

Design of a Dynamically Extensible System for Network Monitoring using Mobile Agents

Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, and Megan Carney
 {tripathi, tahmed, spathak, mcarney}@cs.umn.edu
 Department of Computer Science
 University of Minnesota, Minneapolis MN 55455

Abstract—We present here the design of a framework for building future generation network monitoring systems using mobile agents. It is designed to support dynamic configurability, extensibility, active monitoring, and secure operations. New event types and their detection procedures can be incrementally added in this system, and any desired event data aggregation policies can be defined using the publisher-subscriber model. The use of Prolog provides high level and easy-to-use abstractions for defining and detecting new compound events based on correlation of event data. Active monitoring in this system is supported by defining event dependencies. The use of Ajanta, a secure mobile agent programming platform, provides the necessary infrastructure for protecting the monitoring system. We present here a set of experiments that we conducted using this system to evaluate and demonstrate its capabilities.

I. INTRODUCTION

The focus of this paper is on future generations of network management system architectures utilizing mobile objects/agents to monitor a network for security violations and enforce system policies. Management of computing resources in an enterprise network is becoming an increasingly complex task because many diverse components are frequently added, upgraded, and replaced. System monitoring tends to be an important part of management goals. Monitoring includes a range of activities: check for system component failures, configuration errors, overload conditions, auditing of usage, and intrusion detection. The complexity of monitoring large organizational networks requires new approaches to building system monitoring protocols and functions.

Many large-scale computing environments, such as university campus networks, tend to be relatively open without any firewalls or even physical barriers in accessing computers in public-labs. An intruder can access a computer over the network quite easily, or physically reboot

This work was supported by National Science Foundation grants ANIR 9813703, EIA 9818338, and ITR 0082215.

Participation of Sumedh Pathak and Megan Carney was partly supported by REU (Research Experiences for Undergraduates) funds with NSF grant ANIR 9813703.

a machine. In such large, open environments it is desired that a system administrator can actively monitor all nodes for suspicious activities. Besides network management needs, monitoring services are also required by many of today's emerging applications. For example, a "smart environment" may require monitoring presence of certain users or devices in its space.

The general issues and challenges involved in monitoring large distributed systems have been discussed in the past by several researchers [1], [2]. We identify the following as the important requirements for emerging monitoring systems:

Dynamic Configuration: Dynamic structures are needed to support changes in policies for monitoring, collection, and processing of information. As new hardware and software components are added to a system, the system should support mechanisms to dynamically install monitoring capabilities for such new components and integrate them in the existing monitoring infrastructure.

Dynamic Extensibility: It should be possible to extend the functionalities of existing components to include new monitoring mechanisms and event detection procedures without interrupting the operations of the monitoring system. Moreover, the system should support definition of higher level events based on correlation of existing event types. The system should also support incorporation of new correlation functions across distributed event databases following the paradigm of *cooperative security managers* [3].

Active Monitoring: A monitoring system should be able to alter its detection policies in response to some critical events. For example, in case a critical event is detected, the monitoring system may start operating at an increased level of alertness and initiate monitoring of some additional events.

Support for Administrative Security Policies: A large system is typically divided into multiple administrative domains. A monitoring system should support implementation of desired security policies for event data dissemination across different organizational domains.

Protection of Monitoring Infrastructure: The monitoring system should protect itself from attackers and intruders. Only authorized users should be allowed to install monitoring components, modify existing ones, or send event notification messages. The communication between event publishers and subscribers should be protected. For system integrity considerations, it should be possible to divide a monitoring system into different protection domains, because in a large system there always exists the possibility of a security breach [4].

Performance and Scalability: Often it is impractical to collect and process information at a central site when monitoring a large network. Centralized systems can potentially increase the latency in detecting critical events and impose significant processing load. Therefore, a monitoring system architecture should support any desired organization for decentralized and hierarchical collection and processing of information.

Most of today's monitoring systems rely on SNMP to collect data from various components [2], [5]. The SNMP model supports low level device management; it does not support abstractions for network-wide monitoring policies. The SNMP agents provide a limited and fixed set of functions, requiring tedious procedures to define new policies to monitor. In our approach, SNMP can be integrated as one of the building-blocks for low level device monitoring. Another common approach used in today's monitoring systems is to periodically execute scripts to monitor the status of a component or to process event data. Script based detection procedures tend to be cumbersome to install, debug, and to modify remotely. In contrast to an object-based approach, scripts offer a lower level of abstraction for remote manipulation. This also has the disadvantage of coarse-grain protection, as scripts execute under complete privileges of a specific user.

In this paper we present the design of a framework for active monitoring of network systems using Java-based mobile autonomous objects/agents. A mobile agent represents an object capable of migrating in a network to perform certain designated tasks at one or more nodes [6], [7], [8]. We demonstrate in this paper that mobile agent based network management is a natural fit to meet many of the requirements stated above. Network management and monitoring is a promising area for agents, where they can be used to perform remote information filtering, data correlation, and control functions [9], [10], [11]. Agents can be modified remotely to change their monitoring and aggregation policies and functions, and if needed, new agents could be installed at a node to perform functions different from the existing ones. Following are the main contributions of our work:

- This system demonstrates the utility of the mobile code and agent technology to dynamically install new monitoring agents at a node. Moreover, the functionalities of a monitoring agent can be extended remotely. New event types can be added dynamically with appropriate detection/handling procedures.
- High level abstractions using a logic programming language are used for expressing goals of the monitoring policies. The system supports definition and detection of new, high level compound events, expressed as logic rules in Prolog. To support these abstractions, event data is maintained in our system in logic databases.
- Mechanisms to support *active monitoring* are provided using trigger rules that initiate execution of new detection procedures and correlation functions when certain events occur.
- The dynamic installation of new agents at a node and the modifications of an existing agent's functionalities can be controlled according to any desired system-level security policies. Similarly, the functions for deleting existing agents or their detection functionalities are protected.
- Event dissemination based on the publisher-subscriber model can use secure and authenticated communication. Each agent can implement its own security policies for adding or removing subscribers for the events that it publishes. The publisher-subscriber model allows any desired communication relationships to be dynamically and securely established between agents.

