definition is set of test conditions (predicates) that are related to a single event occurrence. Every predicate compares the message field value with a matching value using a relational operator such as $=, <, \leq, >, \geq$. For example, an example of a primitive event definition using two predicates is:

  `message_id` = 51 or 17 $\leq$ `transaction_number` < 40

  The primitive event classifiers such as packet filters [18, 16, 28, 1, 3] use this model to define events in the filter definition.

- *Database Operations* – Database manipulation operations such as `retrieve, add, delete, or update` represent the primitive events in this filter model [5, 6]. Whenever one of these database operations is performed, a notification is generated and sent to the event filtering system [4]. This notification conveys event information to the filtering system, which then classifies the event and performs any associated actions. This model is normally used by composite event classifiers (such as filtering in active databases).

- *Database and Data-pattern Match Operations* – This model is similar to the previous model. The primitive events are represented by database operations executed in the system. In addition, primitive events are also represented as a data-pattern match in the database [26]. Examples of this model are presented in *rule and database languages* illustrated in Section 3.3.2.

Composite Event Definitions: As discussed in Section 3.2, existing mechanisms have two different considerations for composite events:

- *No composite events support*– Primitive events classifiers (such as packet filters [1, 3, 18, 16, 28]) typically do not support composite events. The primary focus in these techniques is to dispatch incoming events (packets) to endpoint processes rather to track the event history of the system.[5]

- *Supporting composite events*– In composite event classifiers, primitive events can be combined by special *logical* operators to construct composite events. There are two types of operators, *basic operators* and *advanced operators*. The next subsection presents more discussion of this issue.

[5]Although MPF [28] and the PathFinder [1] provide a mechanism to track the IP fragmentation number, this does not adequately support global tracking of event history in a system.

3.4.2 Filter Expression Definitions

The event filtering expression represents the *relation* between the event filter basic elements (predicates/events). In this section, we discuss different models of filtering expressions:

Filter Expression Operators: A filter expression can be either an expression of predicates or an expression of events. The primitive event classifiers use the former but the composite event classifiers use the later. In either case, special operators are required to join the predicates in a filter expression.

There are two classes of filter expression operators:

- *Basic Operators*– Simple logical operators such as `OR`, `AND` and `NOT` operators combine predicates or events to form a filtering expression. The primitive event classifiers [1, 3, 16, 18, 26, 28] support this class of operators to construct a filtering expression. However, some composite event classifiers use basic operators to join primitive events as well as construct advanced operators.

- *Advanced Operators*– These are constructed (derived) from basic operators. The advanced operators make composite event expression easier to define [5, 6]. Section 4 presents more discussion of advanced operators.

Defining the expression operators involves the same trade-offs shown above. Basic operators offer better performance and simplify the implementation. However, basic operators do not provide the same expressive power that the advanced operators provide. In particular, although the advanced operators can be transformed into basic operators, it is not trivial from the programmer perspective since the translation process is complicated. Thus, there are two trade-offs in this issue: (1) performance vs. expressiveness, and (2) implementation simplicity vs. programming simplicity.

Parameterized Filter Expression: Predicates in a filter expression consist of one or more parameters that are used to analyze and compare against a message (*e.g.,* `message_id` and `transaction_number` are predicates parameters in the previous event filter example). Event filtering mechanisms deal with parameters in different ways:

- *Constant Parameters* – Some event filtering mechanisms require filter parameter values to be determined during the filter programming stage. Typically, matching patterns, evaluation expressions, and extracted portions of messages are constants in the event filter program and remain constants during filter operation. For example, in [18, 16, 28] predicates parameters (*e.g.,* `packet type` = `ARP`) must be specifically specified as constant values (*e.g.,* string and integer) and cannot vary during filtering operation.

- *Variable Parameters* – In this type of event filtering scheme, parameters in the filter definition are modifiable. They are updated whenever specified in the filter definition language as shown in [1, 5, 6, 26]. The PathFinder packet filter [1] illustrates this feature. PathFinder uses a parameter called *IP fragmentation number* as a variable parameter to keep track of the IP fragments. The IP fragmentation number parameter value gets assigned/changed during filtering operation rather than during filter programming.

Parameterized event filters are discussed further in Section 4.

Time-Intervals in Event Filtering: Some applications require classifying events based on time-interval functions (such event creation time and event temporal ordering). Moreover, some applications require detection of temporal events. Thus, monitoring time-intervals may be needed to filter events. Defining time and interval functions may be supported in the event filters definitions. Event filtering mechanisms are classified according to support for time-intervals criteria as follows:

- *No time-interval support*– The event filtering mechanisms, (such as packet filters), classify events according to message fields only. Thus monitoring time functions and intervals are not supported in these mechanisms [1, 18, 16, 28].
- *Primitive Support*– The event time creation available in each event must be used explicitly to implement time functions. For example, to indicate that Event_A occurs before `Event_B`, we use the relational expression `Time(Event_A)` $<$ `Time(Event_B)`. Therefore, the absolute time and the temporal ordering between events can be specified in an event filter definition [26, 6].
- *Advanced Support*– A set of advanced time monitoring interval functions are defined [5]. These functions are used transparently without explicitly using the absolute time of an event.For example, to define the same previous example, we use `Event_A before Event_B`. Moreover, the time interval is defined by two points in time: a *start_time* and *end_time*. The time function can be defined *absolutely* as a specific point in time (*e.g.,* 11:31:00PM), or *relatively* to the event occurrence. For example,

  $1:15:00 +$ `Event_A`

  specifies 1 hour and 15 minutes after `Event_A` occurred.

