Independent active program representation using ASN.1

Brad Williamson
Australian Telecommunications Research Institute
Curtin University of Technology
and
Craig Farrell
NDG Software Inc.

The future success of computer communications will largely depend on how effectively applications achieve their desired quality of service (QoS). Active networks move closer to the goal of application specified QoS by allowing user-specified network related computation to be *injected* into the network elements. Although research into active networks is in its infancy, one area that has not yet received much attention is the representation of active programs. The Active Network Encapsulation Protocol (ANEP) has made a significant effort towards producing a common definition, however, as the name suggests, ANEP has only gone as far as the packet header for encapsulating an active program. The contribution of this paper is to show how an independent packet representation such as Abstract Syntax Notation One (ASN.1) can be used as a common substrate for representing active programs. An active networking framework, using ASN.1 whereby any active network solution can be deployed independently of the active network element's architecture is discussed. In this paper, we demonstrate the independent representation using a simple procedural language. This technique however is language independent and can be extended to cover any language including all those currently used in active networks. ASN.1 is ideally suited to this purpose since it was originally conceived as a language for the representation of programs and data types in protocol data units (PDUs). This solution maintains security, mobility and efficiency.

1. INTRODUCTION

When the ARPANET was conceived and deployed in 1969, it was designed with the philosophy that multiple services be provided and distributed across a heterogeneous environment [1]. The *internet*, which was modelled on the ARPANET framework, is now hosting a new generation of applications which require QoS. Although most would concede that QoS over the Internet Protocol (IP) is an illusion, we are seeing the deployment of many

Name: Brad Williamson
Address: GPO BOX U 1987, Perth, Western Australia 6845

Name: Craig Farrell
Address: Suite 200, 12680 High Bluff Drive, San Diego USA 92130

protocols to help realise this illusion.

For a protocol to become ubiquitous, it needs to be standardised. The Internet Engineering Task Force (IETF), amongst others, has recognised this and has in place a *standards track* process. Unfortunately, this process takes time. Alexander et al. [2] observe that the time taken for a protocol to be conceived, developed, presented at SIGCOMM, standardised and then implemented can take between five and eight years. Despite the fact that this delay acts as a *filter* which prevents poorly designed protocols gaining acceptance, the standardisation process may see the protocol become outdated or further revised. Active networks provide a mechanism whereby new services can be deployed *on the fly*, hence removing this standardisation hurdle. Unlike conventional protocols, active networks consider *the user* an important factor.

Active networks view the network as a general computation engine. Data is treated as a program that is executed as it traverses the network. This has led to a new area of research which investigates how networks can perform computations on data which passes through the elements [3]. An active network is an abstraction which allows user-specified actions to be performed within the network switches [3; 4; 5]. Although active networks allow a plethora of applications, they are ideally suited to network management. Active networks promise an increase in the complexity and functionality of network functions that can be offered by the network. This reduces the need for end node network related computation.

Many researchers have offered definitions and descriptions (some of which are described in the next section) of what an active network architecture actually is. Although research into active networks is still relatively new, the current trends emerging from this research indicate that the delays experienced in the *standardisation* of conventional protocols will re-emerge. To minimise the impact of this problem, it is desirable to solve active networking issues using methods that are independent of any active network architecture. It is envisaged that a common representation as proposed in this paper will accelerate the development of these solutions.

The Active Network Encapsulation Protocol (ANEP) [6] has made a significant effort towards defining a common representation for active programs. ANEP, as the name suggests, defines a packet format for encapsulating an active program. ANEP uses a type, length and value (TLV) style encoding, similar to that of Basic Encoding Rules (BER), defined in X.209 [7]. The primary ANEP header specifies the type of active network architecture used, the options and the payload (which is an active program). The options field (also a TLV encoded field) stores information about the program identification.

ANEP only goes as far as defining the active program header. The payload is arbitrarily defined by the developer of the active network architecture. Although the payload can be distingushed by the type identifier field, interoperability remains a challenge. If the vision of a wide area active network is to be realised, it will be beneficial to have a standard method for representing and encoding active programs. The use of the type identifier field implies that there will be many methods of representing an active program, and it follows that network elements may only handle a subset of these representations leading to all the usual problems of interoperability between competing active network architectures.

In this paper, we shall draw upon existing knowledge in PDU specification (e.g. ASN.1 [8]) and encoding schemes (e.g. BER [7], External Data Representation (XDR) [9] and Network Byte Ordering (NBO)) to show how active programs can achieve a high level of interoperability. Any ASN.1 specification can be encoded into a byte code using any of the aforementioned encoding schemes [10]. Throughout this paper, we shall refer to an ASN.1

encoded specification as the *common byte code*. However, this byte code could be an NBO byte code, an XDR byte code or a BER (or any of its derivatives) byte code.

The framework for the deployment of active networks described in this paper uses an ASN.1 value definition, which is used to generate a byte code, to represent active programs. This gives all active network elements the ability to interpret a common definition, even if a primitive structure (e.g. `OCTET STRING`) which represents a byte code is used, meaning that any active network architecture can be deployed. Existing (or future) active network architectures only require a method to produce an ASN.1 value definition, from their programs. It follows therefore that active programs written in any language are translated into an ASN.1 definition. It has been shown [10] that an ASN.1 definition can be encoded using any encoding scheme. Given the ASN.1 definition, any active network element can execute the program irrespective of the active network architecture that produced the PDUs since an active network element needs only to be able to interpret the common byte code.

The process of moving from a high level language to ASN.1 is demonstrated in this paper using a simple procedural language. Despite the simplicity of this language (although some researchers suggest that active network elements should remain simple) it is sufficient to show that *any* language can be mapped to an ASN.1 value definition. The remainder of this paper looks at existing active network architecutres. Given this, we examine the issues which must be considered for a ubiquitous active network. Furthermore, we shall argue that the new framework provides a consistent and coherent approach to active networking without sacrificing security, mobility and efficiency. A solution is discussed whereby ASN.1 is used to represent an active program from a user program.

2. BACKGROUND

Initial work [3; 5; 4] has argued that active networks are characterized by one of two approaches: discrete and integrated. The discrete approach looks at how programs can be uploaded to a programmable node and then executed. The integrated approach, also called the *capsule* approach, treats every packet as a program. In this approach, the network element contains a general purpose processor which executes the data (a program), modifies a packet (if need be) and forwards the packet onto the next host. Thus, if a store and forward network is to be used as an active network, the network will become store, *compute* and forward. Tennenhouse [3] believes that this approach will allow fine grain instructions to be implemented at strategic points in the network. Furthermore, the capsule approach moves away from fixed packet formats meaning that there will be an inherent increase in complexity.