This system has been implemented using Ajanta, a Java-based mobile agent programming platform [12], [13]. It uses the mobile agent execution model and the security architecture of Ajanta as the enabling technologies to realize the capabilities mentioned above. It also integrates Ajanta agents with a Java-based Prolog system [14] to maintain event databases and execute logic programming based correlation functions to detect compound events.

In the next section we present the overview of the system architecture of our agent-based monitoring framework. Section 3 presents the design of the monitor and subscriber agents in this framework. In Section 4 we discuss how the security mechanisms of Ajanta are used to ensure the protection of the monitoring system. In Section 5 we present a set of experiments that we conducted using this system to evaluate and demonstrate its extensible and active monitoring capabilities and highlight the benefits of agent based monitoring. The related work is discussed in Section 6 followed by the presentation of our conclusions.

II. OVERVIEW OF THE SYSTEM ARCHITECTURE

Our agent-based network monitoring framework, is designed to support a dynamically extensible environment

for monitoring network systems. This framework allows one to implement any desired policies for decentralized filtering, collection, and correlation of event data. These policies can be changed dynamically. It also provides mechanisms for defining and detecting new events in the system, and installing appropriate monitoring agents at different nodes in the system.

A. Event Definition and Class Hierarchy

A *basic event* represents some significant change in the state of a resource to be monitored. Higher level *combined events* are derived from other events by applying certain inference rules.

We need a canonical definition and representation of events, independent of any operating system specific details. For example, the concept of events such as *login-failure*, *remote-connection-request*, or *disk-system-full* are present in most operating systems; however, the specific mechanisms in each operating system to detect and report such events tend to be quite different. In Unix, such events are recorded by the system in some log files. We keep the events from different nodes in a standard format.

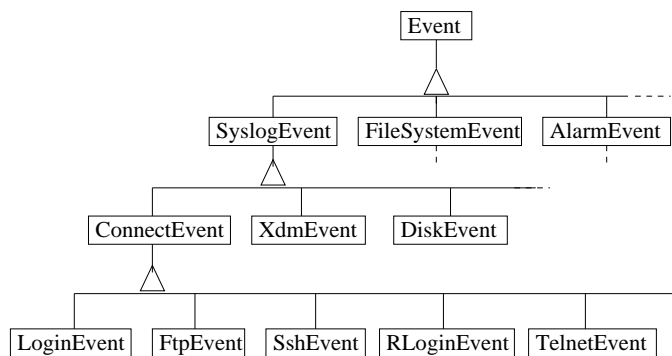


Fig. 1. Event Class Hierarchy (Partial)

Part of the event class hierarchy implemented in our system is presented in Figure 1. The *SyslogEvent* class represents the events generated from the system log files. The strings in the log files are either written by the system services or by other logger facilities. For example, in UNIX, services like SSH, FTP, NFS generate report status like “ssh-authentication success”, “ftp connect”, “disk crash” in the system log files. However, for X Windows activities, the *logger* command is used for logging “user’s console login” events. In Figure 1, *ConnectEvent* class represents all kinds of connections to the host. Its subclasses are events corresponding to FTP, SSH, Telnet, Rlogin etc. *XdmEvent* class is for X Windows activities, and *DiskEvent* class is for any logged disk activities. *FileSystemEvent* class represents the events which check file system consistency by various means, like scripts. The *AlarmEvent* class repre-

sents combined-events which are generated by predefined alarm conditions.

The base *Event* class has the attributes *id*, *name*, *source-location*, and *time of occurrence*. Additionally, subclasses may have more attributes as needed. A class like *SshEvent* represents all the ssh events, and a *name* field in the class distinguishes among “ssh authentication success”, “ssh connect”, “ssh fail” etc. Hence, the *name* field indicates the event type, and the class name indicates the broad category of system function in whose context the event is generated.

B. System Components and Configuration

Figure 3 shows a typical system organization in our framework. The system administration functions are performed through a set of secure nodes, termed *system management stations*. Our framework is designed to allow system administrators to define and create agents that would visit nodes in the network to perform the desired monitoring and filtering tasks.

Each node in the monitored network environment executes a facility, an *agent server*, which allows migration and installation of new agents for event monitoring, data collection, and correlation. Event communication between agents is based on the publisher-subscriber model. A *monitor agent* resides on each monitored hosts and checks system resources to generate desired events. A *subscriber agent* is extended from monitor agent and can subscribe events from other agents to correlate and create combined events.

Associated with each event is the definition of a *detector* object and a *handler* object. The detector object contains the event detection procedure, which is executed by a thread encapsulated within the object. The detection procedure determines how an event is detected and the conditions under which detection is triggered. For basic events, the detection rules can be of many different kinds, and they may vary from one OS platform to another. Therefore, in our framework an agent can probe its environment and then execute the appropriate methods. For example, it may involve continuous monitoring of some system state or log files, checking the utilization of certain system level resources, verifying the checksums of some files, or monitoring the use of system resources such as certain network ports.