  Monitoring intervals can also be represented as *periodic functions*. For example,

  `EVERY MONTH [11, 16:00-11, 22:00]`

  specifies to the interval from 4:00PM to 10:00PM of the 11^{th} day of every month.

  Time-intervals can also be manipulated by more advanced operators such as *overlap* or *extend* operators that perform an intersection or union between time intervals, respectively.

Figure 2: Decentralized Event Filtering

3.5 Performance Enhancements

Section 3.2 outlines the optimization and performance enhancement techniques possessed by current event filtering mechanisms. In this section, we describe these performance enhancement techniques in greater detail.

• **Kernel vs. user-level implementation:** Many packet filters are kernel-resident programs that can be controlled from user-level processes [18, 16, 28, 1]. Conversely, other mechanisms implement event filters as programs running in user space [6, 5]. In general, kernel-resident filters perform better than user-level filters. Kernel-resident filters avoid extra copying of messages across the kernel boundary since only interesting events are passed. Thus, packet filters implemented in the kernel are capable of achieving efficient demultiplexing that is beyond the capability of most user-space packet filters.

• **Efficient filter composition:** Different filter composition techniques have been investigated to achieve an efficient composition and detection of composite events. MPF [28] presented a filter composition algorithm for primitive events in the packet filtering environment. Deterministic Finite Automata [12, 6] and Petri Nets [5], (discussed in Section 3.2) are alternatives for composition of composite events.

• **Efficient Internal Representation:** As discussed in Section 3.2, the internal representation has a major impact on the performance of event filtering. For example, different arrangements of DAG ordering yield different performance results. In addition, in the next section, we show through comparing different internal representation techniques how performance depends on the internal representation adopted by the event filtering mechanism.

• **State Minimization:** Minimizing the search space yields a significant improvement in event filtering performance. Some event filtering internal representations use algorithms to delete any unreachable (useless) states [6] and/or redundant states (common states) [6, 28].

• **Enhanced Instructions:** Some event filtering mechanisms [16, 28] gain better performance by using low-level instructions such as assembly language. Moreover, some event filters use register-based instructions rather than stack-based instructions to reduce the number of memory operations. Section 4 discusses this issue in more detail.

• **Hardware Support:** The PathFinder [1] packet classifier is partially implemented in the hardware of the network adaptor. As a result, PathFinder gains a significant increase in performance compared with other packet filters.

• **Distribution Architecture:** Several types of event filtering distribution architectures are possible:

- Decentralized event filtering – In certain DMP environments, it is beneficial to decentralize event filtering by

Figure 3: Centralized Event Filtering

Figure 4: Distributed Event Filtering

performing it on consumer hosts (shown in Figure 2). This configuration is appropriate when the following conditions occur:

- the consumer hosts are powerful workstation platforms;
- a high-speed network is available to connect the suppliers to the consumer hosts;
- consumers subscribe to most events;
- event filters are relatively complex.

When these conditions occur it may become more efficient to perform filtering in the consumer end-systems.

- Centralized event filtering – In other DMP environments, it is beneficial to centralize the event filtering at a central event server (shown in Figure 3). This configuration is appropriate when the following conditions occur:

  - an event server is installed on a high-performance platform (such as a multi-processor);
  - the consumer hosts are run on less powerful platforms (such as inexpensive PCs);
  - a relatively low-bandwidth (or highly congested) network connects the event server to the consumer hosts;
  - consumers subscribe to a relatively limited subset of events;
  - the complexity and number of event filters subscribed to by consumers does not produce a major processing bottleneck at the event server.

  When these conditions occur, the network and the consumer hosts at the edges of the network are typically the processing bottleneck, rather than the event server. Therefore, a centralized event filtering architecture helps to off-load work from the network and the consumer hosts.

- Distributed event filtering – More complex event filtering scenarios are also possible (shown in Figure 4). For example, the network topology that interconnects suppliers, event servers, and consumers may span multiple routers and switches, across local-area and wide-area networks.

4 A Survey of Event Filtering Mechanisms

Work on event filtering spans a number of domains, including distributed system toolkits [11], network and system management [25, 17], user-level communication protocols [18, 16, 28, 1], and active databases [6]. This section outlines related work on event filtering and evaluates this related work in terms of its support for DMP applications.

4.1 Distributed System Toolkits

Isis [2] supports event filtering as part of its *Reliable Distributed Objects* (RDO) News service [11]. In Isis RDO News, all consumers in a process group receive all events sent by suppliers and filtering is performed at the destinations. This mechanism presents a distributed event filtering that protects the DMP applications from erroneous clients. However, two basic limitations are considered in this technique: (1) consumer filtering is limited to matching on character strings "keywords", (2) due to its decentralized architecture, the filtering in RDO News may not scale to accommodate a large numbers of DMP consumers that possess complex filtering requirements, and (3) it may wast the network bandwidth since the filtering is performed in the destination node.

