

Management of network devices from OPNET Modeler simulation environment

Milan Bartl, Jiri Hosek, Karol Molnar and Lukas Rucka

Abstract—The Simple Network Management Protocol is the main protocol for monitoring and configuring network devices, but it is not the only way of utilizing this protocol. With the help of this tool, additional information about network devices can be obtained. This information can be used, for example, in the Quality of Service support system that is described in this paper. In order to evaluate this system we used the OPNET Modeler simulation environment. The usage of simulation tools within the development of modern network technologies and protocols is very frequent. But sometimes the pure simulation environment is not quite sufficient and we need to combine it with real systems. Therefore we took advantage of one feature of OPNET Modeler and created a simulation scenario that enables interaction of the OPNET Modeler with real network components. For this purpose we implemented the SNMP protocol into the OPNET Modeler and used it for acquisition of management information stored in the Management Information Base database at the network device. The principles of the communication model developed, including the description of separate components of the model, are introduced in the following text.

Index Terms—BER, Esys interface, MIB, OPNET Modeler, SNMP.

I. INTRODUCTION

THE Simple Network Management Protocol (SNMP) is one of the most often used solutions for remote network management. SNMP was introduced in 1988 [1] to meet the growing need for a standard for remote management of network devices. SNMP was developed as a temporary tool and was intended to be replaced by a solution based on the Common Management Information Service/Protocol (CMIS/CMIP) architecture. Today SNMP is still the most popular method of network management because it is easy to implement and features with great interoperability [2].

SNMP is based on the well-known communication platform between the SNMP agent and the SNMP manager. The agent is located at the managed device and makes accessible the configuration information and statistics of this devices to manager stations. The SNMP manager is placed on the device

that manages the network nodes. The manager sends requests, encapsulated into the SNMP operations, to the agent. As a response the agent usually sends back the requested data. In the case of critical events the agent can inform the manager without any previous polling, using a special trap message [2].

The remote administration of network components is not the only possible utilization of the SNMP protocol. Our QoS support system introduces an alternative utilization of this network protocol. We use SNMP for the acquisition of configuration information of the Differentiated Services (DiffServ) technology from the network devices (see Fig. 1). As a consequence, the information obtained allows networking applications to exactly define their demands on network resources and to select the most suitable service class for their data. This solution could provide the right communication between the user application and the edge components of the DiffServ domain, which could significantly increase the efficiency of this QoS technology [3].

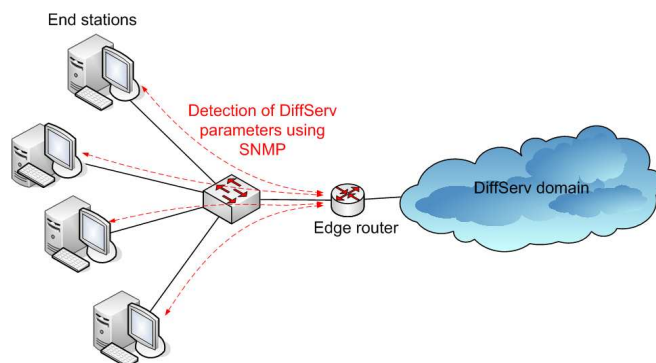


Fig. 1. Utilization of SNMP for collecting DiffServ parameters

For the analysis of the designed system we chose the OPNET Modeler simulation environment (OM). OM is based on an object oriented modelling approach and uses a discrete-event simulation system. Particular components in OM are described by C/C++ source codes and are user accessible [4]. It allows users to modify the source code and to add new custom functionalities if required. A unique feature of OM and the main reason why we decided for this simulation environment is the possibility of an interaction between OM and external systems. This possibility is ensured by a complex set of functions called External System Domain/Definition (ESD) [3], [4]. The external system might represent almost anything from a general algorithm to a specific model of a

Manuscript received May 30, 2010.

M. Bartl, J. Hosek, K. Molnar and L. Rucka are with the Department of Telecommunications of the Faculty of Electrical Engineering and Communication at the Brno University of Technology, Purkynova 118, 612 00 Brno, Czech Republic.

Email: xbartl02@stud.feec.vutbr.cz¹, hosek@feec.vutbr.cz², molnar@feec.vutbr.cz³, rucka.lukas@phd.feec.vutbr.cz⁴).

hardware entity (e.g. user-designed hardware, real network devices). It gives a flexible platform to test new ideas and solutions at low cost and brings more accurate results to the entire research process.