The handler object performs filtering and notification of the events to the subscribers, and it can also store the event in a local database. Each such handler has a list of subscriber agents. To monitor a compound event, one may need to collect and correlate data from different nodes. A subscriber agent can also add (delete) itself to a monitor

Monitor Interface:

```

addEvent (EventDetector detector, EventHandler handler, Vector subscriberList, URN invokerID)
addEventDetector (String eventClassName, EventDetector detector, URN invokerID)
addEventHandler (String eventClassName, EventHandler handler, URN invokerID)
deleteEvent (String eventClassName, URN invokerID)
replaceEventDetector (String eventClassName, EventDetector detector, URN invokerID)
replaceEventHandler (String eventClassName, EventHandler handler, URN invokerID)
addSubscriber (String eventClassName, Vector subscriberList, URN invokerID)
deleteSubscriber (String eventClassName, Vector subscriberList, URN invokerID)

```

Subscriber Interface:

```

public void report(Event event, URN invokerID)

```

Fig. 2. Monitor and Subscriber Interfaces

agent's subscription list, or suspend (resume) its subscription. These functions can also be executed by a system administration agent on its remote monitor agents, and it can also add a new event detection rule and event notification format. The system administrator agent can also install new agents at a node to perform additional event filtering and correlation functions.

In this system, an agent can perform functions in three main categories: event monitoring, event subscription, and correlation of event data. Every agent implements the *RemoteControl Interface* defined by Ajanta, for its owner to remotely terminate or recall it. Additionally, an agent implements the *Monitor Interface* and may also implement the *Subscriber Interface*. The *Monitor Interface* defines methods that allow another authorized agent to add, delete, suspend, or resume a subscriber for an event. It can also add/remove an event, along with its detector and handler objects, in the set of events monitored by the agent.

The *Subscriber Interface* defines the remote method that is invoked by a monitor agent to deliver notification messages to a subscriber. The *Monitor Interface* and the *Subscriber Interface* are shown in Figure 2.

III. DESIGN OF MONITOR AND SUBSCRIBER AGENTS

In our monitoring system, agents are multi-threaded active objects. The architecture of a monitor agent is shown in Figure 4, which is extended from Ajanta's base *Agent* class and implements the *RemoteControl Interface* and the *Monitor Interface*. The only difference between a monitor and a subscriber agent is that a subscriber agent implements the *Subscriber Interface*.

A monitor agent maintains event detectors, handlers, and subscribers corresponding to the events registered through the *Monitor Interface* in a table, termed *monitor table*. A newly defined event together with its detector and handler objects is dynamically loaded into our monitoring system at runtime using Java reflection and class loading

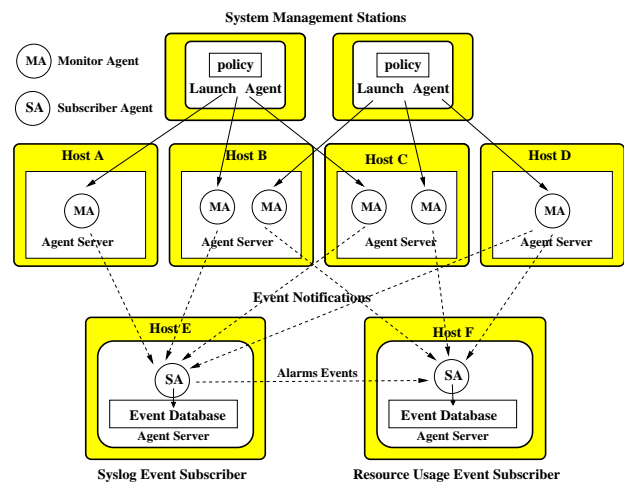


Fig. 3. Architecture for an Agent-Based Infrastructure for Network Monitoring

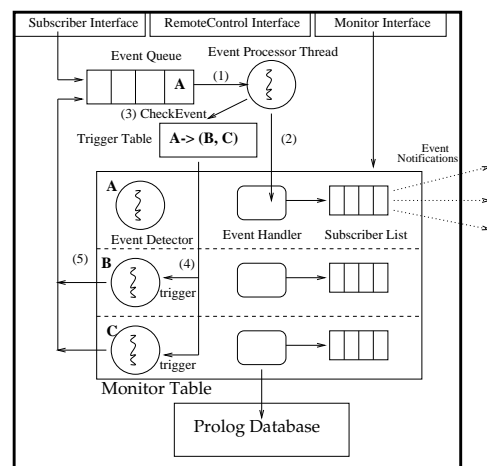


Fig. 4. Architecture for a monitor/subscriber agent

facilities.

A. Event Detectors

Each detector contains a thread, which executes the detection procedure asynchronously with other activities in the agent. All the events detected and created by the event detectors are deposited in an *event queue* to be processed by the *Event Processing Thread* of the agent. For a subscriber agent, an event received from a remote node is also put in this event queue.

A special class of event *TimerEvent* is generated periodically by *TimerEventDetector* objects. Various types of timer events, like heartbeat timer, file consistency check timer, watchdog timer etc, with different timeout periods can be registered in the monitor table.

B. Event Processing Thread and Event Handlers

The event processing thread waits for an event to arrive in the event queue. When an event is delivered to the event processing thread, it first executes the event specific handler, and then through the trigger table initiates asynchronous execution of the dependent detectors.

All event handlers are derived from the base *EventHandler* class, which provides the default functionality of sending event notifications to all currently registered subscribers. The handlers in the subscriber agents in our system also record the events in the agent's Prolog database. This model facilitates any hierarchical organization for disseminating events. A subscriber receiving a remote event forwards the event to the other agents that appear in the event's subscriber list.

C. Trigger Table

A trigger rule indicates which event detectors should be executed when a given event occurs. Such trigger rules are put in the *Trigger Table*. Each detector specifies which other events can trigger its detection process. The trigger table is updated when a detector is installed through a *Monitor Interface* method. In Figure 4, the occurrence of event A will cause triggering of the detection procedures for events B and C.