The Netscript project [11] follows the discrete approach to active networking. It allows programs to be delegated to a Virtual Network Engine (VNE) (an agent) in a similar way to that used to upload a program to a host. Netscript can be applied to a variety of network management applications which include Simple Network Management Protocol (SNMP) agents and signalling. Unlike other architectures, Netscript looks at programming the nodes rather than having every packet contain a program.

The Smart Packets project [12], currently under development at BBN, looks at how programs written in the *sprocket* language can be encoded within a single IPv4 or IPv6 datagram. The philosophy behind the Smart Packets project is to allow customised diagnostic and configuration of networks and networked hosts. Smart Packets follow the *capsule* approach whereby a *spanner*, a stack based CISC type language, program which is compiled from a *sprocket* program is encapsulated within an ANEP PDU. At each network element exists an ANEP daemon which acts as an injection and reception point thus providing store, *compute* and forward functionality. In addition, smart packets allow the integration of normal data packets, whereby the router IP alert function is used to identify ANEP PDUs. Schwartz et al. [12] argue that Smart Packets is a *clean* and *safe* environment whilst providing some interoperability via MIBs, a feature that has been neglected by most active architectures.

The Switchware project [13; 14; 2] seeks to investigate the rapid development and deployment of new network services. Switchware provides a facility for a basic network service which is programmable so that services can be selected, either on a per-user, or even a per-packet basis. Initally, services are deployed using the CaML language [15] which is used in the Switchware testbed. Recently, the Programming Language for an Active Network (PLAN) [16] was developed which provides a script-like functional language at the user level. These programs are encoded into an ANEP packet. Switchware was applied to a host, acting as a bridge, which allows the *hot swapping* between bridging protocols [2].

Georgia Institute of Technology (GaTech) [17; 18; 19] propose an architecture for active networking which uses the `IPOPT_AP` experimental option field, defined by Bhattacharjee et al. [17], of an IP PDU to indicate and encapsulate functions. The functions call a pre-loaded routine, which, in turn, is executed on an active node. The pre-loaded routine can be extended to provide a *turing complete* function. Alexander et al. [2] argue that this architecture is insecure as no method exists to validate loaded modules.

The `IPOPT_AP` field has also been applied to programmable congestion control, in particular, MPEG group of packet (GOP) discard. Bhattacharjee et al. [17; 19] discuss how active networks can look for triggers that indicate that congestion has occurred and thus take corrective action.

Research at Massachusetts Institute of Technology (MIT) [3; 20; 4; 21] have seen the development of a capsule and programmable switch based active network architecture. This active network research has seen significant development in the application of active networks to various network problems [22; 23]. The result of this is a toolkit, called the Active Network Toolkit (ANTS) [21] which allows the development and deployment of active programs.

MIT's capsule approach to active networks had focused on using the option field in the IPv4/IPv6 header to store a code fragment [4] (similar to the approach at GaTech). Wetherall et al. [4] demonstrated a prototype through the use of a subset of Tcl how simple programs, for example, the active *traceroute*, can be encoded within an IP PDU and executed. Research efforts now focus on ANTS [21] which is an active network architecture for building and deploying network protocols. Unlike previous architectures, ANTS allows active protocols to be deployed at the end systems as well as mid points of a network.

Current prototypes of ANTS use the Java [24] Virtual Machine. Programs are written in Java and transferred (over the user datagram protocol (UDP)) to the active host. To demonstrate the effectiveness of ANTS, several examples were used one of which was the multicast negative acknowledgement (NACK) suppression problem [21; 20].

So far, we have examined various active network architectures. In the next section, we consider the issue of whether active networks should have a common representation.

3. OPEN ISSUES

As mentioned in Section 1, the main motivation behind the establishment of active networks is that user-driven code, related to the network, can be injected into the network elements. Each of the architectures described in Section 2 uses a different method to represent programs. Despite the fact that [3] has classified active networks into two approaches, there is no standard language for each type of active program and there is no way to distinguish between an integrated or discrete program. Consider the following program:

```
if dest-addr <> host-addr
    append host ip to pdu
else
    bounce pdu back to source host
endif
```

This program is a simple active *traceroute*. It can be treated as a discrete program which is executed on each packet passing through a host, or this program can be treated as a capsule which is executed at every hop. If an architecture that supports both discrete and integrated approaches is used, then a program syntax that distinguishes between both methods is required.

There is no common method for encoding an active program. We have seen in the previous section two methods for encoding an active program. If the program was written for Switchware, then it would be encoded in CaML byte code. Alternatively, if ANTS is used then a Java byte code is used. MIT's Active IP and GaTech's `IPOPT_AP` architectures each use different encoding schemes on top of Network Byte Ordering (NBO). ANEP, on the other hand, uses an encoding scheme that resembles BER.

Undoubtedly, the proliferation of active network technologies will see the number of architectures increase, as will the number of encoding schemes. If the vision of an active network is realised and a wide area active network is to be deployed then:

(1) Programs need to be encoded using an encoding scheme that is independent of the underlying network layer substrate. An active network architecture independent of the network layer substrate means that issues such as fragmentation, packet loss, discard algorithms etc. need not be considered.

(2) Interoperability between architectures needs to be maintained. If multiple active network architectures exist then the issue of interoperability must be addressed.

(3) Active processing within the network elements must be given the same priority as routing/switching and thus it should not be treated as a secondary function. The effect of this has been demonstrated by the fact that remote monitoring (RMON) agents, which are embedded in routers, tend to drop packets when the traffic load is high [25]. Some reviewers have observed that in advocating the active network paradigm, an additional layer (which interprets or executes a program) may degrade performance. This is correct, however this addition is mandatory if the active network philosophy is to be fulfilled. It follows that network elements have to maintain a high level of efficiency meaning the active network language must be simple. This also implies that routers must not have multiple active architectures.

Interoperability issues with non-active architectures have seldom been discussed in active network texts. An example of such an interoperability issue is interfacing an active network architecture to a network management information base (MIB). With the exception of Netscript [11] and Smart Packets [12], the other architectures do not communicate with a MIB.