II. IMPLEMENTATION OF CO-SIMULATION PROCESS IN OM

A. Communication between OPNET Modeler and Real Systems

There are two possibilities how to interconnect OM with real network equipment. The first possibility is the already mentioned ESD system. The second is called System In The Loop (SITL). Naturally, both of them have their advantages and disadvantages.

SITL is a separately distributed library for OM which provides an interface to link real network hardware or software applications to the OPNET discrete event simulation. External devices are connected to the simulation loop over SITL gateways operated as a bridge interface between the simulation environment and the network interface of the host computer. Packets transmitted between the simulated and real networks are converted between real and simulation formats. The SITL module is mainly focused on real-time communication with devices based on the Ethernet technology while the use of ESD system is much more versatile [4].

As mentioned before, the ESD system allows an interconnection between OM and an external system. This interconnection is called co-simulation [4]. The ESD system consists of several components. These components are Simulation description, External System Definitions model, External System Interface, co-simulation code, and code implemented in an external system or application. A detailed description of the ESD system can be found, for example, in [5].

The co-simulation is established by using a special interface of OM called External System Interface (esys). The esys interface is represented in OM by a process module. Thus it can be implemented anywhere in the model structure. The esys interface ensures the control and data exchange between OM and external systems [4]. The whole ESD system is more complex and more universal than SITL but, on the other hand, it does not provide such a high speed of the communication and simulation. It is because the control of simulation is shared between OM and an external system, which can lead to slower event processing.

B. Configuration of esys interface

The snmp_manager_esys node model

In the node model of the SNMP manager, derived from a standard workstation model, we implemented a new process model, called *esys*. This process model enables communication with real-network components outside the OPNET Modeler and is connected to the User Datagram Protocol (UDP) process model, because it needs to work with IP addresses and UDP port numbers. The extended structure of the workstation node model is shown in Fig. 2.

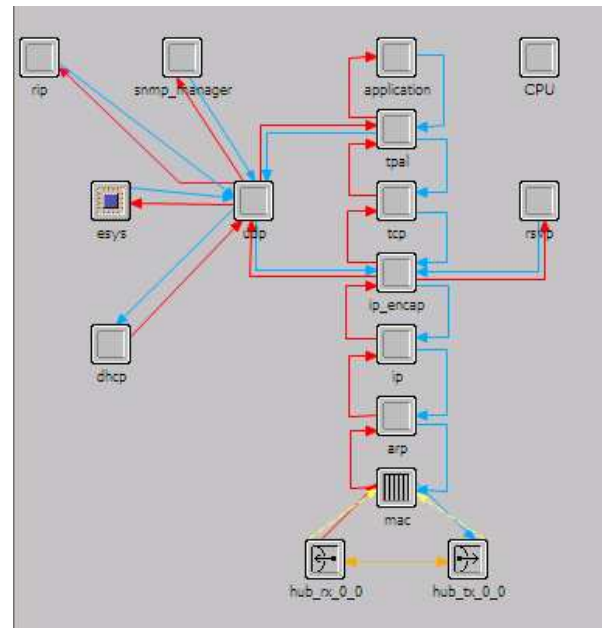


Fig. 2. Structure of the SNMP manager node model

Attributes of the *esys* process model have one special item called External System Definition (ESD) Model. This model is accessible through a button labelled Edit ESD Model in the Attributes window (see Fig. 3).

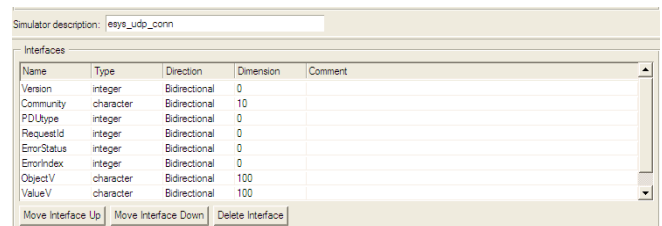


Fig. 3. Configuration of ESD model

The first part contains one row with a text box to enter the name of a special data file called Simulator description. This file will be described below.

The second part defines the esys interface/s connected to this esys process model. These interfaces can transmit data from the simulation to an external application or in opposite direction.

The *Name* field specifies the name through which the interface is accessible.

The following field specifies the data type of the interface. These types can be common C/C++ data types (integer, character, pointer, etc.) or special types of string or bit (for special Value Vector data types in OPNET).

In the *Direction* field the direction of the data exchange is set. There are three possibilities: from OPNET to external code, external code to OPNET or bidirectionally.