4.2 Network and System Management

HP OpenView: [13] HP OpenView provides an implementation of the ISO OSI event report management services [25]. HP OpenView filtering supports the registration of Event Forwarding Discriminators (EFDs) on remote agents in a network. An EFD contains a filter expression based on event type, event value, event generation time, and event frequency. EFD filter expressions are described using GDMO, which is a schema for defining managed objects in a network. ISO OSI does not dictate the implementation of EFDs. However, the implementation of HP OpenView event filtering suffers from a highly inefficient process architecture that requires 4 context switches and 3 interprocess communication exchanges to send and filter each event[22].

Packer Monitoring Program: The *Packet Monitoring Program* (PMP) is a packet monitoring tool that uses event filtering for gathering statistics of packets in the network and analyzing traffic patterns. The packet parsing mechanism in PMP parses the packet according to the Field Parsing Tree (FPT) which is equivalent to the DAG. Any message field that has to be extracted for statistical analysis must be specified in FPT as nodes. For efficiency purposes, the packet headers format are hard-coded in the PMP code and known at compile time. On the other hand, PMP obtain flexibility by providing a dynamic configuration for the recording and statistics rules since they can be specified at run-time. PMP uses interpretive pseudo-machine (IPM) presented in Section 3.3. In this section, we describe the structure of IPM by illustrating some examples. IPM contains two objects:

- *Filtering Objects* (`FObj`) – which causes the message fields to be checked based on test predicates.

- *Recording Objects* (`RObj`) – which records the message fields for statistical purposes.

Each message field (`Fa`)is processed by one or more `RObj` or `FObj` after calling the `Invoke` operation. `RObj` causes the *message field* `Fa` to be recorded for statistical purposes. However, in case of a filtering object, `FObj` is applied over the `Fa` message field and the result of either *True* or *False* is returned. Hence, the `Invoke` instruction takes two parameters, the message field (`Fa`) and the object (`FObj` or `RObj`) as follows:

`Invoke (Fa, FObj)`

The `Fa` message field is to be filtered by the `FObj` filter; or

`Invoke (Fa, RObj)`

The `Fa` message field is to be recorded by `RObj` filter.

`Invoke` specifies two types of instructions:

- *unconditional instructions*– define the invocation of `RObj` over `Fa`, and
- *conditional instructions*– constructed by an `if` statement and an invocation of `FObj` over `Fa`:

```
if (Invoke (Fa, FObj))
        <true-part>
else
<false-part>
```

PMP uses some optimization techniques for efficient filtering composition. If there is a redundant filtering object (i.e. two filters have the same test condition), the Boolean result of the invocation of the first filter is saved and reused when the second filter is invoked. This technique allow evaluating the filtering object without unnecessary recomputation. However, this requires adding two extra fields in the filtering components: the filter result and the indirect pointer[**?**].

The filtering technique used in PMP suffices the following limitations: (1) it does not support detection of composite events, (2) in addition to the space overhead required by the filtering composition technique, it also costs a performance overhead because of the extra lookup needed to get the result, and (3) the expression operators are primitive (AND) and do not accommodate more complex applications.

4.3 Communication Protocols

Several studies have reported measurements based upon various types of *packet filters* (also known as *packet classifiers* [1]). Packet filters were developed originally to support efficient demultiplexing for user-level implementations of network protocols [18]. In addition, they have been used to enable unobtrusive traffic monitoring on promiscuous-mode networks [17]. In the following section, we present the evolution of packet filtering mechanisms and an overview of each technique.

CSPF and BPF : The CMU/Stanford Packet Filter (CSPF) [18] and the Berkeley Packet Filter (BPF) [16] were two influential first generation packet filter implementations. CSPF is a stack-based packet filter that uses binary operation (*pop* and *push*). The *stack-based interpreter* causes either a word from a received packet or a constant to be pushed on the stack. The *binary operation* (such as `EQ, GT`) pops and evaluates the top two words from the stack and pushes the result back on the stack. The CSPF filter engine uses boolean expressions with a tree graph model for filtering. The stack-based interpreter and tree model limit the performance of CSPF. In contrast, BPF achieves better performance. It uses a register-based assembly language (load and store instructions) and an acyclic control flow graph (CFG) instead of stack-based language and tree graph, respectively.

For example, in CSPF each logical operation requires five stack operations (three *pushes* and two *pops*) to be executed. This makes it perform poorly compared to the register-based interpreter that uses one simple compare operation (*i.e.,* jeq, jgt). In addition, using CFG instead of a tree graph model is another reason of the performance difference between BPF and CSPF. A tree model often does unnecessary or redundant computation [16]. For example, in Figure (1-a), `ip.src` and `Ether.Type` are checked twice to evaluate the entire tree. Moreover, BPF handles additional features that are not supported by CSPF (such variable header-length and extracting portion of a packet).

Both CSPF and BPF maintain a list of configured packet filters. The list may reorganized to move frequently accessed filters to the front of the list. This approach works well if there are relatively few packet filters, or if only a small number of consumers are active simultaneously. When hundreds of filters (or hundreds of consumers) exist, however, this approach does not scale well since the time required to filter packets grows linearly with the number of filters.

