INTERNATIONAL JOURNAL OF NETWORK MANAGEMENT
Int. J. Network Mgmt 2008; **18**: 295–302
Published online 4 June 2007 in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/nem.655

ANEMONA: a programming language for network monitoring applications

Elias Procópio Duarte Jr*,†,[1], Martin A. Musicante[2] and Henrique Denes H. Fernandes[1]

[1]*Federal University of Paraná, Department of Informatics, PO Box 19018, Curitiba 81531-990 PR, Brazil e-mail: elias@inf.ufpr.br*
[2]*Federal University of Rio Grande do Norte, Department of Computer Science, Campus Universitário Lagoa Nova, Natal 59072-970 RN, Brazil*

SUMMARY

This work presents ANEMONA: A language for programming NEtwork MONitoring Applications. The compilation of an ANEMONA program generates code for configuring a policy repository and the corresponding policy deployment and event monitoring. The language allows the definition of expressions of managed objects that are monitored, as well as triggers that when fired may indicate the occurrence of associated events, which are also defined by the language. A translator for the language was implemented that generates code for configuring both the policy repository and deployment. The current implementation of the language employs the Expression MIB and Event MIB. Experimental results are presented, including an ANEMONA program that detects TCP Syn Flooding attacks, and a program for detecting steep variations in the utilization of monitored links. Copyright © 2007 John Wiley & Sons, Ltd.

1. INTRODUCTION

As current network management systems are responsible for monitoring and controlling increasingly large and complex networks and systems, distributing management tasks is often required. The policy-based management paradigm [1–3] has been seen as the architecture of choice for dealing with the new challenges and requirements. Policy-based network management provides the expected functionality, and at the same time has the potential to keep a low impact on network performance while increasing the network's dependability. This work describes the ANEMONA (A NEtwork MONitoring Application) language. ANEMONA is a simple programming language that allows the definition of policies which are expressions of managed objects, stored at a policy repository [4]. The language is used to define conditions related to monitored objects that when detected cause the generation of alarms or the execution of predefined procedures. Through ANEMONA it is possible to generate code for specifying, deploying policies and monitoring the corresponding events within the Internet standard SNMPv3 (Simple Network Management Protocol version 3) framework.

The SNMPv3 management architecture defines management entities that fit a distributed policy-based management paradigm [5]. Management entities may assume different roles, keeping a Management Information Base (MIB) that provides both an interface to the data available at the entity and the behavior of that entity.

In the current implementation of ANEMONA, we have employed two well-known but seldom employed standards: the Expression MIB [6] and the Event MIB [7]. The Expression MIB allows the definition and evaluation of expressions built from readings of managed objects. The Event MIB [7] defines

*Correspondence to: Elias Procópio Duarte Jr, Federal University of Paraná, Department of Informatics, PO Box 19018, Curitiba 81531-990 PR, Brazil.
†E-mail: elias@inf.ufpr.br

Received 25 February 2007
Revised 5 March 2007
Accepted 18 March 2007

a set of objects to be monitored and associated conditions: an event is triggered upon the occurrence of a condition which causes either the emission of alarms or the execution of a procedure.

A compiler was developed for the language which generates code that configures both the Expression MIB and Event MIB. The manual configuration of these MIBs is a hard task that requires the manager to learn details of the internal details of those MIBs. By using ANEMONA the manager can easily use the MIBs without needing to understand their internal structure or to write a long list of configuration commands.

The rest of the paper is organized as follows. Section 2 describes the ANEMONA language, as well as the proposed compiler. Section 3 describes the current implementation of the language, including descriptions of both the Expression MIB and the Event MIB. Section 4 describes case studies, including an ANEMOMA program that detects TCP Syn Flooding attacks and another program for determining a steep increase in the utilization of a monitored link.

2. THE ANEMONA LANGUAGE

A program in ANEMONA has three main parts: a *prologue* defining the location of the agent to be configured, a *declarations part* to define the managed objects, and a *control part*.

A program in ANEMONA has the form[1]:

watch ‹*Monitor*› using ‹*Community*›
‹*Declarations*›
begin
‹*Commands*›
end

The first line corresponds to the prologue. It contains just the watch directive. The 'Monitor' field must contain the host in which the agent being configured is executed. The field 'Community' is mandatory. ANEMONA 'Declarations' and 'Commands' are described below.