The triggering event is passed to each detector to allow the detection procedure to be executed with information contained in that event.

D. Interface to Prolog Database

Subscriber agents maintain event data from the monitored nodes in a logic database to generate correlated events. This database is maintained using Prolog. We are using a Java-based Prolog system [14] for this purpose.

An event detector or handler in the agent can access this database object. The event handlers add events to this database as new facts. A detector can use this database object to add a new Prolog rule and execute a query. A monitor agent can also use an event database if it needs to correlate its locally monitored events.

IV. SECURITY OF THE MONITORING ARCHITECTURE

Ajanta [12] is a secure Java-based framework for programming mobile agents. In Ajanta, mobile agents are *mobile objects*, which can migrate autonomously in distributed environments. Agents encapsulate code and execution context along with data, and they are executed on behalf of a user, termed *owner*. The Ajanta system provides *agent servers* to host mobile agents, a set of primitives for the creation and management of agents, and a global naming service. Security is an integral part of Ajanta design, and Ajanta provides components for authentication, public key maintenance, access control, host resource protection, and cryptographic services. Interested readers can refer to [12], [15] for further details. In this section, we discuss how Ajanta's security facilities are utilized to protect the monitoring system.

- Access control on the acceptance of visiting agents is imposed by the agent servers based on the owner's identity. In Ajanta, all globally accessible entities, like an agent's owner, are given uniform resource names (URNs), and the name service securely maintains a name including its public-key certificates. An agent server uniquely identifies an agent's owner using a challenge-response protocol and enforces security policies for acceptance of his/her agents.
- In Ajanta, each agent is given a set of unforgeable credentials, which contain the agent's name, its owner's name, and a set of privileges granted by the owner. Based on an agent's credentials, an agent server grants to the visiting agent restricted access to its local resources, like privilege to read system log files, check process status, or remove old files from the */tmp* directory. Agents at two different nodes can communicate with each other using RMI, provided their hosting servers permit them to open TCP-connections.
- In Ajanta, agent communication can be authenticated, provided the remote hosting server is trusted. For each remotely called object, like the *Monitor* and *Subscriber* object, the agent servers authenticate and impose access control on the methods. Hence, the method invoker needs to provide its name as a parameter to the invoked method as shown in the interface declaration in Section 3.
- An administrator can also securely control its remote agents by either terminating their execution or recalling them. The secure *RemoteControl Interface* ensures that

only the valid owner can disable its agents.

- If different administrative domains are present, our agent servers can easily impose domain specific access policies. In our monitoring architecture, the agent servers are trusted unless the host where an agent server resides is compromised.
- Agents can also impose access control on its methods based on its own access control policy. A monitor agent can impose restriction on its operations defined like adding/deleting detectors, handlers and subscribers. A subscriber agent can impose access control on its event publisher agents based on publisher agents' identities, types of events, and can control the threshold rate of event notification from individual publisher agents.
- In Ajanta, agent servers provide distinct protection domains for its visiting agents, which ensures that visiting agents do not interfere with each other.
- Ajanta's secure class loader ensures that the agent classes that are not core JVM system classes are loaded from a secure repository as specified by the agent server where the agent is created. When a new detector is added remotely, the class loader ensures untrusted code is not loaded in the agent.

V. EXPERIMENTS IN AGENT BASED NETWORK MONITORING

A. Experiment Goals and Scenarios

In this work we have mainly focussed on the following three requirements discussed in Section 1: dynamic configuration, extensibility, and active monitoring. We designed and conducted a set of experiments to test and demonstrate that the functional capabilities of our agent-based monitoring framework met these requirements. In these experiments we did not intend to build a complete monitoring system or an operational intrusion detection system.

We did not run any specific experiments with authentication and access control policies. The Ajanta facilities for secure agent execution and authenticated RMI communication have been extensively used and tested in the past, and therefore integration of these mechanisms in a monitoring system based on the framework presented here is not a major issue. However, a system administrator would need to define the access control policies for an agent in regard to who should be allowed to install or remove an event detector or alter the publisher-subscriber relationships.

We set up an experimental testbed in our lab environment, and then conducted experiments driven by a set of scenarios to incrementally add new monitoring capabilities dynamically. To test the dynamic configurability, our goal

was to show that any desired configuration of the experimental monitoring system could be set up and changed dynamically. Our experiments involved the creation and dispatching of new agents, establishment of any desired publisher-subscriber relationships between agents, and addition of new event detectors and handlers to an agent.

In regard to extensibility, our objective was to show that this system was able to support definition of new event types, and their detection/handling could be easily added to an operating environment. We selected a set of scenarios to define new compound events representing correlation of some existing ones. Through these scenarios, our goal was to demonstrate that the use of Prolog in our system provides an elegant and easy-to-use high level abstraction for both defining new compound (correlated) events and building their detectors. Another goal was to demonstrate the ability of this system to support active monitoring by using trigger rules.

The following is a brief description of the scenarios that we identified for our experimental tests.

Invalid Logins: This corresponds to the condition when a user outside of our research group is logged into any of the machines in our lab.

User Presence: This defines the condition when a user physically logged in on any of the machines in our lab.

Remote User Presence: This condition represents a user login originating from a node outside our lab environment.

Remote and Local Logins: This corresponds to situations when a user is physically logged in on a machine in the lab environment and at the same time a remote login by that user from an outside domain is observed. This can be considered as a suspicious situation.

User Switch Events: This event arises when a user switches to another user. This indicates that the initiating user possesses the password of the second user, or some collaboration between the two users is occurring. The detection of this condition is triggered whenever a successful login event takes place at any machine in the system. It requires correlation of login events from all nodes.