The last important field is called *Dimension*. It specifies the dimension of the array for storing the values. If it is set to zero, only one value can be set on this interface. Setting another value would overwrite the previous one.

The *snmp_manager_esys* process model

In the header block of the *snmp_manager_esys* process model, we need to define the conditions of state transitions, the header files attached and the structure of our internal SNMP packets. Next, the process model can be built. It contains four states shown in Fig. 4.

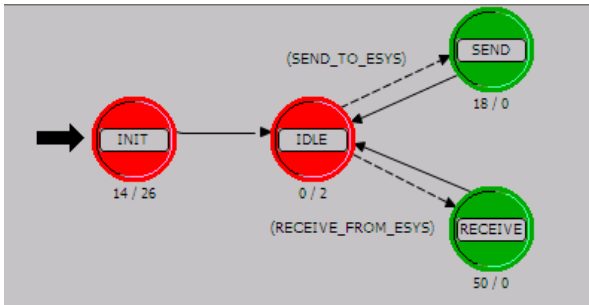


Fig. 4. Process model *snmp_man_esys*

INIT state

Its function is to obtain pointers to objects interconnected with the process (like the UDP process model) and to the process itself. Then it creates an interruption for itself to register the UDP port. It registers the port number 162, reserved by default for SNMP. At the end the memory for error messages and for the IP address string is allocated.

IDLE state

After the initialization the process moves into the IDLE state. It remains here until an interruption is received from outside. The only operation executed here is in the Exit executives, where the pointer to the ICI interface that produced the interruption is obtained. ICI interfaces are commonly used to store additional information about the interruption.

SEND state

Whenever data are received from the UDP process model, i.e. data to be sent out from the simulation, the process transits into this state. In conjunction with the data to be sent the IP address of the target SNMP agent is also obtained. Then, the child process is invoked and the data are written to the esys interfaces. The necessity of using the child process will be explained in the following chapter.

RECEIVE State

As a complement to the SEND state, it reads data from the esys interfaces and stores it in the pre-allocated structure. The structure is then wrapped into a packet and the packet is sent to the UDP process model.

C. Child Process

The process of co-simulation behaves in the following way: Whenever data are written on the esys interface with the `OPC_ESYS_NOTIFY_IMMEDIATELY` condition set, an immediate interruption is invoked in the external code, i.e. at the other side of the esys interface. When this happens, the process at the OPNET side that invoked the interruption will

remain in the send state, because the simulation in OPNET is interrupted [1]. However, we need the main *esys* process to be running because the incoming data, received from the external code, must be processed.

For this purpose we can create a new process that will send the data on behalf of the main process. First, in the child process/es must be declared in parent process model. This declaration is accessible under the File – Declare Child Process Models menu item, where a list of all process models saved in active model directories is shown. The list of active model directories can be configured by adding/removing directories using the File - Manage Model Files menu item.

The Child process contains only two states, called START and EXEC. Immediately after the invocation, the process transits into the EXEC state, which contains the whole source code.

First, we need to obtain the structure containing data from the parent process (SNMP data and IP address). Then we can write these data on the esys interface. With the last value written we create an interruption for the external code. At the end the memory containing SNMP data is de-allocated. This last step is performed after the co-simulation returns from the external code.

D. Simulation Descriptor

This file contains information for the co-simulation builder and linker. The structure of this file is strictly defined. All definitions must be wrapped in the block starting with *start_definition* and ending with the *end_definition*.

```
start_definition
platform:    windows
use_esa_main: yes
kernel:      development
bitness:     32bit
dll_lib:     esys_udp_conn.dll
end_definition
```

We run the co-simulation under the 32-bit Microsoft Windows operating system. We use the OPNET Debugger Console for debugging the simulation, so we need a development kernel.

In our case, the OPNET side should be “in charge”. Setting *use_esa_main* to *yes* means that we use the external dynamic loaded library (DLL file) in the OPNET simulation. The co-simulation is started from OPNET GUI as a usual simulation.

If *use_esa_main* is set to *yes*, we need to define the name of the DLL file with the external code. This name is the parameter of the *dll_lib* item.