Declarations in ANEMONA contain directives of the form:

‹*OID*› is ‹*Type*›:‹*sampling*›

where 'OID' is an object identifier (being declared). The field *Type* defines the type of object, while the field *sampling* corresponds to the sampling method used for the object. ANEMONA supports the predefined types: Integer, OctetString, OID (object identifier), IPAddress, Counter32, Unsigned, TimeTicks and Counter64. These types are a subset of those defined for SMI [8].

The sampling methods include absolute, delta and modified, corresponding respectively to absolute values, delta values, and a Boolean, which is true if the object's value is changed.

The control directives in ANEMONA include expressions (returning values), commands, macros, triggers and function definitions. Each of these classes of directives is explained in the following subsections.

2.1 Expressions

ANEMONA possesses a rich set of expressions, including arithmetic, relational, Boolean, bit-wise, string concatenation and conditional expressions.

Arithmetic operations include the four basic operations as well as integer modulo (%). Arithmetic operators can be applied to operands of the following types: Integer32, Counter32, Counter64, IpAddress, Unsigned and TimeTicks. Arguments to the arithmetic operators can be of mixed types. Conflicts are solved using the following rules, listed in order of precedence:

[1]*Reserved words and program syntax are written with this typeface, while nonterminals are written ‹like this›.*

DOI: 10.1002/nem

1. The unary subtraction '–' always returns a value of type Integer32.
2. If both operands of a binary operator are of the same type, then the result of the operation will be of that same type.
3. If one operand of a binary expression is of type Counter64, then the result of the operation will be of type Counter64.
4. If one operand of a binary expression is of type IpAddress, then the result of the operation will be of type IpAddress.
5. If one operand of a binary expression is of type TimeTicks, then the result of the operation will be of type TimeTicks.
6. If one operand of a binary expression is of type Counter32, then the result of the operation will be of type Counter32.
7. In all other cases, the result of the operation will be of type Unsigned.

Relational operators in ANEMONA include equality (==), inequality (!=), greater-than (>), greater-or-equal-than (>=), less-than (<) and less-or-equal-than (<=). Their evaluation returns an Unsigned value. They follow the rules of the C language.

Boolean operators include conjunction (and), disjunction (or) and negation (not). Their evaluation returns an Unsigned value.

Bitwise operations include bit-wise and (AND), bit-wise or (OR), bit-wise not (NOT) and bit-wise exclusive or (XOR). Both operands of these operators must be of the same type (which is also the type of the result). The only permitted types for these expressions are OID and IPAddress.

For types OctetString and OID, the concatenation operation '+' is defined. Both operands must be of the same type.

ANEMONA defines some predefined functions for casting and information about objects. For instance, the function Counter32(. . .) converts any integer value to the type Counter32. The function exists(. . .) takes an object and returns an unsigned (representing a truth-value). This function indicates whether the argument is a valid instance of an object.

Expressions can be grouped by using parentheses.

2.2 Macro Definitions

Macro definitions are used to bind an identifier (the name of the macro) to an object or to the result of an expression. Macros can be used anywhere in the program, simplifying the programmer's task. However, the main use of macro definitions in ANEMONA is in order to reference the entries of the results table on the *Expression MIB*. Macro names will reference those entries whose expressions are defined within the program. After the execution of an ANEMONA program, the tool will report the name of the object bound to each macro definition, so that the administrator can collect the result of intermediary expressions.

The syntax of macro definitions is given below:

bind ‹*MacroName*› to ‹*Expression*›

where ‹*MacroName*› is an identifier (the name of the macro) and ‹*Expression*› is any expression of the language.

2.3 Basic Commands

The basic commands in ANEMONA include conditional statements, assignments and notifications.

Conditional statements in ANEMONA are implemented by an operation that keeps updated the value of an object. The statement

if ‹*Expression*› then ‹*Value*$_1$›
else ‹*Value*$_2$›
rec by ‹*OID* | *MacroName*›

DOI: 10.1002/nem

represents a conditional statement in which the object being updated is given by a macro name or an object ID (appearing after the keyword rec by. The value of this object will become ‹$Value_1$› or ‹$Value_2$›, depending on the truth-value of the expression.

Assignment commands in ANEMONA are implemented by the set primitive, whose syntax is given as follows:

set ‹*OID*› at ‹*IpAdd*› using ‹*Community*› to ‹*IntegerValue*›

In this command, the value of the object ‹*OID*› will be changed to ‹*IntegerValue*›. The IP address and community used by the object are mandatory.

Notice that the value to be assigned must be an integer.

