Control of Network Operation Generator from OPNET Modeler Environment

Milan Bartl[†], Karol Molnar[††] and Jiri Hosek[†††],

Faculty of Electrical Engineering and Communication, Brno University of Technology, Brno, Czech Republic

Summary
This article concerns interfacing OPNET Modeler simulations with a network operation generator. Data are extracted from the OPNET simulation, and the network operation generator (IxChariot), which generates data flow in real network, is controlled by these data. Creating a middleware application transmitting data from OPNET to IxChariot is described.

Key words:
Esys, IxChariot, traffic generator, OPNET Modeler

1. Introduction

As a part of a project aimed to develop new QoS features, this article presents a middleware to interconnect the OPNET Modeler with the network operation generator IxChariot.

Based on a simulation in the OPNET Modeler, the IxChariot generates data flow with certain parameters. The IxChariot also gathers information about network performance. This allows instant testing of theoretical results in physical envi-ronment while giving an immediate feedback to the simula-tion.

2. OPNET Modeler

2.1 Inside a Simulation

A simulation inside OPNET Modeler processes data that will be sent to the network operation generator. Only a single workstation was placed on the workspace for development purposes. Its attributes contain additional user-defined integer value that represents the output data that will be used for the generator control.

This value is then transported via inner network architecture of the workstation to a special process that writes the data to the External System (ESYS) interface. This interface is used to transfer data inside and outside of simulation [5]. A trace of packets inside workstation's architecture is shown on the picture below.

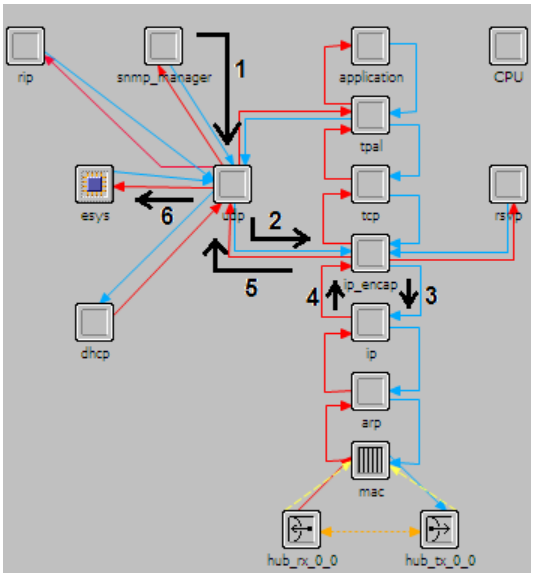


Fig. 1 Packet trace inside workstation.

There is another additional process called snmp_manager that is not a member of default processes of the workstation. This process was developed as a part of another project and its purpose is packet generation (more in [4]).

An integer value was created as one of its attributes and was promoted to global node's attributes. This value will be used to set DSCP and ECN bits [6] in an IP header of packets generated by network operation generator. Its range is from 0 to 255. Meaning of the value will be described later.

During the simulation is the value transported inside packets produced by the process described above.

2.2 Esys Process

Packets received by the esys process are then unpacked and the integer value is extracted. This value is supposed to be sent through the ESYS interface to the outer environment.

Manuscript received July 5, 2011
Manuscript revised July 20, 2011

At this point there is a slight complication however. According to the documentation [3], in the very moment of writing data to the interface a simulation pause occurs, and execution is passed to an external application that processes data incoming from the OPNET Modeler simulation. The process that caused this pause is caught in an unready state, being unable to process any data incoming right after the return of execution to the simulation. If the external application writes data back to the ESYS interface, these data will be lost because of the esys process being unable to process them.
OPNET Modeler offers a possibility to solve this complication by using a so called "child process". It is a process invoked by another existing process. The invoking process can pass any data to this process and let the computation up to it. With this mechanism the *esys* process gets time to return into a "ready" state for processing any incoming data.