All of the architectures presented in the previous section exhibit both language and encoding scheme dependencies. To achieve ubiquitous active networking, network elements only need to be able to understand a common program format. For all of the discussed architectures, multiple representations are required by the network elements irrespective of the underlying architecture.

In the next section, we describe a new framework which is independent of any encoding scheme or language and can represent all of the existing active network architectures. Our framework is designed to integrate easily with other network management entities (such as SNMP agents and MIBs) as well as fully exploiting the concept of active networking.

4. THE SOLUTION

The active network framework proposed is demonstrated using a simple language. The program is converted to an ASN.1 [8] value definition which is encoded using BER [7]. Using the capsule approach to active networking, these programs (encapsulated inside a datagram) will traverse a route and the program will be executed at every intermediate node supporting this language. The packet is decoded from BER back to a value definition so the contents (program) can be interpreted.

It was decided to use ASN.1 and BER as the specification and encoding schemes since:

(1) ASN.1 [8] is already a standardised specification. ASN.1 has been used in a variety of network applications such as the Message Handling System framework [26] and the Intelligent Network framework [27]. Furthermore, ASN.1 is already used to specify network management PDUs, for example SNMP [28] and Common Management Information Protocol (CMIP) [29]. This is an important consideration since it will facilitate the integration of existing network management functions into an active network.

(2) In the context of an active network, ASN.1 inherently provides the ability to specify an active program which acts like a protocol specification. Steedman [30] claims that ASN.1 is reminiscent of a high level programming language. It is argued that the protocol designer can describe the program without the need for encoding rules.
By having an active program act as a PDU specification, the packet as a whole can be combined and treated as one active PDU specification. Other architectures, mentioned previously, view the packet payload as being isolated from the headers. This is an important feature when using the capsule approach and an active program is required to modify the packet header as it traverses the network. The isolated nature of the payload suggests that extra functionality is required to map fields in the header to the program. However, if a combined specification was used, then mapping fields becomes *natural*.

(3) Debugging an active program would be a lot easier using a standardised specification. Because we treat a program as a packet specification, packet analysers that understand BER can decode the program.

(4) A more fault tolerant solution is provided. As we will show, if a capsule needs to maintain state then local memory is required. We can represent local memory as an ASN.1 specification. If the active router is shut down, and a backup router is active then all active memory can be transferred to the next host.

(5) ASN.1 offers the ability to provide definition security using the `ENCRYPTED` data type. BER can also be encrypted using the Rivest, Shamir and Adleman (RSA) algorithm, or alternatively Canonical Encoding Rules (CER)[1] can be used as the encoding scheme. Tantiprasut et al. [31; 10] have also shown that ASN.1 specifications can be encoded in any encoding scheme meaning our framework is encoding scheme independent. This is important since it was observed, in Section 2, that several encoding schemes have been proposed by the various research groups.

5. REPRESENTATION

In this section, we outline a simple active network language that is converted to an ASN.1 specification. The active language proposed is a simple procedural language (with only one procedure – the mainline) and is provided as a *proof of concept* for our work. For reasons of code security and speed, we decided to omit advanced language features (e.g. recursion and multiple functions), however this language remains functionally complete. This language is specified in ASCII text, parsed and converted to an ASN.1 specification. Once a valid ASN.1 specification has been defined, it is encoded using a standard encoding scheme for transmission.

For all of the other architectures described in Section 2, either they need to map their language into an ASN.1 value definition or need to build a compiler which translates their language directly into an ASN.1 value definition. ASN.1 has the ability to map any binary sequence onto any value definition which includes an `OCTET STRING`, as it will be the case for binary data, compiled programs or byte code. It is noted that the use of an `OCTET STRING` will require a higher level definition because the contents are opaque to the network element. However, for the remainder of this paper, our language is an ASN.1 defined active language.

In this paper, we use a simple programming language, however any other language can be mapped to an ASN.1 definition. Tantiprasut et al. [31; 10] describe how any PDU definition can be mapped to an ASN.1 definition using any arbitrary encoding scheme. This can be extended to bytecode (encapsulated within a PDU) mapping, thus it follows that any language that can be represented as a bytecode can be used in our framework. That is, our framework is language independent. This is important since we observed, in Section 2, that several different languages have been proposed by the various research groups.

At the highest level, an active program is one that is made up of a code segment and a data segment. The code segment is one that stores instructions to be processed at the active node. The data segment stores any data that needs to be carried with the program. The following fragment of an ASN.1 definition shows how an active program is represented. The code segment is an `INSTRUCTION` and the data segment is a `SEQUENCE OF VARTYPE`.

As shown below, the data and code segments are separated to provide code security and quick data retrieval. Code security is provided whereby `CODESEGMENT` is not changed throughout execution. Furthermore, this separation allows the data segments to be easily loaded at runtime.

```
ACTIVEPROGRAM ::= [APPLICATION 2] SEQUENCE
{
    codeseg     [1] CODESEGMENT,
```

[1] Steedman [30] refers to CER as *Confidential* Encoding Rules.

```
    dataseg     [2] DATASEGMENT
}
```

Data which can be stored in a program can be one of the following data types: `BOOLEAN`, `INTEGER`, `OBJECT IDENTIFIER`, `OCTET STRING`, `SEQUENCE OF BOOLEAN`, `SEQUENCE OF INTEGER`, `SEQUENCE OF REAL`, `SEQUENCE OF OBJECT IDENTIFIER` and `SEQUENCE OF OCTET STRING`.

In our active programming language, they are defined as:

```
VAR
   <variable1> : <type> [DEFAULT value],
   <variable2> : <type> [DEFAULT value],
      .
      .
      .
   <variablen> : <type> [DEFAULT value];
```

Of these data types, `OBJECT IDENTIFIER` is of the most importance. Although we have some active hosts, it is desirable to have some access to a MIB to query variables. The purpose of the `OBJECT IDENTIFIER` is to define a value which tells the active interpreter to substitute the value of the variable binding in its place. For example, the statement `A = [1.3.6.1.2.1.1.4.1];` would tell the active interpreter to retrieve the value of the MIB variable `mib2.interfaces.ttl.1` into variable A.