Notifications in ANEMONA are signaled by using the directive notify, whose syntax is defined as follows:

notify ‹*IpAdd*› ‹*Community*› ‹*OID*›

This command simply sends a notification to a given manager. The contents of the trap is given by an object. Notice that the community is mandatory.

Triggers

Triggers in ANEMONA are implemented by the directive when, whose syntax is given as follows:

```
when ‹Expression› do
  ‹List of Commands›
end
```

The above command is implemented in such a way that when the guard of the command becomes true then the list of commands is executed. This command is implemented by:

- programming the expression within the *Expression MIB*;
- for each command of the list, use the *Event MIB* to:
  — configure a trigger to monitor the result of the expression;
  — configure an event for each action to be taken.

3. THE CURRENT COMPILER

The ANEMONA system includes a compiler and a run-time system. The compiler was constructed using standard techniques [9]. The run-time system is composed by an implementation of a subset of the Event and Expression MIBs. ANEMONA is a front-end tool: it takes a higher-level description of the management information and generates a set of basic operations, such as *snmpget* and *snppset*, to configure the Expression and Event MIBs, described below.

The Expression MIB [6] allows the definition of expressions which are built with existing management objects. After an expression is evaluated the result is available as a MIB object. Thus, the Expression MIB is a way to create new, customized MIB objects for monitoring.

The Expression MIB supports three different types of sampling: absolute, delta (difference from one sample to another), and changed (indicates whether or not the value of the object has changed since the last sample). If there are no delta or changed values in an expression, the evaluation occurs on demand. For expressions with delta or change values, the evaluation goes on continuously, every sampling interval. In this case requesters get the value as of the last sample period.

The Expression MIB has three sections: Resource, for management of the MIB's use of system resources; Definition, which contains tables that define expressions; and Value, which contains values of evaluated expressions.

DOI: 10.1002/nem

The Definition section contains the two main tables used to define expressions. The expression table, indexed by expression owner and expression name, contains the parameters that apply to the entire expression, such as the expression itself, the data type of the result, and the sampling interval if it contains delta or changed values. The object table, indexed by expression owner, expression name and object index within each expression, contains the parameters that apply to the individual objects that go into the expression, including the object identifier and sampling type.

The syntax of expressions, as well as the procedure for MIB configuration is given in Kavasseri and Stewart [6].

The Event MIB [7] allows a local or remote object to be monitored; when a trigger condition is met, an action is executed, which is the generation of a notification or setting a MIB object, or both.

The MIB has four sections: triggers, objects, events, and notifications. Triggers define the conditions that lead to events. Events may cause notifications. The trigger table lists what objects are to be monitored and how, besides relating each trigger to an event. Other tables exist that define the type of test to be done for the trigger. The objects table lists objects that can be added to notifications based on the trigger, the trigger test type, or the event that resulted in the notification.

Two types of tests can be defined: Boolean and existence. A Boolean test requires the type of monitored object to be integer, and the definition of a test. In the trigger section a reference value is defined and is compared with the sampled value. If the execution of the comparison returns true, given the defined test, then an event occurs. An existence test leads to an event when the monitored object instance exists, or does not exist, or even if its value has changed since the previous sampling.

The event table defines what happens when an event is triggered: sending a notification, setting a MIB object, or both. It has supplementary, companion tables for additional objects that depend on the action taken. The notification section defines a set of generic notifications to go with the events.

The Expression MIB provides custom objects for the Event MIB [7]. A complex expression can be evaluated and then be subject to testing as an event trigger, resulting in an SNMP notification. Without these capabilities such monitoring would be limited to the objects in predefined MIBs. The combination of the Expression MIB and the Event MIB provide powerful tools for the self-management of large and complex systems.

4. CASE STUDIES

This section describes two case studies, which consist of ANEMONA programs employed for practical network monitoring, and experimental results obtained from their execution. The first program generates a notification for the administrator whenever a TCP SYN Flooding attack is detected. The second program detects steep variations in the utilization of monitored links.

The environment in which experiments were executed for both case studies described here consisted of Intel and AMD processor-based hosts running Linux, NET-SNMP [10] agents, our own implementations of both the Expression and Event MIBs, our ANEMONA translator, HTTP, FTP and telnetd servers. The network connecting the hosts was a 100 Mb/s Ethernet.

4.1 The Detection of TCP Syn Flooding Attacks

In the attack known as TCP SYN Flooding a host is flooded with TCP connection request segments with their flag SYN set, but an unreachable source IP address. The host then replies with a segment in which flags SYN and ACK are set. The three-way handshake is never completed, and the host will not receive any reply, as the timeout goes off. The number of connection requests may be enough to fill the request queues, leaving the host's services unavailable to process valid requests.