Multiple User Switch: This represents the situation when a user switches to two or more other users. In our system we do not expect any user to have access to more than two accounts.

FTP Alarms: This corresponds to a situation when successful or failed logins are observed from a host that was refused FTP connections. This is an alarm for the administrator to check for a possible FTP attack situation.

Status of /tmp directory: This detects the situation when the /tmp directory on a host exceeds certain space limit or has some very old files.

CPU Usage Alarm: This detector checks if the CPU uti-

lization on a host is higher than some threshold or some runaway process has been executing for a long time.

Checking file consistency: This detector periodically checks if any changes have occurred in the `/bin` directory. Malicious users or worms may overwrite common commands in the `/bin` directory.

B. Experimental Testbed

Our experimental system was set up on our lab environment of ten nodes, and it was similar to the configuration shown in Figure 3. An agent server was started on each of the nodes. The *system management station* reads a configuration file which contains information for launching monitor agents with some initial event detectors. The management station then sends one monitor agent to each host to be monitored. For our experiments, these agents monitored events of class `SyslogEvent`, namely `FtpEvent`, `RLoginEvent`, `TelnetEvent`, `SshEvent`, `XdmEvent` etc. The events were then sent to the subscribers where more powerful correlative event detectors were added.

The system management station provides a user interface through which we launch new monitor agents and subscriber agents, or modify existing agents by adding or deleting event detectors. It is also possible to have multiple system management stations launching agents in the system, as shown in Figure 3.

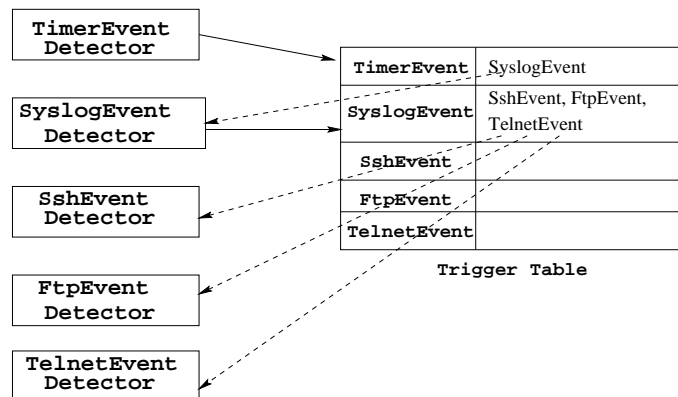


Fig. 5. Example of trigger dependency (Partial)

Figure 5 illustrates the trigger dependencies used by the monitor agents in our experiments. The *TimerEventDetector* periodically generates a *TimerEvent*, which then triggers the *SyslogEventDetector*. This detector reads any new lines logged in the syslog file since its last inspection. For each new line that it reads, it creates a new *SyslogEvent*, and triggers all the dependent detectors. The triggered event detectors extract the string from the *SyslogEvent*, and try to match the string with a predefined regular expression using a Java-Perl object. If the pattern is matched, an event

is generated and added to the event queue for subsequent handling by the processing thread.

We conducted experiments to demonstrate that one could easily add a new kind of `SyslogEventDetector` to a running monitor agent. To detect a new event from the syslog file, we first examine the structure of the event description line in the syslog file, and then generate the Perl pattern to detect it. For example, the following line is generated in syslog file for an SSH service.

```
May 23 11:33:30 fourier.cs.umn.edu sshd[1538]:
connect from user@plato.cs.umn.edu
```

The corresponding intermediate form of the pattern for the above line that we use to generate the Perl regular expression is shown below:

```
(WORD)(NUM) (NUM:NUM:NUM) (TOKEN) sshd[(NUM)]:
connect from (TOKEN)@(TOKEN)
```

We then define a new event class and a corresponding detector class to perform event detection using the Perl pattern. On a successful match, a list of matched tokens as specified above is returned, and the detector builds the corresponding event object. This new detector are then remotely installed on an agent.

In our experiments, two subscriber agents were created to receive notifications for all of the events monitored by each of the monitor agents. Each subscriber agent maintains event records in a Prolog database. As described in the next section, several new compound event types were incrementally defined in this system using Prolog-based correlative detectors. These were dynamically installed on the subscriber agents. Some of these detectors generated *alarm events*, which were delivered to the management station.

C. Compound Event Detectors

This section presents an example query in Prolog, explains how Prolog rules are added to detectors, and then describes how Prolog was used to implement six of the sample scenarios discussed earlier.

C.1 An Example Prolog Query

An example of a query would be to search for logins by a particular user. A login event fact in the Prolog database has the form:

```
loginEvent(eventId, eventName, host, date,
           timeInMillis, processID, initiatingHost,
           initiatingUser, targetUser, success).
```

The last element in this rule contains true or false to indicate whether the login succeeded or not, the `initiatingHost` is the host that is attempting to connect to the machine that generated this event. A query that would look

for all login events from the user “codeRed” is written below. In Prolog, words that begin with capital letters are unbound variables, and words that begin with lowercase letters are bound to that value.

```
loginEvent(EventId,EventName,Host,Date,
           TimeOccurred,Pid,ConnectingHost,
           InitUser,codeRed,Success).
```

Running the query in the Prolog engine will produce the following result.:

```
EventId = uniqueEventID,
EventName = loginConnect,
Host = newton.cs.umn.edu,
Date = date(thu,jul,19,17:16:06,cdt,2001),
TimeOccurred = 995494566000,
Pid = 3135,
ConnectingHost = IhackIISServers.com,
InitUser = null,
Success = true;
```

This query will return true for any login event where “codeRed” is the target user. If we would like to check for all logins from “codeRed” on July 19, 2001, our rule would look like:

```
loginEvent(EventId,EventName,Host,
           date(thu,jul,19,Time,Zone,2001),
           TimeOccurred,Pid,ConnectingHost,
           InitUser,codeRed,Success).
```

C.2 Building of a Prolog-based Detector

A Prolog-based detector object stores its Prolog rule within itself. When triggered by an event, the detector adds its rule to the Prolog database. Next, an object array is created whose length matches the number of arguments in the query. This object array is passed to the Java-Prolog interface. In the query discussed above, an object array with ten elements will be created. The ninth element of this array, which represents the user, is defined to be “codeRed”.