III. IMPLEMENTATION OF SNMP PROTOCOL INTO OM

The SNMP protocol is not implicitly implemented in the current version of the OPNET Modeler 15.0, but OM enables the functional modelling of proprietary network technologies and protocols. To be able to create an SNMP message within the model of SNMP manager described above and send the message through the esys interface to a real network component, we implemented the SNMP protocol into the

OPNET Modeler environment. The SNMP protocol is written in C programming language. To meet our requirements we defined the SNMPv2 message structure [2] in the following way:

```

/* Variable Bindings*/
typedef struct variablebin{
    char *ObjectID;
    char *Value;
} VARIABLEBIN;

/* SNMP PDU */
typedef struct snmppdu{
    int PDUtype;
    int RequestId;
    int ErrorStatus;
    int ErrorIndex;
    VARIABLEBIN *VariableBin;
} SNMPPDU;

/* SNMP Packet */
typedef struct snmppacket{
    int Version;
    char *Community;
    SNMPPDU *SNMPPdu;
} SNMPPACKET;

```

Each SNMP message is created dynamically. First, the SNMP operation type (*PDUtype*) is defined and then the message is filled with the OID (*ObjectID*) or specific MIB value (*Value*) according to the operation type.

In order to transmit the SNMP message as described in RFC 1067 [1], it was necessary to encode and decode every message, using the Basic Encoding Rules (BER). BER is one of the encoding formats defined as part of ASN.1 (Abstract Syntax Notation One) and specified by ITU (International Telecommunication Union) in the X.690 recommendation [6]. A single instance of SNMP message has to be encoded into a string of octets. BER defines how the objects are encoded and decoded so that they can be transmitted over a transport network [7].

Since BER is not available in the current version of OPNET Modeler, we also had to implement it. The corresponding functions, programmed in language C, perform the encoding and decoding of every field of the SNMP message, more exactly the data type identifier, the length description in bytes and the encapsulated data. In this way every SNMP message is encoded into an octet sequence. Subsequently, this sequence is transmitted over the communication link to the destination node, where it is decoded back into the original SNMP message. We implemented the following BER-related functions:

BER encoding

- `int SaveIntToBuff(char *buff, int i)` – it calculates the number of bytes needed to express the numerical value.
- `int SaveBERInteger(char *buff, int i)` – it converts a numerical value into a Type-Length-Value structure.

- `int SaveBERString(char *buff, char *str)` – it converts the input string into a Type-Length-Value structure.
- `int SaveBEROID(char *buff, char *oid)` – it converts the OID into a Type-Length-Value structure.
- `int SaveBERVarBind(char *buff, VARBIN *vb)` – it returns the number of bytes needed to express the OID and its corresponding value.
- `int SaveBERVarBindList(char *buff, VARBIN *vb)` – it returns the number of bytes needed to express the *VarBind* item.
- `int SaveBERSNMPDU(char *buff, snmppdu *pdu)` – it returns the number of bytes needed to express the *RequestId*, *ErrorStatus*, *ErrorIndex* and *VarBindList* items.
- `int SaveBERSNMPMessage(char *buff, snmppacket *packet)` – it returns the number of bytes needed to express the *SNMP version*, *SNMP community string* and *SNMP PDU* items.

BER decoding

- `int GetBerInteger(char *buff, int *i)` – it reads the numerical value from the *buff* memory, returns the memory size and saves the numerical value into variable *i*.
- `int GetBERString(char *buff, char **str)` – it reads the string from the memory, returns the number of bytes needed and saves the string into variable ***str*.
- `int GetBEROID(char *buff, char **oid)` – it reads the OID value from the memory, returns the number of bytes needed and saves the OID into variable ***oid*.
- `int GetBerVarBind(char *buff, VARIABLEBIN *vb)` – it reads the *VarBind* item from the memory, and returns the number of bytes needed to express the OID and its corresponding value.
- `int GetBerVarBindList(char *buff, VARIABLEBIN *vb)` – it returns the number of bytes needed to express the *VarBind* item.
- `int GetBerSNMPDU(char *buff, snmppdu* pdu)` – it returns the number of bytes needed to express the *RequestId*, *ErrorStatus*, *ErrorIndex* and *VarBindList* items.
- `int GetBerSNMPMessage(char *buff, snmppacket **packet)` – it returns the number of bytes needed to express the whole structure of the SNMP message.

IV. EXTERNAL APPLICATION

In order to implement a communication process between OPNET Modeler and a real network node we created an external application that is used as middleware between the OM model and the network interface of the local workstation. The application reads data from OPNET Modeler, translates them into SNMP requests and sends them to a network node. Then it waits for the SNMP response from an agent

implemented in the destination node. When the SNMP response arrives, it is translated back into a data structure compatible with OPNET and is entered into the simulation model.

We used the functions of OPNET API described in the previous chapter for sending and receiving data to and from OM. These functions are declared in the *esa.h* header file, which also contains the functions for initializing the co-simulation and registering the callback functions.