3. External application

3.1 Structure

The external application is compiled in a form of a dynamic loaded library (DLL file). It contains three important parts: a main function for initializing procedures, so called callback function for data processing and functions for communication with the IxChariot API. The first and the second function mentioned have to be exported using one of the prefixes shown in Fig. 2. Exporting these functions makes them accessible from the OPNET Modeler core.

```
Prefix from OPNET Modeler library:
extern "C" DLLEXPORT

Standard C prefix:
__declspec( dllexport )
```

Fig. 2 Exporting prefixes.

3.2 Main Function

The main function includes procedures for initialization of co-simulation (simulation within OPNET Modeler that communicates with outer environment). These functions are called *Esa_Init* and *Esa_Load*. Definition of these functions contains a header file *esa.h* that must be included in the external application.

After the initialization, the callback function needs to be attached to a certain ESYS interface. The Fig. 3 shows a function that creates this interface-callback connection.

```
Esa_Interface_Callback_Register
 (esaHandle, &status,
 callback_interface,callback,
 (EsaT_Interface_Array_Callback_Proc)
 NULL, (void *)NULL);
```

Fig. 3 Esa_Interface_Callback_Register.

The *esaHandle* identifier is a pointer to an OPNET Modeler simulation instance. The status variable is used to store a return status of the function. The other two pointers stand for a proper interface and the callback function. The last two arguments are not used.

During this initialization phase, execution of the current co-simulation is on the side of the external application. After the initialization finishes, it is necessary to pass execution to the OPNET Modeler simulation. Calling a function *Esa_Execute_Until* with a time parameter *ESAC_TIME_MAX* passes full control over the co-simulation to the simulation. Basically, the simulation gains control until the time it is sup-posed to finish.

3.3 Callback

This function is attached to a certain ESYS interface. Whenever are data written to this interface, execution is passed to the external application and a code inside the callback function is processed. When the end of the callback function is reached, control is passed back to the simulation.

At first, data from the ESYS interface are read. In this case, data represent an integer value mentioned in the chapter 2. A procedure of reading data from the ESYS interface is shown in Fig. 4. The first function returns a pointer to the required interface. The second function stores data from the interface into a local variable called *dscp*.

```
iface = Esa_Interface_Get(esaHandle,
   "top.office.manager.esys.DSCP");

Esa_Interface_Value_Get(esaHandle,
 &status, iface, &dscp);
```

Fig. 4 Extracting data from ESYS interface.

Next part of the callback implements functions for IxChariot (TCL/C) API control. These functions will be described later.

After setting and running the network operation generator, an end of the callback function is reached. It is not desirable, however, to continue running the simulation. Termination of the co-simulation is compounded of two phases: termination of the OPNET Modeler simulation and of the external application. The former phase contains calling special *Esa_Terminate* function, the latter phase is handled by calling standard *exit* function.

```
Esa_Terminate(esaHandle,
      ESAC_TERMINATE_NORMAL);
exit(0);
```

Fig. 5 Termination of co-simulation.

4. Network Operation Generator IxChariot

4.1 IxChariot APIs

Network operation generator IxChariot, a product of company IXIA, disposes two APIs for interfacing with other applications. One of the interfaces is based on C programming language, while the other one uses TCL functions [1].

In the following chapters, approaches using both of the APIs will be described. Advantages and disadvantages of each approach will be discussed in the conclusion at the end of the article.

4.2 TCL API

To control IxChariot through TCL API, the external application has to implement a mechanism to call a TCL script that contains the very functions from the API. To enable this feature, a computer that is running the co-simulation must dispose a TCL runtime environment called "TCL Shell".
This chapter describes structure and contents of the TCL script and the calling mechanism in the external application.

4.2.1 TCL script
At the beginning, it is necessary to load libraries with the API functions. TCL language implements standard functions *load* and *package require* for these occasions (see Fig. 6) [2].

```
load ChariotExt
package require ChariotExt
```

Fig. 6 Loading libraries.