With the exception of the `SEQUENCE OF` types, a default value can be assigned to the variable. It was intended that the `SEQUENCE OF` variables be used for collecting and storing a log of a particular data type as it traverses the network (e.g. collecting a list of host names). An example declaration of a data segment is:

```
VAR
   HOPCOUNT : INTEGER DEFAULT 0,
   TIME : REAL DEFAULT 0.0,
   MIBVAR : OBJID DEFAULT [1.3.6.1.2.1.1.4.1];
```

This would be mapped to the following ASN.1 value definition:

```
dataseg { -- SEQUENCE --
   integer 0,
   real 0.00000000000000000E+00,
   objectid { 1 3 6 1 2 1 1 4 1 }
}
```

Throughout the program, each variable can be referred to by its name, however when encoded, a numerical identifier is used. The first variable is referred to as variable 1, second variable as variable 2 and so on.

In addition to a data segment, the headers in an active program are encapsulated meaning they can also be used as variables. From an ASN.1 specification, fields in the PDU header can be referred to in the same way as variables in the main program. Given the following IP header (defined in ASN.1) [10]:

```
IpPDU ::= SEQUENCE
{
    version         INTEGER,
    hlen            INTEGER,
    service         INTEGER,
    total-len       INTEGER,
    id              INTEGER,
    flags           BIT STRING,
    offset          BIT STRING,
    ttl             INTEGER,
    protocol        INTEGER,
    checksum        INTEGER,
    srcaddr         INTEGER,
    dstaddr         INTEGER,
    options         OptionType OPTIONAL,
    data            IpDataType
}
```

We can refer to any of these fields in our active program. For example, to swap the source and destination addresses[2] we can do the following:

```
VAR
   TEMP : INTEGER DEFAULT 0;
BEGIN
   TEMP = srcaddr;
   srcaddr = dstaddr;
   dstaddr = TEMP;
END;
```

As mentioned previously, our work focuses on the *capsule* approach to active programs. Should a capsule need to maintain state at an active node, it can do so in one of two ways. Firstly, a program can modify a MIB variable, or alternatively the program can modify a variable which is stored in the active router. These variables are defined in the same way as the data segment.

The code segment is made up of a sequence of `INSTRUCTION` which can be one of: `PROGRAM`, `COMPARISON`, `REPETITION` and `ASSIGNMENT`.

The `ASSIGNMENT` statement in our active programming language is the same as a typical programming language construct whereby a value is assigned to a variable. An assignment is expressed as `variable = expression`. To provide interoperability with network management frameworks, the assignment can also be expressed as `objectid = expression`. For example, the active program (shown below) turns on an RMON ethernet statistics probe (given an index of 6000).

```
BEGIN
  [1.3.6.1.2.1.16.1.1.1.21.6000] = 2;
  [1.3.6.1.2.1.16.1.1.1.21.6000] = 1;
END;
```

This program is represented as the following ASN.1 value definition:

```
decodedprogram ACTIVEPROGRAM ::= { -- SEQUENCE --
  codeseg program { -- SEQUENCE OF --
     assignmentsnmp { -- SEQUENCE --
        variable {1 3 6 1 2 1 16 1 1 1 21 6000},
        expression value integer 2
     },
     assignmentsnmp { -- SEQUENCE --
        variable {1 3 6 1 2 1 16 1 1 1 21 6000},
```

[2]For simplicity, the destination and source addresses is an integer. It has been observed that the address can also be stored as an `OCTET STRING`.

```
        expression value integer 1
     }
  },
  dataseg { -- SEQUENCE OF --
  }
}
```

The defined ASN.1 data type EXPRESSION (as the name suggests) stores an expression which is to be evaluated by the interpreter. EXPRESSION is conceptually an expression tree where expressions can be encapsulated inside other expressions.

The COMPARISON takes on a standard programming language comparison. Given the following construct:

```
IF (condition) THEN
  PROGRAM;
```

It is mapped onto the following ASN.1 type:

```
COMPARISON ::= SEQUENCE
   {
        expression  [1]     EXPRESSION,
        instruction [2]     INSTRUCTION
   }
```

For the purpose of this work, it was decided not to implement the ELSE statement. Although this addition is not difficult, its omission is to keep the active programming language simple.

In the active programming language, the REPETITION statement takes on the form of:

```
WHILE (condition) DO
  PROGRAM;
```

As with any WHILE statement, the encapsulated program is executed repetively until the condition evaluates to FALSE. The WHILE statement is mapped onto the following ASN.1 type definition:

```
ITERATION ::= SEQUENCE
   {
        expression  [1]     EXPRESSION,
        instruction [2]     INSTRUCTION
   }
```

6. IMPLEMENTATION AND RESULTS

6.1 Encoding

All active programs developed under this framework will follow the process as shown in Figure 1. At the source end, the program is parsed and converted into an ASN.1 value definition. Once a definition has been derived, it is converted to a byte code for transmission over the wire. For the purpose of this exercise, we shall use BER [7]. The intermediate nodes then decode this byte code back into an ASN.1 value definition which is executed.

If the data segment in the program is modified, the ASN.1 value definition is re-encoded and routed to the next hop. Upon completion, the variables can be retrieved by decoding the byte code back to the ASN.1 value definition and examining the data segment.

Variables stored in the data segment are persistent. That is, the data segment is not reinitialised as it traverses the execution path. Initialisation of the variables in the data

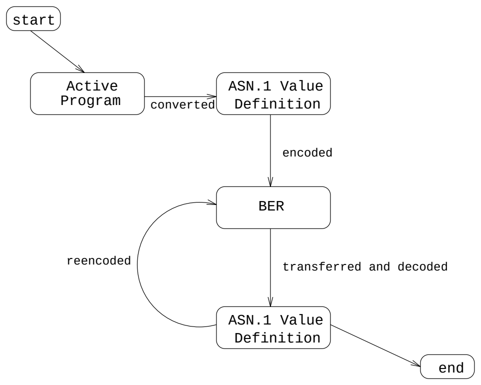

Fig. 1. Active Program Process

segment is set at compile time. The fact that the data segment is persistent means that variables maintain state information from hop to hop. This is demonstrated in the active *traceroute* example shown in the next section.

Initially the original program is parsed and compiled to an ASN.1 value definition.

6.2 Decoding

Active hosts that receive an active packet execute the code. Before execution, the packet must be decoded into the ASN.1 type definition. When a packet arrives at an active node, the active program is separated from the PDU header. The program is passed to a SNACC [32] function where the `ACTIVEPROGRAM` data structure is built. Although we have separated the PDU header, the header must remain available throughout execution.

Following a successful decode, the program is executed. Using the SNACC `ACTIVEPROGRAM` data structure, the interpreter recursively inspects every value and executes an instruction. Throughout execution, the active program may modify variables in the data segment and PDU header.