If the query is successful, all the undefined entries of the object array are set by the Java-Prolog interface. If there are multiple matches to the query, multiple object arrays are returned in a vector. The detector uses these results to generate the appropriate events. A detector may generate one event for each matched result. A detector can also be programmed to generate a single event based on all the matched results. Upon completion of the query, the detector erases any rules it has added to the Prolog database.

C.3 InvalidUserAlarm Detector

A basic compound event detector is the InvalidUserAlarm event detector. This detector is triggered by LoginEvent, SshEvent, XdmEvent and FtpEvent. An administrator could define a list of valid users. An entry for a valid user in our system is below:

```
validUser(mcarney).
```

When a user connects to a machine, by any service, the user would be checked against this list to see if that user should have access to the system. In Prolog, multiple sets of conditions are separated by semicolons. If the first set of conditions fails, the Prolog engine will look at the next set. Underscores inside a rule indicate a variable that Prolog will not attempt to match or bind to a specific value. In Prolog, this is referred to as an anonymous variable. The rule for the InvalidUserAlarm detector is:

```
invalidUserLogin(EventID,Host,Date,User):-
  loginEvent(EventID,_,Host,Date,_,
             _,_,User,true),
  not(validUser(User)),User \= null;
  sshEvent(EventID,_,Host,Date,_,
           _,_,User,true),
  not(validUser(User)),User \= null;
  xDmEvent(EventID,_,Host,Date,_,
           _,_,User,true),
  not(validUser(User)),User \= null;
  ftpEvent(EventID,_,Host,Date,_,
           _,_,User,true),
  not(validUser(User)),User \= null.
```

When the triggering event is found in the database the values of Host, Date, and User are bound to the corresponding values of that event. If the query succeeds, an InvalidUserAlarmEvent will be generated with the data that was returned.

C.4 UserPresence Detector

The UserPresence detector checks to see if a user is logged in through the console. This detector is triggered by an XdmEvent. It pulls out all XdmEvents for the user of the triggering event, and checks the last one. If the last XdmEvent has the event name “xdmOn” the user is still logged in, otherwise the user has already logged off. This rule could be easily modified to create an event every time a user, whom an administrator would like to watch, logs in through the console. In a collaboration environment such an event could be used to indicate a user is present and ready to participate.

C.5 OutsideDomainLogin

OutsideDomainLogin detects logins from non-local domains. In our experiment, we defined domains that were considered local, and all other domains were considered to be remote. However, the system could be refined in the future to check domains based on IP addresses. This detector is triggered by LoginEvent, SshEvent, and FtpEvent. A local domain entry in our database has the form:

```
localDomain('newton.cs.umn.edu').
```

The rule `OutsideDomainLogin` is very similar to the rule for the `InvalidUserAlarm` detector, except that it does not consider `XdmEvents`. As in the `InvalidUserAlarm` detector, after the triggering event is located in the database one more check is run. In this case, we check to see if the host is remote or local.

C.6 OutsideAndLocalLogin Detector

The `OutsideAndLocalLoginEvent` detector is triggered by an `OutsideDomainLogin` event. It checks to see if the user is logged in through the console at the same time that the user is logged in remotely. It uses the same method as the `UserPresence` detector, but is not triggered by it. This situation could indicate a compromised account.

C.7 UserSwitch Detector

The `UserSwitchEvent` detector is a basic event which detects when a user has gained access to another account through the Telnet, FTP or Rlogin services. This detector defines four rules to search the database. Three of these rules define Telnet, FTP and Rlogin switch user. The main rule looks for a switch user event based on these three rules. Below is the rule for `telnetSwitchUser`, the rules for FTP and Rlogin are similar. Because a successful Telnet login generates two lines in the log file, process ids are used to match a Telnet connection with its login process. In our observations, this difference has always been three.

```
telnetSwitchUser(InitUser,
  TargetUser,EventId,Type):-
  telnetEvent(_,telnetInitiating,_,_,_,
    TelnetPid,_,InitUser,_,_),
  loginEvent(EventId,loginConnect,_,_,_,
    LoginPid,_,_,TargetUser,_,
    LoginPid>TelnetPid,
    LoginPid-TelnetPid == 3,
    InitUser /= TargetUser, InitUser /= null,
    TargetUser /= null, Type=telnet.
```

When executing this detector's rule the Prolog engine will first look for `telnetSwitchUser` event using the `telnetSwitchUser` rule, then similarly for FTP and Rlogin.