MPF : Recent research on packet filters/classifiers addresses scalability limitation by enabling the composition of multiple filters. [28] describes the composition technique used in the Mach Packet Filter (MPF). MPF is designed to support user-level implementations of layered protocol stacks (such as TCP/IP). Often, packets destined for separate TCP connections on an endsystem contain a common prefix, followed by variation at a single point in the packet. For example, active TCP connections on an endsystem will have many IP header and TCP header fields in common (such as source and destination IP addresses). In this case, the TCP destination port number is the only demultiplexing field that varies among packets. Thus, a hash table can be used to efficiently demultiplex packets to different endpoints.

MPF uses a register-based assembly language with some added instructions to deal with this issue. MPF has three major advantages over the previous packet filters: (1) efficient demultiplexing of incoming messages by combining similar filters together, (2) dispatching fragmented packets and out-of-order fragments, and (3) enabling *dynamic binding* between filters and processes such that filters can be re-assigned to another process dynamically. This dynamic binding is applicable in MPF by changing the hash table entry to handel new *IP_ports* [28].

MPF has the some limitations:

- MPF[28] provides a mechanism to track a limited amount of event history to support IP fragmentation. However, this is a rather restricted mechanim since it does not support global tracking of event history in the system.
- The optimization technique described in [28] possesses low setup/remove latency. However, it does not generalize to more general complex composition of filters.
- The hash table processing overhead due to retrieving and updating entries per fragment arrival degrades the performance of MPF.

PathFinder : The PathFinder tool described in [1] presents a more general technique for coalescing filters with common prefixes. PathFinder is a packet classifier that combines software and hardware to optimize the composition of complex filtering patterns. The software portion of PathFinder builds a directed-acyclic graph (DAG) interpreter (similar to CFG in BPF) based upon patterns specified by a user-defined declarative syntax. The PathFinder interpreter matches fields of incoming packets using information stored in the DAG. PathFinder has several novel features that makes it perform better than earlier packet filters:

- *Caching key-pattern* – PathFinder adaptively reorders the DAG when a match occurs on a composite pattern. The reordering process synthesizes and caches a new "lightweight" representation (called a *key-pattern*). The new key pattern reduces the number of comparisons for subsequent packets matching the composite pattern.
- *Simple operation semantics of test and set operations* – PathFinder uses a *cell* to match (*test*) incoming packets or load (*set*) a value extracted from a packet. The cell consists of four parts (1) the *offset* where the matched part of the message starts, (2) the *length* of the matched part, (3) the *masking* pattern, and finally (4) the *value* to be compared with the extracted part of the masked part. Due to the simple semantics of cell operations (compared to other semantics based on assembly language) the *test* and *set* (cell) operations can be implemented more efficiently in PathFinder than in the previous packet filters. In other packet filters, a sequence of push and pop instructions (in the case of CSPF) and sequence of load/store and compare assembly instruction (in the case of BPF and MPF) are required to implement an equivalent single cell instruction.
- *PathFinder hardware support* – PathFinder off loads portions of the packet classification logic into hardware, which may run on a network adaptor. Despite losing the flexibility by having a packet filter in hardware, the existence of a packet filter in a network adaptor provides a significant increase in the performance of the packet filter operation.

The primary limitations with PathFinder are:

- The high latency required to setup/remove patterns from the DAG.
- The software implementation of the DAG uses an interpreter, rather than a compiler. This precludes a variety of performance optimizations.
- PathFinder provides a mechanism called "dual line" to deal with fragmented packets. This mechanism is considered to be more flexible than the previous ones. However, the "dual-line" facility in PathFinder is still restricted mechanism in tracking global event history (composite events) for the following reasons: (1) activation of the event represented by the "dual-line" depends on the activation of the event represented by the "primary-line". This imposes a real restriction on the process of tracking composite events since composite event should be detected independent of each other, and (2) the only possible operator that can be used between the "dual-line" and the "primary-line" is `AND` operator which implies more restriction in the composite events expression.
- Some optimizations in PathFinder lack generality, which further limits its functionality and efficiency in other environments. For example (1) the proposed composition optimization technique does not work when common suffixes or intermediate states exist. This case is expected in DMP event filtering environment where messages become complicated and have no format restriction (as it is the case in protocol packets), (2) values in a cell operation are restricted to be loaded only once (when first line is matched) during filter operation. Loading multiple values in the cells of the *dual line* is possible but it is restricted to be simultaneous and when matching of first line (*primary line*) occurs, and (3) the inability to support relational (such as LE, GT,..etc) and logical operations (such as complement). This limits the application of PathFinder in the other application domains such as DMP event services and traffic monitoring.

4.4 Active Databases

Support for *triggers* is an important distinction between active and standard databases. A trigger is an event-condition-action expression where an event can be either a primitive or composite event[6]. In an active database, event filters are used to detect a composite event. A number of approaches were presented for modeling and detecting composite events (triggers) in active databases [20]. However, in this section,

[6]Notice that. in this section, composite event, trigger and event filter have the same meaning.

we discuss the three approaches proposed in [5], [6] and [26]. We categorized these approaches based on the event composition techniques they used: *Finite Automat, Petri nets* and *Rule-based* technique. This section is divided into three subsections: Section 4.4.1 discusses general and common characteristics of event filtering (event composition) in active database systems, Section 4.4.2 describes briefly the specifics of each approach.