Upon successful execution, the program is reencoded and reencapsulated ready for forwarding to the next node.

The process of decoding, executing and then reencoding provides some useful features which are not available in other active network architectures. Firstly, the encoding scheme can be different for part of an active route. For example, if we have an active program that is executed on part of two intranets, the active architecture can be configured to transmit a program in BER within each intranet. Active programs for the internet link however can be transferred using CER or over an IPSEC tunnel with encryption, as some reviewers have pointed out. This allows some measure of security to be provided. Another example would be an active program that is transported over different frame types. Suppose a program is transported over IP and Transport Protocol Class 4 (TP4). Active programs that travel over IP may be encoded using NBO and when they travel over TP4, they may be encoded using BER. Despite the fact that these protocols will treat the active program as opaque data, it logically follows that protocol analysis will be simpler as the packet is no longer disjoint. That is, the data is related to the header.

Using a known encoding scheme will aid in the filtering of active programs that use packet headers or known network management data types (e.g. `OBJECT IDENTIFIER`).

For security reasons, not all MIB variables may be changed by an active program. Given an active program which changes a MIB variable, a filter can be applied which masks the variable binding. This could be used to prevent the active program from modifying a particular MIB variable.

Currently, independent encoding is limited in the existing or proposed models. When a program is encoded into its byte form (i.e. Java byte code, CaML byte code), it cannot be changed throughout execution.

6.3 Examples

In this section, we examine various examples of how this active framework can be used in various areas of network management. The first two examples are taken from [4; 16] and are used to relate this work to other active network architectures. Whilst the examples shown here are coded in our simple procedural language, we should iterate that these examples could have been coded in any language and implemented on any of the existing (or proposed) active network architectures. All that is required is a method to translate the code into the generated byte code.

6.3.1 Active Traceroute

traceroute [33] is a common utility which utilises the time to live (TTL) field, in the IP datagram, and the `ICMP_TIME_EXCEEDED` response to determine the mid points of a route. Every time a PDU makes a hop, the TTL field is decremented. When the TTL field equals 0, an `ICMP_TIME_EXCEEDED` response is sent to the sender. Thus, *traceroute* sends out packets, incrementing the TTL field each time (starting at 0) until the route is found. The increasing TTL value means that every router a packet traverses will be discovered.

This example highlights the importance of an active network as the conventional algorithm relies on the source node to gather all the information about a route. Wetherall et al. [4] have provided an example, called an active traceroute, whereby the host address is returned to the original source. The example to follow shows how an active traceroute is implemented, firstly as an active program which is specified by the user and secondly, as a generated ASN.1 value definition. It is this value definition which is translated into a generated byte code (using an encoding scheme) for transmission.

```
VAR
  STATUS : INTEGER DEFAULT 1,
  HOSTS : SEQOCTETSTR,
  TEMP : OCTETSTR DEFAULT " ";
IF (STATUS = 1) THEN
BEGIN

   /* Append IP address to list of hosts, stored in HOSTS */

   HOSTS = HOSTS UNION host-ipaddr;

   /* Check to see if finished. If so, then set a status flag
      so the return path is not traced. */

   IF (host-ipaddr = ip-dest) THEN
   BEGIN
      STATUS = 0;

      /* Swap source and dest packets in the IP header so the
         packet is returned to the sender. */

      TEMP = ip-dest;
      ip-dest = ip-source;
      ip-source = TEMP;
   END;
END;
```

The generated ASN.1 code is as follows:

```
decodedprogram ACTIVEPROGRAM ::= { -- SEQUENCE --
     codeseg comparison { -- SEQUENCE --

             expression brackexp exp { -- SEQUENCE --
                         expression1 value integer 1,
                         evaluator 8,
                         expression2 var 1
                     },
             instruction program { -- SEQUENCE OF --
                     assignment { -- SEQUENCE --
                             variable 2,
                             expression exp { -- SEQUENCE --
                                     expression1 var 2,
                                     evaluator 7,
                                     expression2 var 2000
                                 }
                         },
                     comparison { -- SEQUENCE --
                             expression brackexp exp { -- SEQUENCE --
                                         expression1 var 111,
                                         evaluator 8,
                                         expression2 var 2000
                                     },
                             instruction program { -- SEQUENCE OF --
                                     assignment { -- SEQUENCE --
                                             variable 1,
                                             expression value integer 0
                                         },
                                     assignment { -- SEQUENCE --
                                             variable 3,
                                             expression var 111
                                         },
                                     assignment { -- SEQUENCE --
                                             variable 111,
                                             expression var 110
                                         },
                                     assignment { -- SEQUENCE --
                                             variable 110,
                                             expression var 3
                                         }
                                 }
                         }
                 }
         },
     dataseg { -- SEQUENCE OF --
         integer 1,
         seqoctstr { -- SEQUENCE OF --

             },
         octetstr '20'H  -- " " --
     }
 }
```

This version of the active traceroute provides the same functionality as *traceroute*, however it does not rely upon the Internet Control Message Protocol (ICMP). This program simply appends the host address onto the host sequence in the data segment as it traverses the network from source to destination. It assumes however that the host address is locally obtained.

When this active program reaches the end of the route, the source and destination headers in the IP datagram are swapped, thus the packet is routed back to the original source. To prevent the return route being stored in the data segment, a status variable is set.

Although this program has the same functionality as the active traceroute described in [4], the features highlighted in the previous section are evident here; for instance, the active traceroute can be extended to execute over multiple protocol stacks; for example, if a PDU is carried by ATM, it may be desirable to append the ATM NSAP or E.164 [34] address to the PDU.

A further extension is the addition of security. By changing encoding schemes, security can be improved. For example, by using CER, information about a route can be encrypted. CER [35] is designed to offer more security by enciphering parts of a transmitted encoding. As a consequence, issues in security such as who signs, encrypts and authenticates the packet will need to be considered. This simple change reduces a weakness in conventional *traceroute* whereby a protocol analyser could be used to passively watch *traceroute* at work in order to determine a route.

The active traceroute exhibits better fault tolerance. Given that the end system assumes a "good" route already exists, if a route is severed, it logically follows that the example can be extended to include diagnostic code which is returned (by querying the MIB) explaining why/when the route went down. Moreover, the program can be made to wait or discover an alternative route.