The external application developed is compiled as a dynamically loaded library. All the functions that should be available to the co-simulation coordinator have to be exported with the "C" DLLEXPORT code extern. There are two functions of this type: *esa_main* and *callback*.

The *esa_main* function completes the basic initialization procedures of the co-simulation and includes the Winsock library initialization (for communication through the network interface) and registration of the callback function to the esys interface.

The *callback* function is called whenever data are sent out through this interface [8]. The data are read from the interface, encapsulated into an SNMP request and sent to the SNMP agent. After obtaining the response, the corresponding data are converted and forwarded to the simulation model through the esys interface.

For sending the SNMP request there is an internal function (without an export) called *snmp*. This function realizes the SNMP communication, using the functions and structures from the *snmp.h* and *Mgmtapi.h* header files of the Windows OS [9]. It can create an SNMP Get-Request and Get-Next-Request message to obtain values from the MIB database of the SNMP agent. When the response from the agent is received, data are converted to a structure readable by the OM simulation model.

First, it is necessary to call the *SnmMgrOpen* function. It takes four arguments and returns a special variable containing a pointer to a newly created session:

```
LPSNMP_MGR_SESSION SnmMgrOpen(
    __in LPSTR lpAgentAddress,
    __in LPSTR lpAgentCommunity,
    __in INT nTimeOut,
    __in INT nRetries
);
```

lpAgentAddress is a string containing the IP address of the target device operating the SNMP agent. *lpAgentCommunity* contains a string that is used in the SNMP communication as a password. *nTimeOut* is an integer that specifies the communications time-out in milliseconds. The last argument, *nRetries*, specifies the number of attempts to establish the communication.

The next step is to create the *SnmVarBindList* structure, which is used to contain the list of I/O data structures. The definition of this structure is the following:

```
typedef struct {
    SnmVarBind *list;
    UINT len;
} SnmVarBindList;
```

I/O data structures that are members of *SnmVarBindList* are called *SnmVarBind* and their definition is:

```
typedef struct {
    AsnObjectName name;
    AsnObjectSyntax value;
} SnmVarBind;
```

Both structure types must be allocated by a special function called *SnmUtilMemAlloc*. This function works similar to the common C/C++ allocation function *malloc*.

After creating the structures, the *SnmVarBind* must be explicitly filled with input data. Then, the length value of is increased from zero to one to have the first item of *list* (with index zero) accessible. The *name* item should be filled with an ID corresponding to the demanded object ID. This ID must be of a special data type called *AsnObjectName* and can be created from string pointer (LPSTR data type) using the *SnmMgrStrToOid* function.

```
BOOL SnmMgrStrToOid(
    __in LPSTR string,
    __out AsnObjectIdentifier *oid
);
```

The last item that must be specified before sending the Get/Get-Next SNMP request is the *list.value.AsnType*. The value should be set to *ASN_NULL*.

When the structure is filled with data, the request can be sent via the *SnmMgrRequest* function. If this call succeeds, the *SnmVarBind* structure is filled by the SNMP agent with values assigned to the requested objects. These data are stored in the *list.value* item and have the following format:

```
typedef struct {
    BYTE asnType;
    union {
        AsnInteger32 number;
        AsnUnsigned32 unsigned32;
        AsnCounter64 counter64;
        AsnOctetString string;
        AsnBits bits;
        AsnObjectIdentifier object;
        AsnSequence sequence;
        AsnIPAddress address;
        AsnCounter32 counter;
        AsnGauge32 gauge;
        AsnTimeticks ticks;
        AsnOpaque arbitrary;
    } asnValue;
} AsnAny;
```

These data types are incompatible with the OPNET Modeler, so a conversion is necessary. Converted values are saved in a special structure, *SNMPRET*.

```
typedef struct
{
    char* oid;
    int type;
    char* string;
    int integer;
    unsigned int uinteger;
    long linteger;
    ULONGLONG ulinteger;
} SNMPRET;
```

This structure contains a constant used to specify the value type and the ID of the returned object (saved as a string). The conversion of complex data types is executed by the *readAsnOid*, *readAsnAddress* and *readAsnString* functions. These functions take the object as an argument and return a standard string containing the converted value.

After processing the data obtained, the corresponding structures are de-allocated by calling the *SnmUtilMemFree* function. Finally, the *SnmMgrClose* function ends the SNMP manager session (opened by *SnmMgrOpen*) and returns the pointer to the *SNMPRET* structure.