4.4.1 Event Filtering in Active Databases

Many different techniques have been presented to support event filtering in active databases. The common goals of this work are *modeling* and *detecting* composite events. Modeling of event composition is defined as combining primitive events using special operators called "event composition event operators". For example, [6] uses regular expression notation with pre-defined operators to model composite events. Different algorithms were developed for composite event detection such as finite automata, Petri nets and rule-base language (discussed in Section 3.2). The following is an outline of some common characteristics of these different approaches:

• **Primitive event:** It is the basic entity of composite events and is defined as a database manipulation operation, namely, a retrieve, add, delete or update the database.

• **Event Filter Operators:** These are also called event filter *constructors*. They are used to construct a composite event (event filter) by combining primitive events together in an expression. For example, *AND, OR, NOT, SEQUENCE*[7] are event operators. More examples in other operators can be found in [6, 5].

• **Time functions:** The three approaches supports time functions in event composition definition. The following are some examples:

- *Absolute time* – `[12:30PM]`,
- *Relative time* – `before (EventA, EventB)`,
- *Periodic function* – `EVERY MONTH[12, 17:00-13, 20:30]`

In addition, temporal-ordering[8] of events is also supported in composite event definition as presented in [26, 6, 5].

• **Parameterized Events:** Events can be associated with parameters (or attributes) such as network address, transaction id, error number and so on. Some parameters values are saved over different event occurrences; for example:

```
Overload(X) :- Users_Count(X,C),
CPU_Response(X,T), C > 100, T > 0.5.
```

The common parameter `X` indicates that `Users_Count` and `CPU_Response` must have the same machine name (`X` in this case).

• **Masks/Conditions:** Events[9] can be masked by conditional expressions. Event (or event filters) may only be fired if the conditional predicate evaluates to *true*. Otherwise the predicate will "mask" the occurrence of the corresponding event. The concept of conditional predicate appears under different names in the literature. It is called a *mask* in [6], *guard functions* in [5], and *conditional predicates* as in [26].

4.4.2 Event Filtering Categories in Active Databases

Event filtering in active databases are divided into three categories, based on its event composition techniques, as follows:

• **Finite Automata:** The COMPOSE system [6] uses regular expressions with special operators, called event composition operators, to define composite event expressions. COMPOSE has two types of event composition operators, basic and additional operators. There are four basic operators, namely, `OR, NOT, relative, relative+` operators. The additional operators are constructed from these basic operators to make composite events easier to define. Composite events can be implemented and detected using Deterministic Finite Automata (DFA) (explained in Section 3.2). The task of composite event detection is straightforward. First, the finite automata, (implementing the event expression) are fed as input the primitive events that makes up the event expression. This, Subsequently, this determines which composite event the primitive event contributes. Second, for each of the determined composite events, a check is made to see if all of its primitive events occurred already. If so, the automata associated with a composite event reaches an accepting state, and the composite event is matched.

There are many advantages of using DFA to represent composite events: (1) utilizing the expressive power of regular expressions and (2) the ability to use DFA optimization techniques for more efficient composite events implementation. COMPOSE notation is extended beyond regular expression and DFA to support parameters and masks. For optimization purposes, reachability and minimization analysis are used to eliminate unreachable states and to minimize the number of states, respectively.

• **Petri nets** : This approach uses a modified version of Petri nets called *SAMOS*[10] *Petri nets (S-PN)* to model and detect composite events. Event composition expression is constructed using basic operators called event composition constructors. There are six basic constructors:

- `OR, AND, NOT` – logical operations.
- `SEQUENCE` – requires events occurrence to be in order.
- `TIMES` – indicates the n-th occurrence of an event in a specified time interval.
- `''*''-constructor` – indicates the first occurrence only of an event in specified time interval.

[7] SEQUENCE indicates a composite event of sequence of events occur in a consecutive manner.

[8] events occur according to specified time-precedence.

[9] and *event filters* also as filters are composite events.

[10] SAMOS is the name of the prototype active database used in this approach.

Table 7: Comparing Event Filtering of Active Databases

More information about these operators and examples can be found in [5]. The S-PN uses a *step-by-step* (or incremental) procedure to detect a composite event. In this case, the order sequence of all primitive events forming a composite event is known by a composite event detector that checks all ordered sequences whenever a primitive event occurs. If an ordered sequence is matched, one forward step is made and the place is marked. Consequently, "step forward" continues until the last element of the appropriate sequence is marked. This implies that the corresponding composite event is detected [5]. Many advantages have been reported in [5] justifying the S-PN approach:

- The ability to model complex composite events that includes parameters and time-interval functions.
- Convenient for implementing composite event detectors due to the simplicity of the data structure that represents S-PN.
- In general, fewer states are needed, compared to the DFA approach, to represent composite events.

In S-PN, event occurrence is represented by just marking an event place with a token. Therefore, events reserve only one place in S-PN, regardless of the number of occurrences. In contrast, for each event in a DFA model, the number of states is as many as the number of event occurrences (*i.e.,* is every event occurrence associates with one state in the automata).