6.3.2 Active Ping

ping is another common UNIX utility which uses `ICMP_ECHO_REQUEST` and `ICMP_ECHO_RESPONSE` PDUs to determine if a host is alive. *ping* works by sending out an `ICMP_ECHO_REQUEST` to the destination host. If the destination host receives this request, then an `ICMP_ECHO_RESPONSE` is returned.

As with the active traceroute, active ping is a well known example used in active networks [4; 16]. Following is a representation of the active ping in its original source form.

```
VAR
  STATUS : INTEGER DEFAULT 1,
  TEMP : OCTETSTR DEFAULT " ";
IF (STATUS = 1) THEN
BEGIN

   /* If the packet has reached the destination then swap the
      IP address so the packet can be re-routed to the original
      source. */

   IF (host-ipaddr = ip-dest) THEN
   BEGIN

      /* The status is set to 0 so that when the packet returns
         the ping does not re-iterate. */

      STATUS = 0;
      TEMP = ip-dest;
      ip-dest = ip-source;
      ip-source = TEMP;
   END;
END;
```

Active ping follows the same structure as the active traceroute whereby the source and destination addresses are swapped when it reaches the destination. Like the active traceroute, security and fault tolerance are improved using our scheme.

The ability to move *ping* into the network demonstrates the concept of an active network. Given that a user specified program is in the network, we can easily extend this to provide user specific diagnostic functions without having to rely upon ICMP. For example, if the active ping program cannot be forwarded, the program can query the local MIB to retrieve the status of the next hop.

6.3.3 Other Examples

In this section, we consider some other examples, one of which demonstrates the interoperability between an active network and existing network management architectures.

The Path Maximum Transmission Unit (MTU) discovery technique [36] exploits the don't fragment (DF) bit in the IP datagram and the `ICMP_DEST_UNREACH` PDU to discover (by estimation) the minimum transmission unit for an arbitrarily defined internet path. Mogul and Deering [36] argue that PDUs sent out with a size bigger than the path MTU are non-optimal as the PDUs are prone to fragmentation.

Discovering the path MTU involves repeatedly sending out an IP datagram, of an arbitrarily defined size, with the DF bit set. If the MTU is large, an `ICMP_DEST_UNREACH` PDU with a *fragmentation needed and DF set* code is returned to the source. The MTU is decreased until a datagram can be delivered to the destination.

Another method of discovering a better approximation of the path MTU is to query a MIB on each intermediate router. Assuming that the MIB exists, the variable `mib2.interfaces.ifmtu.1` (where `1` can be substituted for the interface number) can be queried. By querying the MIBs along a route, a better approximation of the MTU can be found. Determining the mid points of a route will require the routing table to be queried or the execution of a *traceroute*. This increases the complexity in discovering a better MTU.

```
VAR
  MTU : INTEGER DEFAULT 32767,
  STATUS : INTEGER DEFAULT 1,
  TEMP : OCTETSTR DEFAULT " ";
IF (STATUS = 1) THEN
BEGIN

   /* If we are at the end of the path then return packet to
      sender by swapping the source and destination addresses. */

   IF (host-ipaddr = ip-dest) THEN
   BEGIN
      STATUS = 0;
      TEMP = ip-dest;
      ip-dest = ip-source;
      ip-source = TEMP;
   END;

   /* Query the MIB and if the MTU is less than the stored MTU
      then update the stored MTU. */

   IF ([1.3.6.1.2.1.2.1.4.1] < MTU) THEN
      MTU = [1.3.6.1.2.1.2.1.4.1];
END;
```

By combining the MIB query and active networks, a better and quicker approximation of the path MTU can be discovered. The program (shown in its native form) can query the `ifmtu` MIB variable as it traverses the network, in much the same way as *traceroute*. If the MTU is less than the MTU variable in the data segment then it is replaced with the new variable.

The previous three examples show how our active network framework can be used in the IP stack. We will now show that our proposed active framework can be used in ATM by looking at how active networks can be used for congestion control in the Available Bit Rate (ABR) service. Using active networks in a switched architecture such as ATM was first proposed in [17; 37].

The ABR service, defined by the ATM Forum [38] and the ITU-T [39] as a service which uses leftover bandwidth for sources with unpredictable traffic requirements, such as data services. When a call is being established, the source negotiates, amongst other things, an initial cell rate (ICR) and a peak cell rate (PCR). When the call is connected, data can be transferred at a rate (the allowed cell rate (ACR)) between the ICR and PCR. Ideally, the connection would like to maintain the PCR, however if congestion occurs, the connection is reduced to an explicit rate (ER)[3] which is less than the ACR. The ER is the point that allows the source to transmit whilst utilising the full remaining bandwidth. Additionally, the ER is calculated in a way such that the other ABR sources maintain a fair share of the link.

Determining the ideal ER is a problem that has been investigated by many researchers [40; 41; 42; 43; 44; 45; 46; 47]. All of the proposed algorithms are “hard wired” into the switch meaning that if two switches have different algorithms then interoperability problems may exist. Utilising an active network architecture suggests that it becomes possible for the source to “program” the switches along the path with the desired ER algorithm.

```
VAR
```

[3]The desired transfer rate.

```
  MCR : INTEGER  DEFAULT 365566,
  ER : INTEGER DEFAULT 365566,
  FRM : BOOLEAN DEFAULT TRUE;
IF (FRM = FALSE) THEN
  IF (switch-macr < ER) THEN
     ER = switch-macr;
```

The algorithm is encoded using the previous techniques into resource management (RM) cells and replaces the RM cell format, defined in [38; 39], which only stores the parameters required for the "hard wired" algorithm.

This active ER algorithm provides a user-driven congestion control implementation. It is no longer reliant on any specific frame types, or predefined ER algorithms.

7. SUMMARY

Many researchers have investigated and concluded that active networks provide a flexible platform for the deployment of user-driven services which are related to the network.

Most of the frameworks that have been studied look at how active programs are used and the applications where active programs can be applied. Despite this, there has been little research into how active programs are represented. The ANEP [6] has gone as far as defining an active program header, however the encoding scheme is left up to the active network developers. Smart Packets [12] from BBN is concerned with how *sprocket* programs can be encoded within IPv4 and IPv6 PDUs. Switchware [13; 14] uses a different byte code representation to encode its CaML program. GaTech and MIT have built models which utilise variations on the IP option fields to specify, encode and transport their active programs. It is noted that GaTech and MIT use Java in their current models. For this reason, an independent active program representation is required.