Program Neptune [11] was used in this case study in order to attack a host running an ANEMONA program written to detect such an attack. Neptune generates segments with source and target addresses and ports given by the user.

DOI: 10.1002/nem

Time of the attack	`tcp.tcpAttemptFails`
	(sampled as delta, interval 6 s)
0 s	0
6 s	0
12 s	170
18 s	300
24 s	301
30 s	300

Table 1. Case study 1: detection of a TCP Syn Flooding attack

In order to detect an attack like this, object `tcp.tcpAttemptFails` may be used, which counts how many times TCP changes from state SYN-SENT or SYN-RCVD to the state CLOSED, plus the number of times it changes from state SYN-RCVD to state LISTEN. Since object `tcp.tcpAttemptFails` is a counter, it was sampled as a delta, using intervals of 6 s, by using the Expression MIB configured by our ANEMONA translator.

With the network under normal operation conditions, connections to FTP, HTTP and telnetd services were established.

The first attack was issued against port 23, used by telnet. 5000 segments were generated, and we obtained the delta values shown in Table 1. After 30 s, the delta value of `tcp.tcpAttemptFails` reached a steady state in 300. By the end of this attack, the absolute value of `tcp.tcpAttemptFails` was 4887.

Based on these results, we assigned 25 as the delta value for `tcp.tcpAttemptFails`, using intervals of 6 s between samples. The following ANEMONA program implements the application above:

```
watch: victim.inf.ufpr.br using private
tcp.tcpAttemptFails.0 is Counter32: delta
begin
  when tcp.tcpAttemptFails.0 > 25
  do
    notify admin.inf.ufpr.br private  tcp.tcpAttemptFails.0
  end
end
```

The program above watches host `victim.inf.ufpr.br` sampling object `tcp.tcpAttemptFails` as a delta. When the delta value from this object is greater than 25, a notification is sent to `admin.inf.ufpr.br`, holding `tcp.tcpAttemptFails`. The actions produced by the translation of this program are available in Denes *et al.* [12]. Considering a number of attacks observed, it took an average of 7 s for a notification to be generated.

4.2 Detection of Link Utilization Upsurge

This experiment consists of the execution of an ANEMONA program that generates a notification when there is a quick, steep increase on the utilization of a link. In order to evaluate this solution, another program that generates a stream of UDP (User Datagram Protocol) segments was employed. This program sends a large number of UDP segments to a given destination process, increasing the utilization of the communication link to that process.

SNMP objects `ip.ipInReceives` and `ip.ipOutRequests` count the number of IP datagrams received by and sent to a given host. The summation of these two values was employed in order to monitor the utilization of a given link. Since objects `ip.ipInReceives` and `ip.ipOutRequests` are counters, they were sampled as delta, using intervals of 6 s. The following ANEMONA program samples these objects as deltas, performs their summation and assigns the result to an instance of the Expression MIB results table, called utilization.

DOI: 10.1002/nem

```
watch: ahost.inf.ufpr.br using private
ip.ipInReceives.0 is Counter32: delta
ip.ipOutRequests.0 is Counter32: delta
begin
  bind utilization to (ip.ipInReceives.0 + ip.ipOutRequests.0);
end
```

The program above was executed in two different situations: under a low network utilization and with a heavy utilization. When the network load was low, representative values of number of datagrams counted (achieved by summing ipInReceives and ipOutRequests, sampled as deltas), registered at intervals of 1 min, ranged from 80 to 88.

In order increase the network utilization, we established a number of FTP connections to the monitored host. Initially one FTP session was established, and then two, three and four sessions.

To each new session, the number of datagrams transmitted through the network was computed three times, at intervals on 1 min; results are shown in Table 2.

After that, the number of UDP datagrams was monitored In the beginning, the stream was monitored in only one way, with the monitored host running the server and the client installed in another computer of the network. Then, the stream was monitored in two ways, with servers and clients running in each one of the hosts used. Table 3 shows representative values for the number of datagrams counted, using intervals of 6 s, considering the stream generator running in one way and also in two ways.