• **Rule-Based Languages:** In this approach, rule-based language (such as *Prolog*) may be used to define event composition. Events are combined using `AND, OR and NOT` operators only. Parameters and time functions are supported in event composition definition. The *temporal ordering* between events can also be specified in the composite events. Table 7 compares and contrasts the previously described approaches according to some criteria.[11].

For example, the rule

`Crash-at-5 :- Crash, 5:00 A.M. = 20`

specifies that the composite event the `Crash-at-5` will be triggered if `Crash` event occurs within 20 minutes (after or before) from 5:00 A.M.

5 Limitations with Existing Event Filtering Mechanisms

The existing packet/event filtering mechanisms are not suitable as filtering techniques in high-performance DMP environment due to many limitations in these mechanisms. To overcome these limitations, we are developing an object-oriented framework for event filtering based on OMG CORBA [8], CORBA is the emerging standard for open distributed object computing. In this section, we discuss the limitations of the existing mechanisms in four major criteria: *functionality, performance, scalability* and *user-interface*. In addition, we explain the enhancements that are being implemented in our framework to overcome the existing limitations.

[11] ND in the table denotes "Not Defined".

5.1 Functionality in Application Domain:

The packet filtering has been used in many areas, including network monitoring, event services and user-level communication protocols. In this section, we discuss the lack of the functionality of the described in section 4 mechanisms to support high-performance event filtering.

Extracting Message Subsets: This function is highly demanding function when filters deal with large messages and only small portion of this message/packet is interesting. In centralized event filter model, the features of extracting message subset saves the consumer the effort of parsing and extracting message fields in addition to a tremendous saving in the network bandwidth. In this case, event filters should be capable of extracting the desired part of the message/packet and pass it up to the requesting consumer. This function is commonly used in trace generation applications where only few bytes out of a packet are saved in a trace file. It also commonly used in high-performance event services where the consumer is a low-performance endsystem (*e.g.,* a PC) and is willing to report just a small portion of an event message. All of the above mechanisms, except BPF and MPF, do not support this functionality. BPF and MPF partially support it by extracting just one contiguous portion of the packet. This is not sufficient when more than one portion of the packet is desired (*e.g.,* consumers subscribing to the same events but having different views according to their interests). For example, a consumer may be interested in *type* and *address* portion of the message, while another consumer is interested in getting *type* and *time* portion of the same message. These portions may not be contiguous in the message format.

Adaptive Filtering Parameters: Most of the packet filters described above require packet filter parameters to be specified by a schema definition language before installation. Typically, matching patterns, evaluation expressions and portions to be extracted are constants in the packet filter language before execution. This imposes a restriction on the functionality and the flexibility of packet filter operations since these values may not be known before installation time or before a particular event occurs. For example, in monitoring applications, it is useful to monitor the network activity of any machine sending ARP requests frequently during the day. Another example of an event service application is a consumer interested in receiving all events from any device sending more than five failure notifications within twenty four hours. In this case, `src_ip` is not known until the *frequent ARP requests* event happens. A later one `dev_addr` is not known until "five" failures occur within 24 hours. Other examples can also be shown to present the importance of dynamic binding of filter parameter. Although PathFinder provides a similar functionality to deal with IP fragmentation, it does not provide a general implementation of this function due to the same limitations explained in section 4.3.

State-based Filters: In many event filter applications, filtering is based upon changes to the value of measurands over time (such as an aperture constraint exceeding a pre-defined threshold), as well as other frequency-based filtering criteria (such as a particular error condition occurring x times in a row). In contrast, packet filters have traditionally permitted the definition of expressions that operate primarily on "stateless" filtering criteria (such as matching the value of a field in a packet against a particular value). In general, the only support for "state-based" filtering has been *ad hoc*, focusing on support for filtering based on IP packet fragments [28, 1]. *Dynamic binding* of filter parameters function can be useful in implementing this criteria.

External Events: Packet filters described above do not consider any information out side the range of the packet fields. This limits the operation of the packet/event filter where filtering of events/packet may depend on external factors such as date, time, counter, duration, ..etc [26, 9]. Therefore, packet/event filters should be extended to accommodate external events.

Filter Composition: First-generation packet filters [18, 16] do not support filter composition. However, recent ones (such as MPF [28] and PathFinder citePeterson:94) do support it. MPF and PathFinder support filter decomposition by factoring out the common prefix of the combined filters. however, this approach lacks generality for two reasons (1) does not support other composition parameters such as complement (negation of a filter) and (2) more efficient composition techniques can be employed as explained in the next section 4.2.

Protection Domain: DMP applications usually run in userspace rather than in an OS kernel. Therefore, optimization techniques (such as compilation, explicit dynamic linking, and parallelism) are more feasible in our DMP environment. In contrast, existing packet filter implementations are based on interpreters, either stack-based [18], register-based [16, 28], or DAG-based [1]. Interpretation is used in lieu of compilation due to the kernel-resident environment of traditional packet filters. For security reasons, most operating systems do not provide convenient ways of compiling and dynamically linking filter code into the OS kernel.

Dynamism: Dynamic mapping between filters and destined processes in the consumer end is a required function for DMP applications. Consumers may add, delete, or update their subscriptions rapidly in response to changes in external conditions. For instance, the automated network fault management correlation engine may need to install new filters in response to alarms triggered by managed objects in a high-speed network. Previous mechanisms use a static mapping between filters in the kernel and processes in the user space.