In this paper, we have presented an active framework whereby the program is encoded using ASN.1. As mentioned previously, ASN.1 can be used to represent a PDU with an arbitrary encoding scheme. Thus, the program representation in ASN.1 provides a *natural* extension to the PDU header; providing a more coherent solution.

Choosing ASN.1 as the transfer syntax removes the restriction on the encoding scheme that is used. Unlike other proposed active network architectures, active programs are not restricted to a single encoding scheme. The proposed active infrastructure is independent of hardware and provides security as well as a coherent encapsulation; thus the PDU header and data are not disjoint.

Tennenhouse [3] believes that security, mobility and efficiency are the key to an active network. The ASN.1 specification provides code security as the proposed language is small and instructions cannot be changed on the fly i.e. the program is restricted to changing the data segment and packet header. ASN.1 provides access security as encoding rules can be changed at different hops making the active program less vulnerable to access violations.

Using this framework does not impede the mobility of implemented active network architectures. The framework described in this document currently applies to the capsule approach. It logically follows that our framework can be extended to the programmable switch (discrete) approach by including instructions which "force" an active program to stay at a node. It thus follows that an extra data type (e.g. `PACKET`) would be required to interpret the field.

The proposed framework maintains efficiency. If the capsule approach is used, the programs should remain small and have the same computational complexity. The efficiency of the programmable switch will depend on the algorithms deployed.

An open question which remains as a result of this research is the encoding size of active programs. This is important as active network elements must remain efficient. Although the examples presented here use BER, other TLV encoding schemes can be used which may be more efficient. We would point out that the size of the encoding is merely a function of the encoding scheme. TLV based encoding schemes can produce encodings of similar size to NBO when generality clauses are relaxed [48]. Packed Encoding Rules (PER) [49; 30] is an example of TLV based encoding which representations that are of similar size to NBO.

All of the architectures presented in Section 2 exhibit both language and encoding scheme dependencies. The philosophy behind active networks was that any host can deploy user driven services into the network elements. Ubiquitous active networking cannot be achieved if a number of different architectures and languages are used. One of the prime motivations for our model is that these dependencies are removed. In our model, a variety of active network architectures can be implemented in *any* language and then be mapped onto a common PDU specification. Many active network implementations already send compiled programs (stored as a byte code) to be executed at the active network element. The variety of data structures that exist in ASN.1 provides a high degree of flexibility meaning that any language can be represented in ASN.1. Bytecodes and executables can be represented as an `OCTET STRING`. The router only needs to interpret the ASN.1 byte code for execution. In the case of a program represented as an `OCTET STRING`, the byte code is extracted and copied to the relevant virtual machine for execution. An active network architecture only needs to be translated into our ASN.1 value definition for an active network element to interpret it. The active network framework we propose can support all of the currently proposed active network architectures.

8. CONCLUSION

We believe that we have described a useful framework for specifying, building, deploying and executing active programs using the capsule approach. The architecture provides security, mobility and efficiency for active programs while at the same time being language and encoding scheme independent.

The proposed architecture will support other active architectures. Existing architectures are translated into an ASN.1 value definition which is translated into a (generated) byte code supported by the active network elements. These elements interpret the byte code; they require no understanding of the active network architecture which generated the code. Thus, by deploying the ASN.1 byte code into routers, interoperability issues between active network models, architectures, languages and encoding schemes can be avoided. Our framework means that researchers will be able to free themselves from the distractions of programming language preferences and encoding scheme complexities. Researchers can now focus on important active network issues such as comparing different active network architectures for speed, flexibility and functionality. It is envisaged that this work will encourage researchers to embrace the challenge of finding problems which are clearly better solved (or uniquely solved) using the active network paradigm.

In conclusion, we note that ASN.1 was originally conceived as an abstraction for representing programs and data types in PDUs. This fits *exactly* with the philosophy behind the active network paradigm.

9. ACKNOWLEDGEMENTS

This work is supported by the Cooperative Research Centre for Broadband Telecommunications and Networking (CRC-BTN). The authors would like to thank Elizabeth Murphy, Donald Reid and the anonymous reviewers for proofreading and feedback.

REFERENCES

[1] David D Clark, "The design philosophy of the DARPA internet protocols," *Computer Communication Review*, vol. 18, no. 4, pp. 106–114, Aug. 1988.

[2] D Scott Alexander, Marianne Shaw, Scott M Nettles, and Jonathan M Smith, "Active bridging," in *ACM SIGCOMM*, 1997.

[3] David L Tennenhouse and David J Wetherall, "Towards an active network architecture," *Computer Communication Review*, vol. 26, no. 2, Apr. 1996.

[4] David J Wetherall and David L Tennenhouse, "The ACTIVE IP option," in *7th ACM SIGOPS European Workshop, Connemara, Ireland*, Sept. 1996.

[5] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[6] D Scott Alexander, Bob Braden, Carl A Gunter, Alden W Jackson, Angelos D Keromytis, Gary J Minden, and David Wetherall, "Active network encapsulation protocol (ANEP)," Rfc draft, University of Pennsylvania – Computer and Information Science Department, July 1997.

[7] ITU-T, "Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1)," Recommendation X.209, International Telecommunication Union, Aug. 1989.

[8] ITU-T, "Specification of Abstract Syntax Notation One (ASN.1)," Recommendation X.208, International Telecommunication Union, Aug. 1989.

[9] Sun Microsystems, Inc, "RFC 1014: XDR: External Data Representation Standard," June 1987.

[10] Duke Tantiprasut, John Neil, and Craig Farrell, "ASN.1 protocol specification for use with arbitrary encoding schemes," *IEEE/ACM Transactions on Networking*, vol. 5, no. 4, pp. 502–513, Aug. 1997.

[11] Yechiam Yemini and Sushil da Silva, "Towards programmable networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1996.

[12] Beverly Schwartz, Wanyi Zhou, Alden W Jackson, W Timothy Strayer, Dennis Rockwell, and Craig Partridge, "Smart packets for active networks," in *IEEE Openarch*, 1999.

[13] D Scott Alexander, William A Arbaugh, Michael W Hicks, Pankaj Kakkar, Angelos D Keromytis, Jonathan T Moore, Carl A Gunter, Scott M Nettles, and Jonathan M Smith, "The switchWare active network architecture," *IEEE Network*, vol. 12, no. 3, pp. 29–36, May 1998.

[14] J M Smith, D J Farber, C A Gunter, S M Nettles, Mark E Segal, W D Sincoskie, D C Feldmeier, and D Scott Alexander, "Switchware: Towards a 21st century network architecture," Tech. Rep., University of Pennsylvania – CIS Department, 1997.