C.8 MultipleUserSwitch Detector

To enforce a system policy, such as a rule that any particular user can have access to no more than two unique accounts, we could easily add a `MultipleUserSwitch` detector with a simple rule that would look at `userSwitch` events in the database. The rule below would detect "User1" switching to "User2" and then "User2" switching to "User3", or "User1" switching to "User2" and "User1" switching to "User3". The rule would be triggered by a `UserSwitch` event.

```
multipleSwitch(EventId,User1,User2,User3):-
```

```
  userSwitchEvent(EventId,_,_,_,User1, User2,_),
  userSwitchEvent(.,.,.,.,User2, User3,_),
  User1 \= User3;
  userSwitchEvent(EventID,_,_,_,User1, User2,_),
  userSwitchEvent(.,.,.,.,User1, User3,_),
  User2 \= User3.
```

C.9 FtpAlarm Detector

The `FtpAlarmEvent` detector checks for successful or failed logins from hosts who have been refused FTP connections. This detector is triggered by `FtpEvent`. Here is the Prolog rule it uses:

```
ftpAlarmEvent(EventId,User,Host):-
  ftpEvent(EventId,ftpConnectionRefused,
    Host,_,_,_,_,_,User,_,_),
  loginEvent(.,.,Host,.,.,.,.,.,.);
  ftpEvent(EventId,ftpConnectionRefused,
    Host,.,.,.,.,.,User,_,_),
  sshEvent(.,.,Host,.,.,.,.,.,.);
  ftpEvent(EventId,ftpConnectionRefused,
    Host,.,.,.,.,.,User,_,_),
  ftpEvent(.,ftpSuccess,Host,.,.,.,.,.,.).
```

This rule will find all `FtpEvents` which have the event id of the triggering event and the event name "ftpConnectionRefused". It will then check if there are corresponding login events from that domain.

VI. RELATED WORK

We have discussed earlier some of the limitations of SNMP and cron jobs for system monitoring. SNMP can be integrated as one of the low-level building blocks in our framework. Cron jobs are typically executed during off-peak hours and they do not support continuous monitoring for abnormal conditions. Due to the dynamic nature of today's networks, it is easy for such scripts to miss an intermittently connected machine. Installing a new script on a remote machine tends to be a fairly time consuming and manual task, as most systems tend to lack a coherent and secure middleware service for automatically and securely installing new scripts on remote machines. Generally, the monitoring and correlation functions executed by the scripts deal with low level representation for events, strongly tied to the operating system platform.

We are also interested in monitoring a system for critical events that represent abnormal situations. In this regard, our monitoring framework relates to intrusion detection systems (IDS) such as Haystack[16], IDES[17], MIDAS[18], and DIDS[4], [3]. However, network monitoring not only provides functionality to monitor users' intentional misuses or intrusions but also monitors system inconsistencies introduced by any means. The primary objective of our design has been to provide an extensible infrastructure to allow active monitoring of critical events

rather than investigation of a specific set of techniques for intrusion detection. Thus, one should be able to easily incorporate in this framework a new or existing technique for intrusion detection.

Current hierarchical IDSs, like Emerald [19] and Grid [20], solve the problem of a single point of failure and scalability. However, they are difficult to reconfigure or add new capabilities and functionalities. These IDSs usually have to be restarted to make any changes take effect. A mobile agent based monitoring system can be dynamically reconfigured, or new functionalities can be added in the system by creating and launching new agents with added functionalities to co-exist with the old ones.

Advantages of using agents have been mentioned in the context of an IDS called AAFID (Autonomous Agents For Intrusion Detection) [10]. Agents can be upgraded when increased functionality is required while keeping backward compatibility. AAFID is programmed in Perl and it demonstrates the feasibility of an agent-based intrusion detection system, but it lacks a well defined middleware or mobile agent programming environment to fully exploit the benefits of the agent paradigm in such systems. In contrast, our work has focused mainly on the building of an extensible and secure middleware using the mobile code technology supported by a mobile agent platform.

Our work also has some relationship to active database systems [21], which provide mechanisms by which a database is able to automatically respond to events. The use of trigger mechanisms with event detectors in our design provides this kind of functionality. This trigger mechanism in our design is implemented outside the Prolog-based logic database. Here also, our primary objective has been to support dynamic extensibility so that new detectors and triggers can be added at runtime. The use of Prolog to detect correlated events provides us higher level abstractions to define and detect events, and the use of triggers facilitates event-based execution of such detectors.

The recent developments in the field of mobile agent technology has motivated several researchers to investigate its application to network management [9], [22]. The benefits of code mobility in network management are discussed in [23]. These efforts have mainly focused on code mobility to reduce network traffic. In contrast, our work has focused on building a secure infrastructure to support extensibility, dynamic configuration, and active monitoring. There are also some commercial efforts ongoing today on the use of Java/Web for network management; however, they are not striving to take advantage of the mobile agent technology. Overall, practical experience demonstrating the capabilities of mobile agents in network management is lacking. Our work has been mainly motivated to address

such needs.

VII. CONCLUSIONS

This paper identifies the important requirements of emerging network monitoring systems. Through the development of an experimental monitoring system, we have demonstrated that code mobility supported by a mature mobile agent platform can be effectively used to meet some of the important requirements of emerging network monitoring systems. In this work, we focused on the requirements of dynamic configurability, extensibility, active monitoring, and security. Dynamic configuration for network monitoring is achieved as monitor agents are launched for newly added network components at runtime without restarting the monitoring system. A dynamically pluggable, modular architecture for event detection and processing is shown. The mechanisms to dynamically add or change event detectors, handlers, and subscribers to a monitoring agent support the extensibility requirement.

An important contribution of this paper is to show that high level specifications for monitoring goals can be easily expressed in Prolog and integrated with monitoring agents. Prolog enables us to define new compound events based on the ability to correlate event data. Such goals are used directly to build the corresponding event detectors. Our system provides the functionality to add these new detectors dynamically to the monitoring system. We have presented here a number of examples of Prolog based detectors.

Another important aspect of our system is active monitoring, which is achieved by triggering events based on related events. A trigger dependency structure can be set up without relying on the type of database supported by agents or the detection procedure used in the event detectors. In our experimental testbed, we showed how trigger dependencies can achieve active monitoring of our lab environment.