Our preliminary experiments on prototypes of the framework indicate that the appropriate choice of filtering optimization is affected significantly by the degree of dynamism required by a DMP application. Applications that tolerate high installation latency (such as SV telemetry processing, whose filtering constraints are generally fixed at initialization-time) benefit from the compiler-based optimizations. Conversely, applications that require frequent interactive updates (such as fault management applications initiated by network administrators) benefit from the parallel-based or tri-based schemes due to their lower (re)configuration latency.

5.2 Performance Enhancements:

In this section, we proposed some optimization techniques to reduce the time required to match an event message with the subset of consumers that have subscribed to receive that message. The following techniques is designed to increase in the performance substantially over the existing packet filters:

Efficient Filter Composition: The composition approach presented by previous packet filters [MPF, PathFinder] is not sufficient.

Another technique we are exploring involves optimizing event filtering by composing multiple filter expressions together using a tri-based data structure. This tri-based approach is a generalization of the DAG-based techniques presented in [1]. Every node in the tri implements a particular type of branching mechanism. The branching mechanism selected at a node employs a lookup function (such as a hashing function, a binary search, or a series of inlined relational expression comparisons) that is tuned to the type and the range of data values. The advantages of using tries is that they reduce the number of operations required to process M messages of size L through N event filters from $O(M \times L \times N)$ to $O(M \times L)$. This represents a substantial performance improvement when N becomes large.

A related optimization involves the use of automated compiler parsing and code generation techniques. In this approach, a finite automaton is created based upon combining a set of context free grammars that describe the composite filter expressions. A novel technique known as "skip-ahead parsing" [12] is used to eliminate unnecessary access to data fields during the parsing operations. An advantage of this approach is that the generated code may be optimized using highly efficient compiler optimization technology. The drawback to using compilation tactics is the relatively high latency required to compile and install, modify, or delete an event filter. OS support for explicit dynamic linking [21] helps to reduce this overhead to some extent, but the latency remains higher than the techniques based on parallel processing or tries.

More optimal composition can be achieved by considering all common states (test or load states) in the combined filters (appears as a prefix, a suffix or as an intermediate state). Common states create unnecessary redundant work that should be avoided in the composition process (union, intersection, complement,..etc). This approach can be incorporated with the *skip-ahead parsing* or *Tri-DAG* composition techniques.

More research studies has been done to provide more general and more efficient composition techniques [12] [6, 5].

Parallelism: One optimization technique we are exploring involves the use of parallel processing to improve the event filtering performance [23]. The main advantage of this approach is the relative easy of dynamically configuring and reconfiguring event filters in response to changing consumer subscriptions.

None of the previous work in packet/event filtering had touched this issue. One of the big motivations behind using parallelism in event filtering is utilizing the power of multiprocessor platforms. Most of the previous work in packet filters were concentrating on optimizing the parsing models such as DAG or CFG vs Tree model employing a sequential operations. Our proposed model is to extend DAG graph so independent test/load states (cells) will be combined together on one big state (called *parallel-state*) that represents a parallel execution of the contained sub-states. These sub-states can be ordered vertically in the *parallel-state* and preceding and following connections to the rest of the graph will be through the *parallel-state*. The event/packet filter passes a parallel-state if all subs-tate are already matched. Therefore, the subs-tate that has maximum execution time determines the running time of its *parallel-state*.

The challenging issues in parallelism are (1) determining the independent states in the graph which could be generated automatically by a smart compiler or it could be the user responsibility to define the independent states that could be run concurrently when coding the filter definitions and (2) synchronization process that should take place when multiple states are running concurrently [Schmidt]. The disadvantage of this approach is that the current generation of parallel processing platforms exact a performance penalty for sharing data on the I/O bus.

Support for Real-Time Event Services: In real-time event services, one or more consumer may subscribe for number of events which have different levels of priority. A large number of event messages may exist that may impose some unbounded delay on high-priority event messages. For critical faults and recovery procedures, consumers in charge of fault management may require a certain events to be reported within a specific time constraints. This time limit (called refreshness period [Kaiser]) is negotiated during subscription time in case of centralized event filtering. In either centralized or decentralized event filtering, filtering technique should be extended to support real-time events. In existing packet/event filters, incoming packets/events are buffered to the end of the queues based in FIFO model regardless how long queue is. However, in real-time event filtering, queuing and task scheduling is done based on the priority reference of a certain event.

High-priority events are classified by special-purpose filters called *real-time* event filters which are automatically generated (or combined using the proposed optimal composition technique) at the suppliers and consumer end whenever a consumer subscribes for it. There are two level of filtering in the supplier end: (1) *in coming real-time events filters* to classify high-priority event in the supplier end (and before sent report generation starts) and (2) traditional *out going event filters* to classify events according to the consumers subscription. Similarly, in the consumers end, an *in coming real-time filters* is used to to take care of high-priority events in the consumer end. Consequently, the preference will be given to high-priority events in both ends[13].

