ANEMONA: A language for programming NEtwork MONitoring Applications

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

[1]: *Federal University of Paraná, Dept. Informatics*
P.O. Box 19018 Curitiba 81531-990 PR Brazil
e-mail: {mam,elias}@inf.ufpr.br

[2]: *State University of Santa Catarina, Dept. Computer Science*
P. O. Box 631 Joinville 89223-100 SC Brazil
e-mail: denes@joinville.udesc.br

Abstract

We present ANEMONA: a language for distributed network management programming. The compilation of an ANEMONA program generates code for configuring DisMan MIB's thus automating a complex task that is prone to errors if done manually. 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 the Expression MIB and the Event MIB. Experimental results are presented, including an ANEMONA program that detects TCP Syn Flooding attacks, and a program for detecting steep variations on the utilization of monitored links.

Keywords: Network Management, SNMP, Distributed Management, Programming Languages.

1 Introduction

As current network management systems are responsible for monitoring and controlling increasingly large and complex networks and systems, the distributed management paradigm [1] has been seen as the architecture of choice for dealing with the new challenges and requirements. A distributed network management system 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 [3].

The Internet standard network management architecture, SNMPv3 (Simple Network Management Protocol version 3) defines management entities that fit a distributed management paradigm [2]. Entities may behave as managers, agents, or proxies. An entity keeps a MIB (Management Information Base) that defines both an interface to the data available at the entity, and the behavior of that entity.

The IETF (Internet Engineering Task Force) DISMAN (DIStributed MANagement) Working Group has defined a set of MIB's that allow entities of a distributed system to share the tasks of monitoring and controlling the managed resources [3].

Among other standards, DISMAN has proposed both the Expression MIB [4], which allows the definition and evaluation of expressions built from readings of managed objects; and the Event MIB [5], that allows a set of objects to be monitored, and associated conditions to be defined; an event is triggered upon the occurrence of a condition, emitting alarms or executing a defined procedure.

This work describes the ANEMONA (A NEtwork MONitoring Application) language. ANEMONA is a simple language that allows the definition of expressions of managed objects to be monitored as well

*On leave at Université de Tours - LI/Antenne de Blois, France. Partly supported by CAPES (Brazil) BEX 1851/02-0.

as the definition of conditions related to those objects that when detected cause the generation of alarms or the execution of pre-defined procedures.

A compiler was developed for the language, that generates code that configures both the Expression MIB and Event MIB. The manual configuration of these MIB's is a hard task that requires the manager to learn details of the internal details of those MIB's. By using ANEMONA the manager can easily use the MIB's 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 gives a brief description of both the Expression MIB and the Event MIB. Section 3 describes the ANEMONA language, as well as the proposed compiler. Section 4 describes case studies, including an ANEMOMA program that detects TCP Syn Flooding attacks and another program for determining a steep increase on the utilization of a monitored link.

2 The Expression MIB & The Event MIB

The compilation of an ANEMONA program generates commands for the configuration of the Expression MIB and the Event MIB. This section gives an overview of these MIB's, which are IETF standards proposed by the Distributed Management (DisMan) Working Group [3].

2.1 The Expression MIB

The Expression MIB [4] 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 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.

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 [4].

2.2 The Event MIB

The Event MIB [5] 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 the 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 [5]. 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.

3 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 the 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 [9].

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 class of directives are explained in the following subsections.

3.1 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`, `Unsigned`, `TimeTicks` and `IpAddress`. Arguments to the arithmetic operators can be of mixed types. Conflicts are solved using the following rules, listed in order of precedence:

1. The unary subtraction "`-`" always returns a value of type `Integer32`.
2. If both operands of a binary operator have the same type, then the result is of that type.
3. If one operand of a binary expression is of type `Counter64`, then the result is of type `Counter64`.

[1] Reserved words and program syntax are written with `this typeface`, while nonterminals are written ⟨ *like this* ⟩.

4. If one operand of a binary expression is of type `IpAddress`, then the result is of type `IpAddress`.

5. If one operand of a binary expression is of type `TimeTicks`, then the result is of type `TimeTicks`.

6. If one operand of a binary expression is of type `Counter32`, then the result is of type `Counter32`.

7. In all other cases, the result of the operation is 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 [8].

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 parenthesis.

3.2 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.

3.3 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* ⟩

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.

3.4 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:

1. Programming the expression within the *Expression MIB*,
2. For each command of the list, use the *Event MIB* to:
   (*i*) Configure a trigger to monitor the result of the expression and
   (*ii*) Configure an event for each action to be taken.