[15] Xavier Leroy, "The Caml special light system," Computer Software, Nov. 1995.

[16] Michael Hicks, Jonathan Moore, D Scott Alexander, Carl Gunter, and Scott Nettles, "PLANet: An active internetwork," in *IEEE Infocom*, 1999.

[17] Samrat Bhattacharjee, Kenneth L Calvert, and Ellen W Zegura, "An architecture for active networking," Tech. Rep. GIT-CC-96/20, Georgia Institute of Technology - College of Computing, 1996.

[18] Samrat Bhattacharjee, Kenneth L Calvert, and Ellen W Zegura, "On active networking and congestion," Tech. Rep. GIT-CC-96/02, Georgia Institute of Technology - College of Computing, 1996.

[19] Samrat Bhattacharjee, Kenneth L Calvert, and Ellen W Zegura, "An architecture for active networking," in *High Performance Networking (HPN)*, Apr. 1997.

[20] Ulana Legedza, David J Wetherall, and John Guttag, "Improving the performance of distributed applications using active networks," in *IEEE Infocom*, 1998.

[21] David J Wetherall, John V Guttag, and David L Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," in *IEEE Openarch*, 1998.

[22] David M Murphy, "Building an active node on the internet," M.S. thesis, Laboratory for Computer Science – Massachusetts Institute of Technology, May 1997.

[23] Van C Van, “A defense against address spoofing using active networks,” M.S. thesis, Massachusetts Institute of Technology – Department of Electrical Engineering and Computer Science, May 1997.

[24] Sun Microsystems, “The java specification language,” White paper, Sun Microsystems, 1995.

[25] Mustapa Wangsaatmadja, Windiaprana Ramelan, Bradley Williamson, and Craig Farrell, “Remote packet capture using RMON,” in *AUUG/APWWW Winter Conference*, 1996.

[26] ITU-T, “Message handling services: Message handling system and service overview,” Recommendation X.400, International Telecommunication Union, Apr. 1993.

[27] ITU-T, “Q-series intelligent network Recommendation structure,” Recommendation Q.1200, International Telecommunication Union, Nov. 1993.

[28] M. Schoffstall, M. Fedor, J. Davin, and J. Case, “RFC 1157: A Simple Network Management Protocol (SNMP),” May 1990, See also STD15 [50]. Updates RFC1098 [51].

[29] ITU-T, “Common management information protocol specification for CCITT applications,” Recommendation X.711, International Telecommunication Union, Aug. 1991.

[30] Douglas Steedman, *ASN 1: The Tutorial and Reference*, Technology Appraisals, 1990.

[31] Duke Tantiprasut, “Protocol analysis via ASN.1 specification,” Honours Thesis, Dec. 1995.

[32] Michael Sample, “Snacc 1.1: A high performance ASN.1 to C/C++ compiler,” Computer Software, July 1993.

[33] Van Jacobson, *traceroute(8)*, Feb. 1989, Manual Page.

[34] ITU-T, “Numbering plan for the ISDN era,” Recommendation E.164, International Telecommunication Union, Nov. 1991.

[35] ITU-T, “Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER),” Recommendation X.690, International Telecommunication Union, June 1994.

[36] J. Mogul and S. Deering, “RFC 1191: Path MTU discovery,” Nov. 1990.

[37] Samrat Bhattacharjee, Kenneth L Calvert, and Ellen W Zegura, “Implementation of an active networking architecture,” White Paper, July 1996.

[38] The ATM Forum, “Traffic management specification version 4.0,” Tech. Rep. af-tm-0056.000, The ATM Forum, Apr. 1996.

[39] ITU-T, “Traffic control and congestion control in B-ISDN,” Recommendation I.371, International Telecommunication Union, Nov. 1993.

[40] Raj Jain, Shivkumar Kalyanaraman, Sonia Fahmy, Rohit Goyal, and Seong-Cheol Kim, “Source behavior for ATM ABR traffic management: An explanation,” *IEEE Communications Magazine*, vol. 34, no. 11, pp. 50–57, Nov. 1996.

[41] Raj Jain, Shivkumar Kalyanaraman, Rohit Goyal, Sonia Fahmy, and Ram Viswanathan, “The ERICA switch algorithm for ABR traffic management in ATM networks, part I: Description,” *IEEE/ACM Transactions on Networking*, 1997.

[42] Raj Jain, Sonia Fahmy, Shivkumar Kalyanaraman, and Rohit Goyal, “The ERICA switch algorithm for ABR traffic management in ATM networks, part II: Requirements and performance evaluation,” *IEEE/ACM Transactions on Networking*, 1997.

[43] Yoon Chang, Nada Golmie, and David Su, “A rate based flow control switch design for ABR service in an ATM network,” in *Twelfth International Conference on Computer Communication*, 1995.

[44] Y Chang, N Golmie, and David Su, “Study of interoperability between EFCI and ER switch mechanisms for ABR traffic in an ATM network,” in *Fourth International Conference on Computer Communications and Networks*, 1995.

[45] Yan Moret and Serge Fdida, “ERAQLES an efficient explicit rate algorithm for ABR,” in *IEEE Globecom*, 1997.

[46] Yan Moret and Serge Fdida, “Design and analysis of an ABR explicit rate algorithm: ERAQLES,” *IEEE/ACM Transactions on Networking*, 1997.

[47] The ATM Forum, “Enhanced PRCA (proportional rate-control algorithm),” Tech. Rep. 94-0735R1, The ATM Forum, Aug. 1994.

[48] Nilotpal Mitra, “Efficient encoding rules for ASN.1-Based protocols,” *AT&T Technical Journal*, vol. 73, no. 3, pp. 80–93, May 1994.

[49] ITU-T, “Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER),” Recommendation X.691, International Telecommunication Union, July 1995.

[50] J. Case, M. Fedor, M. Schoffstall, and J. Davin, “STD 15: Simple Network Management Protocol,” May 1990, See also RFC1157 [28].

[51] J. Case, C. Davin, and M. Fedor, “RFC 1098: Simple Network Management Protocol SNMP,” Apr. 1989, Obsoletes RFC1067 [52]. Updated by RFC1157 [28].

[52] J. Case, M. Fedor, M. Schoffstall, and J. Davin, “RFC 1067: Simple Network Management Protocol,” Aug. 1988, Obsoleted by RFC1098 [51].