6 Concluding Remarks

This paper motivates and outlines research being conducted at Washington University on optimizations for high-performance event filtering. Efficient event filtering helps to reduce the large volume of event traffic processed by dynamic multi-point (DMP) applications. Unlike most related work, our optimizations are targeted for DMP applications. Existing tools for event filtering (such as Isis, HP OpenView, and packet filters) do not adequately address the usability, extensibility, performance, and scalability requirements of DMP applications.

To improve usability and extensibility we are developing an object-oriented framework for event filtering based on CORBA [8]. The framework supports the automated generation and optimization of filter expressions based on the CORBA interface definition language (IDL). By selecting CORBA, we plan to leverage off emerging distributed object development tools [10] and open system protocol specification techniques [15].

To improve DMP application performance and scalability, we are exploring optimization techniques to reduce event filtering overhead. These techniques employ parallel processing, dynamic tri-based search structures, and compiler technology based on context-free grammars to reduce the time required to filter events received.

References

[1] 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*. USENIX Association, November 1994.

[2] Kenneth Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, 1994.

[12] This is left as future extension.

[13] discussion of kernel vs user-level in real-time support is left as a pending issue for future work.

[3] Robert T. Braden. A Pseudo-Machine for Packet Monitoring and Statistics. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, August 1988.

[4] S. Chakravarthy and D. Mishra. Snoop: An expressive event specifiation language for active databases. In *Data and Knowledge Enginnering*, volume 14, pages 1–26, Nvember 1994.

[5] 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*, February 1994.

[6] Narian Gehani, H. V. Jagadish, and Oded Shmueli. COMPOSE A System for Composite Event Specification and Detection. In *Book Cahpter in Advanced Database Concepts and Research Issues*, pages 454–469. Lecture Notes Computer Science, 1993.

[7] Michael R. Genesereth and Steven P. Ketchpel. Software Agents. In *Communication of ACM*, pages 48–54. ACM, July 1994.

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

[9] David Holden. Presictive Languages for Management. In *Integrated Network Management I*, pages 585–596. IFIP, 1989.

[10] C. Horn. *The Orbix Architecture*. Tech. Rep., September 1994.

[11] Isis Distributed Systems, Inc., Marlboro, MA. *Isis Users's Guide: Reliable Distributed Objects for C++*, April 1994.

[12] Mahesh Jayaram, Ron K. Cytron, Douglas C. Schmidt, and George Varghese. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Submitted to the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*. ACM, 1994.

[13] Keith S. Klemba. Openview's Architectural Models. In B. Meandzija and J. Westcott, editors, *Proceedings of the* 1^{st} *International Symposium on Integrated Network Management*, pages 565–572. IFIP, 1989.

[14] Lee LaBarre. Management By Exception: OSI Event Generation, Reporting and Logging. In *Integrated Network Management I*, pages 308–323. IFIP, 1991.

[15] D. Lea and J. Marlowe. PSL: Protocol and Pragmatics for Open Systems. In *Tech. Rep. 94-0369*, Septemeber 1994.

[16] Steve McCanne and Van Jacobson. The BSD Packet Filter. In *Winter USENIX*, pages 259–269. USENIX Association, January 1993.

[17] Jeffrey C. Mogul. Effecient Use of Worksattions For Passive Monitoring of Local Area Networks. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1990.

[18] Jeffrey C. Mogul, Richard F. Rashid, and Micheal J. Accetta. The Packet Filter: An Effecient Mechanism of User-Level Network Code. In *Proceedings of the* 11^{th} *Symposium on Operating Systems Principles*, pages 39–51. ACM, November 1987.

[19] Dan Murray. Developing Reactive Software Agents. In *AI Expert Magazine*, pages 27–30. ACM, March 1995.

[20] T. Risch. Monitoring Database Objects. In *Proceedings of VLDB*, Augest 1989.

[21] D. C. Schmidt and T. Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems. In *IEE/BCS Distributed Systems Enginnering Journal*, December 1994.

[22] D. C. Schmidt and T. Suda. Scalable High-Performance Event Filtering for Dynamic Multi-point Applications. In *Proceedings of the* 1^{st} *International Workshop on High-Performance Protocol Architecture*, December 1994.

[23] D. C. Schmidt and T. Suda. Measuring the Performance of Parallel Message-based Process Architecture. In *IEEE INFOCOM Conference*, April 1995.

[24] J. Scholten and J. Pothuma. Monitoring Multimedia Systems. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, pages 377–380. IEEE Computer Society, April 1992.

[25] I. O. Standarization. *Information Processing Systems - Open Systems Interconnection - Part 5: Event Report Management Function*. Tech. Rep. CCITT X.734/ISO 10164-5, 1993.

[26] Ouri Wolfson, Soumitra Sengupta, and Yechiam Yemini. Managing Communication Networks by Monitoring Databases. In *IEEE Transactions on Software Engineering*, pages 944–953, September 1991.

[27] Tak W. Yan and Hector Garcia-Molina. SIFT - A Tool for Wide-Area Information Dissemination. In *1995 USENIX Technical Conference*, pages 16–20. USENIX Association, January 1995.

[28] Masanobu Yuhara, Brain Bershad, Chirs Maeda, and J. Elliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Winter 1994 USENIX Conference*. USENIX Association, January 1994.

[29] John A. Zinky and Fredric M. White. Visualizing Packet Traces. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 293–304, Augest 1992.