Considering the results in Table 3, we chose 8000 as a critical value for the delta of the number of datagrams. The program below samples `ip.ipInReceives.0` and `ip.ipOutRequests.0` as deltas and performs their summation, assigning the result to an entry of the Expression MIBs table of results, called `utilization`. When the value assigned to `utilization` is greater than the critical value, 8000 in this case, a notification is sent to the specified host with the object instance assigned. In a representative result, after 19 s a notification was delivered to the monitored host.

```
watch: denes.cce.ufpr.br using private
ip.ipInReceives.0 is Counter32: delta
ip.ipOutRequests.0 is Counter32: delta
begin
  bind utilization to (ip.ipInReceives.0 + ip.ipOutRequests.0);
  when utilization > 8000
```

Number of FTP sessions			
1 min interval	2 min interval	3 min interval	
1 session	2055	2926	3003
2 sessions	3052	3244	2762
3 sessions	3686	3318	3837
4 sessions	3796	3569	3230

Table 2. Case study 2: traffic generated by FTP sessions

Interval	One-way	Two-way
0 s	90	86
6 s	16 091	96 463
12 s	55 483	57 489
18 s	81 360	62 784
24 s	65 859	125 463
30 s	78 524	124 944

Table 3. Case study 2: number of IP datagrams counted

DOI: 10.1002/nem

```
  do
    notify admin.inf.ufpr.br private utilization
  end
end
```

The actions generated by the translation of the programs in this paper are available in Denes *et al.* [12], where the reader can check the large number of complex commands required to manually configure the Expression and Event MIBs, which are automatically generated by compiling ANEMONA programs.

5. CONCLUSIONS

ANEMONA is a language for programming network monitoring applications. A compiler was developed that configures a policy repository and the corresponding policy deployment and event monitoring. In the current implementation a compiler was developed that generates code for configuring both the Expression MIB and the Event MIB.

Writing an ANEMONA program is much simpler than configuring a policy repository manually. Two case studies were presented, which generate notifications when a TCP SYN Flooding attack and a steep increase in the utilization of a link are detected.

Future work will include extending the language to monitor QoS (Quality of Service) parameters, and the integrated use with COPS (Common Open Policy Service) [13] and COPS-PR (COPS Usage for Policy Provisioning) [14].

REFERENCES

1. Tsarouchis C, Denazis S, Kitahara C, Vivero J, Salamanca E, Magana E, Galis A, Manas JL, Carlinet L, Mathieu B, Koufopavlou O. A policy-based management architecture for active and programmable networks. *IEEE Network* 2003; **17**(3): 22–28.
2. Nikolakis Y, Magaña E, Solarski M, Tan A, Salamanca E, Serrat J, Brou C, Galis A. A policy-based management architecture for flexible service deployment in active networks. *Lecture Notes in Computer Science* 2004; **2982**: 240–251.
3. Guo X, Yang K, Galis A, Cheng X, Yang B, Liu D. A policy-based network management system for IP VPN. In *Proceedings of the International Conference on Communication Technology (ICCT 2003)*, Beijing, China, 2003.
4. Westerinen A, Schnizlein J, Strassner J, Scherling M, Quinn B, Herzog S, Huynh A, Carlson M, Perry J, Waldbusser S. Terminology for policy-based management. *RFC 3198*, November 2001.
5. Harrington D, Presuhn R, Wijnen B. An architecture for describing SNMP management frameworks. *RFC 2271*, January 1998.
6. Kavasseri R, Stewart B. Distributed management expression MIB. *RFC 2982*, October 2000.
7. Kavasseri R, Stewart B. Event MIB. *RFC 2981*, October 2000.
8. McCloghrie K, Perkins D, Schoenwalder J. Structure of management information version 2 (SMIv2). *RFC 2578*, April 1999.
9. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques and Tools*. Addison-Wesley: Reading, MA, 1986.
10. The NET-SNMP Project Home Page. http://net-snmp.sourceforge.net [14 April 2007].
11. Project Neptune. *Phrack Magazine* http://www.phrack.org/archives/48/p48-13/ [23 May 2007] 1996; **7**(48).
12. Denes H, Musicante M, Duarte EP Jr. ANEMONA code examples. *Technical Report 001/2007*, UFPR/DInfo. http://www.inf.ufpr.br/info/techrep/ [14 April 2007].
13. Walker J, Kulkarni A (eds). Common open policy service (COPS) over transport layer security (TLS). *RFC 4261*, December 2005.
14. Chan K, Seligson J, Durham D, Gai S, McCloghrie K, Herzog S, Reichmeyer F, Yavatkar R, Smith A, COPS usage for policy provisioning (COPS-PR). *RFC 3084*, March 2001.

DOI: 10.1002/nem