The system presented here shows the capabilities of mobile agent technology for network monitoring. One might argue that our system launches agents only to a single node and is not different from mobile code technology based on Java reflection and code loading. However, the requirements of a network monitoring system demand secure mechanisms for remotely controlling and modifying the monitoring system's components. The Ajanta system provides these mechanisms.

There are several directions for our future work in this area. The current work has not yet addressed the requirements of performance and scalability, and cross-domain administrative security policies. In this work we have only used a subset of the paradigms for mobile agent based network monitoring presented in [24]. We plan to deploy this

monitoring framework on a larger configuration in our departmental network to investigate the areas not addressed here. This will address scalability and cross-domain security issues. Also, we plan to investigate other mobile agent based network monitoring paradigms, such as the use of migratory *inspector* and *auditor* agents.

The use of mobile agent technology in network management and monitoring is a relatively new and largely unexplored area of research[9]. Our work demonstrates the potential of this technology in building the future generation network monitoring systems.

REFERENCES

- [1] Masoud Mansouri-Samani and Morris Sloman, "Monitoring Distributed Systems," *IEEE Network*, pp. 20–30, November 1993.
- [2] Raouf Boutaba, Karim El Guemhioui, and Petre Dini, "An Outlook on Intranet Management," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 92–9, October 1997.
- [3] Gregory B. White, Eric Fisch, and Udo Pooch, "Cooperating Security Managers: A Peer-Based Intrusion Detection System," *IEEE Network*, pp. 20–23, January/February 1996.
- [4] Biswanath Mukherjee, L. Todd Haberlein, and Karl N. Levitt, "Network Intrusion Detection," *IEEE Network*, pp. 26–41, May/June 1994.
- [5] William Stallings, "SNMP and SNMPv2: the infrastructure for network management," *IEEE Communications Magazine*, vol. 36, no. 3, pp. 37–43, March 1998.
- [6] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum, "Mobile Agents: Are they a good idea?," Tech. Rep., IBM Research Division, T.J.Watson Research Center, March 1995, Available at URL <http://www.research.ibm.com/massdist/mobag.ps>.
- [7] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998.
- [8] Neeran M. Karnik and Anand R. Tripathi, "Design Issues in Mobile Agent Programming Systems," *IEEE Concurrency*, vol. 6, no. 6, pp. 52–61, July–September 1998.
- [9] Robert Pinheiro, Alex Poylisher, and Hamish Caldwell, "Mobile Agents for Aggregation of Network Management Data," in *First International Symposium on Agents and Applications, and Third International Symposium on Mobile Agents*, October 1999, pp. 130–140.
- [10] Jai Balasubramanian, Jose Omar Garcia-Fernandez, David Isacoff, E. H. Spafford, and Diego Zamboni, "An Architecture for Intrusion Detection using Autonomous Agents," Tech. Rep. Coast TR 98-05, Department of Computer Sciences, Purdue University, 1998.
- [11] Mark Crosbie and Eugene H. Spafford, "Defending a Computer System using Autonomous Agents," in *Proceedings of the 18th National Information Systems Security Conference, Baltimore MD, USA*, October 1995, pp. 549–558.
- [12] Anand Tripathi, Neeran Karnik, Manish Vora, Tanvir Ahmed, and Ram Singh, "Mobile Agent Programming in Ajanta," in *Proceedings of the 19th International Conference on Distributed Computing Systems*, May 1999.
- [13] Neeran Karnik and Anand Tripathi, "A Security Architecture for Mobile Agents in Ajanta," in *Proceedings of the International Conference on Distributed Computing Systems 2000*, April 2000.
- [14] "JIPL: Java Interface for Prolog," Available at URL http://Prolog.isac.co.jp/index_e.html, 2001.
- [15] Neeran Karnik and Anand Tripathi, "Security in the Ajanta Mobile Agent System," *spe*, vol. 31, pp. 301–329, January 2001.
- [16] S.E. Smaha, "Haystack: An Intrusion Detection System," in *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, 1988.
- [17] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Newumann, and C. Jalali, "IDES: A Progress Report," in *Proceedings of the 6'th Annual Computer Security Applications Conference*, 1990.
- [18] M.M. Sebring, et al., "Expert systems in intrusion detection: A case study," in *Proceedings of the 11'th National Computer Security Conference*, October 1988.
- [19] Phillip A. Porras and Peter G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances," in *Proceedings of the Nineteenth National Computer Security Conference*. 1990, May, pp. 296–304.
- [20] S. Staniford-Chen, S. Cheung, R Crawford, M. Dilger, J. Frank, J. Hoagland, C. Wee K. Levitt, R. Yip, and D. Zerkle, "GrIDS: A graph based intrusion detection system for large networks," in *Proceedings of the 19th National Information Systems Security Conference*. National Institute of Standards and Technology, October 1996, pp. 361–370.
- [21] Norman W. Paton and Oscar Díaz, "Active database systems," *ACM Computing Surveys*, vol. 31, no. 1, pp. 63–103, 1999.
- [22] P. Bellavista, A. Corradi, and C. Stefanelli, "An Open Secure Mobile Agent Framework for Systems Management," *Journal of Network and Systems Management (JNSM)*, vol. 7, no. 3, pp. 323–339, September 1999.
- [23] Mario Baldi, Silvano Gai, and Gian Pietro Picco, "Exploiting Code Mobility in Decentralized and Flexible Network Management," in *Proceedings of the Workshop on Mobile Agents (MA'97) – LNCS 1219*, April 1997, pp. 13–26.
- [24] Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, Megan Carney, and Paul Dokas, "Paradigms for Mobile Agent-Based Active Monitoring of Network Systems," Tech. Rep., Department of Computer Science, University of Minnesota, July 2001, <http://www.cs.umn.edu/Ajanta>.