3.5 Implementation Issues

The ANEMONA system includes a compiler and a run-time system. The compiler was constructed in C [8], using standard techniques [10]. The run-time system is composed by (subset) implementation if the *Event* and *Expression MIBs*. The exact extent of these implementations can be found in [7].

The ANEMONA configuration language was devised to assist in the distributed object management using the *Expression* and *Event MIBs*. ANEMONA is a front-end tool: it takes a higher-level description of the management information and generates a set of operations basically, *snmpget* and *snppset* operations to configure the *Expression* and *Event MIBs*, described in the following section.

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 as TCP SYN Flooding attack is detected. The second program detects steep variations on 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 computers running Linux, NET-SNMP [11] 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 10Mbps 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 won't 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.

Time of the attack	`tcp.tcpAttemptFails` (sampled as delta, interval 6s)
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.

Program Neptune [6] 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.

In order to detect an attack like this, object `tcp.tcpAttemptFails` may be used that 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 six seconds, 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 got the delta values shown in table 1. After 30 seconds, the delta value of `tcp.tcpAttemptFails` reached the steady state in 300. By the end of this attack, the absolute value of `tcp.tcpAttemptFails` was 4887.

Based on these results, we chose as critic value 25 to the delta value of `tcp.tcpAttemptFails`, using intervals of 6 seconds 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 [7]. Considering a number of attacks observed, it took an average of 7 seconds for a notification to be generated.

4.2 Link Overload Detection

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

Number of FTP sessions	1 minute interval interval	2 minutes interval	3 minutes 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-ways
0 seconds	90	86
6 seconds	16091	96463
12 seconds	55483	57489
18 seconds	81360	62784
24 seconds	65859	125463
30 seconds	78524	124944

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

were sampled as delta, using intervals of 6 seconds. The following ANEMONA program samples these objects as deltas, performs their summation and assigns the result to an instance of the Expression MIBs results table, called `utilization`.

```
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
```

This 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 in intervals of 1 minute, 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, in intervals on 1 minute, results are shown in in the 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 seconds, considering the stream generator running in one way and also in two ways.

Considering the results in table 3, we chose 8000 as a critic value for the delta of the number of datagrams. The program below samples `ip.ipInReceives.0` and `ip.ipOutRequests.0` as deltas, perform 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 critic value, 8000 in this case, a notification is sent to the specified host with the object instance assigned. In a representative result, after 19 seconds a notification was delivered in 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
  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 [7], where the reader can check the large number of complex commands required to manually configure the Expression and Event MIB's, which are automatically generated by compiling ANEMONA programs.

5 Conclusions

ANEMONA is a language for programming distributed management applications. A compiler was developed that generates code for configuring both the Expression MIB and the Event MIB, which are IETF standards proposed by the Distributed Management Working Group. Writing an ANEMONA program is much simpler than configuring these MIB's manually. Two case studies were presented, which generate notifications when a TCP SYN Flooding attack and a steep increase on the utilization of a link are detected.

Future work includes extending the language to configure other DisMan MIB's such as the Notification MIB [12].

References

[1] W.Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2,* Addison-Wesley, Reading, MA, 1999.

[2] D.Harrington, R.Presuhn, and B.Wijnen, "An Architecture for Describing SNMP Management Frameworks," *Request for Comments 2271*, January, 1998.

[3] *IETF Distributed Management Working Group (DisMan),* `http://www.ietf.org/html.charters/disman-charter.html`, 2003.

[4] R.Kavasseri, and B.Stewart, "Distributed Management Expression MIB," *Request for Comments 2982*, October 2000.

[5] R.Kavasseri, and B.Stewart, "Event MIB," *Request for Comments 2981*, October 2000.

[6] *Project Neptune, Phrack Magazine,* Vol. 7, No. 48, July 1996.

[7] H.D.H.Fernandes, E.P.Duarte Jr, and M.A.Musicante, "ANEMONA: Uma linguagem de Configuração para Aplicações Práticas de Gerência Distribuída," (*in Portuguese*) *Proceedings of the 20th SBC Brazilian Symposium on Computer Networks (SBRC'2002)*, Búzios, RJ, Brazil, 2002.

[8] B.W.Kernighan, and D.M.Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

[9] K.McCloghrie, D.Perkins, and J.Schoenwalder, "Structure of Management Information Version 2 (SMIv2)," *Request for Comments 2578*, April 1999.

[10] A.V.Aho, R.Sethi, and J.D.Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[11] *The NET-SNMP Project Home Page,* `http://net-snmp.sourceforge.net`

[12] R.Kavasseri, "Notification Log MIB," *Request for Comments 3014*, November 2000.