V. SIMULATION SCENARIO

In real network conditions the SNMP manager creates the SNMP message and sends it toward to the SNMP agent. By default, the SNMP message is delivered through the network to the SNMP agent, using the User Datagram Protocol (UDP) transport protocol. Our evaluation test-bed consisted of a real network represented by a C1841 Cisco router and a simulation model in the OPNET Modeler environment. Because of this combination of real and simulated systems, this solution had a more complex architecture than the majority of real infrastructures but, at the same time, it offered several interesting research opportunities. The whole evaluation test-bed has been composed of the SNMP manager created in OM, the esys interface, an external application, and the SNMP agent implemented in a real router with SNMP support. The architecture of the test-bed is shown in Fig. 5.

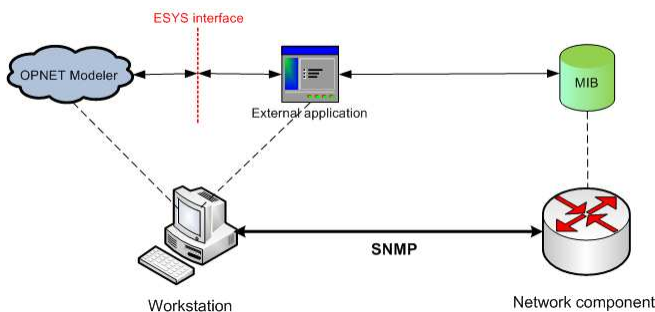


Fig. 5. Architecture of the evaluation test-bed

The communication process of our simulation scenario is described in the following text. The SNMP manager creates the SNMP message in OM and sends it like a common packet to the esys interface in its node model. The SNMP message is passed through the esys interface to the external application. At this moment the simulation in OM is interrupted and the control of the co-simulation is passed to the external application. After the transmission through the esys, interface data conversion must be provided. This conversion is necessary because OM and the external application use incompatible data types. The external application opens a new session with the SNMP agent, places the converted data into an SNMP message and sends it to the agent in a UDP datagram. The SNMP agent is operated by a real hardware

device. After processing the request, the SNMP agent sends the answer in an SNMP response message back to the external application. The external application extracts data from the response and forwards them to the esys interface. In the esys interface, the data are converted to the OM format and sent back to the simulation environment. The simulation in OM is resumed and the control of the co-simulation is passed back to OM. In OPNET Modeler, the data received from the esys interface are wrapped into the OPNET packet and sent to the process model of the SNMP manager for further processing.

VI. CONCLUSION

We have created a communication system capable of mutually exchanging information between real network components and the OPNET Modeler simulation environment. This system can create an SNMP message inside the simulation environment of OM, encode it using the BER algorithm, and transmit it to a real network device. The destination network device can search for and read the required value from the MIB database and send it back to OM. The interconnection of real and simulating environments opens the way towards complex simulation scenarios that can be used, for example, within the development of complex communication systems for network management or quality-of-service assurance.

ACKNOWLEDGMENT

This paper has been supported by the Grant Agency of the Czech Republic (Grant No. GA102/09/1130) and the Ministry of Education of the Czech Republic (Project No. MSM0021630513).

REFERENCES

- [1] Case, J., Fedor, M., Davin, J., *Simple Network Management Protocol*, RFC1067, 1988.
- [2] Mauro, D., Schmidt, K., *Essentials SNMP, Second Edition*, O'Reilly Media, 2005.
- [3] Hosek, J., Rucka, L., Molnar, K., DiffServ Extension Allowing User Applications to Effect QoS Control. *Proceedings of the 13th WSEAS International Conference on Communications*, 2009, pp. 39-43.
- [4] Opnet Technologies, *OPNET Modeler Product Documentation Release 15.0*, 2009.
- [5] Hosek, J., Rucka, L., Molnar, K., Mutual Cooperation of external application and OPNET Modeler simulation environment. *Proceedings of the International Workshop RTT 2009 Research in Telecommunication Technology*, 2009, pp. 1-5.
- [6] *ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, ITU-T Standard X.690, 2002.
- [7] Larmouth, J., *ASN.1 Complete*, Academic Press, 1990.
- [8] Opnet Technologies, Using Modeler's Co-simulation Capabilities to Integrate with External Systems, *Proceedings of the OPNETWORK 2008*, 2008.
- [9] Microsoft Corporation (2009). Simple Network Management Protocol. Microsoft Development Library [Online]. Available at: <http://msdn.microsoft.com/en-us/library/aa377993%28v=VS.85%29.aspx>