With the libraries successfully loaded, next step is to create variable for a test (packet flow generation context) and a pair (definition of a packet flow). The test variable should have set a filename to enable its storage later.

```
set test [chrTest new]
set pair [chrPair new]
chrTest set $test FILENAME <filename>
```

Fig. 7 Test and pair variables.

When the test variable is created, its duration can be set (see Fig. 8). Test options variable is loaded at first, using function *getRunOpts*. After, the duration is set on fixed time limit, and a required number of seconds is provided.

```
set runOpts [chrTest getRunOpts $test]
chrRunOpts set $runOpts TEST_END
      FIXED_DURATION
chrRunOpts set $runOpts TEST_DURATION
      <seconds>
```

Fig. 8 Test options.

Parameters of the generated packet flow are bound with the pair variable. These parameters are IP addresses of endpoints, a communication protocol, a script describing character of the communication (packet size, packet generation interval etc.), and a quality of service (QoS) settings. Setting of these parameters is shown in the Fig. 9. An expression *[lindex $argv X]*, where X stands for a number, means access to script's arguments. The number in the expression represents an index in a vector of arguments. The arguments passed to the script will be shown in the next chapter.

Setting QoS is, however, more complicated. IxChariot implements an external file for saving so called "QoS templates". These templates contain a unique title and a mask. The mask contains information about bit combination of the DSCP field inside IP headers of the generated packets. The information is stored in a form of an integer value with range from 0 to 255. This value is converted from decimal to binary format, which reveals the resulting bit settings.

```
chrPair set $pair E1_ADDR [lindex
      $argv 0]
chrPair set $pair E2_ADDR [lindex
      $argv 1]
chrPair set $pair PROTOCOL [lindex
       $argv 2]
chrPair useScript $pair <script>
```

Fig. 9 Communication parameters.

To set the pair with the QoS value from the argument vector, it is necessary to create a new template first. However, a *newQosTosTemplate* function that is used for this purpose throws an error in the case when a template with required title already exists. To avoid this behavior, the function is passed as an argument to a *catch* function. The *catch* function executes the function in its argument field, and in the case of failure, the error output of the executed function is stored in the *err* variable (see Fig. 10) and the script execution continues.

Right after is the *err* variable tested, and if it contains information about the error described above, another function will be called which would change already existing template.

The created/modified template is then assigned to the pair variable using standard procedure from the Fig. 9.

```
expr [catch {chrApi newQosTosTemplate
      CHR_QOS_TEMPLATE_TOS_BIT_MASK
      <title> [expr [lindex $argv 3]]
      } err]

if {$err == "creating QoS template
      failed: Value is invalid."}
   {chrApi modifyQosTosTemplate
      CHR_QOS_TEMPLATE_TOS_BIT_MASK
      <title> [expr [lindex $argv 3]]
      }

chrPair set $pair QOS_NAME <title>
```

Fig. 10 QoS settings.

When all settings are finished, the pair is associated with the test variable, and the test is started (see Fig. 11).

```
chrTest addPair $test $pair
chrTest start $test
```

Fig. 11 Starting the test.

The test is supposed to run for a limited time interval. The script acts like a supervisor and implements a control whether this limit has been reached, and, in the case the test is still running, explicitly stops the test run. In the Fig. 12, the function *isStopped* indicates whether the test stopped in a required time period, and the function *stop* forces the test to stop.

```
if {![chrTest isStopped $test
<timeout> ]} { chrTest stop $test }
```

Fig. 12 Stopping the test.

After the test has stopped, the script saves its configuration along with collected results to a file (defined in the Fig. 7). After that the script ends.

```
chrTest save $test
return
```

Fig. 13 Save and exit.

4.2.2 Script calling

The script has to be invoked from the code of the external application. To run the script, the TCL Shell program must be called with a proper path to the script as an argument. If the script itself takes any arguments, these arguments will be simply added behind the path to the script. The call from a command line can look like in the Fig. 14.

```
tclsh85 tcp_qos.tcl 192.168.166.252
      192.168.166.253 TCP 156 2>&1
```

Fig. 14 Script calling from command line.

The *tclsh85* is an executable of the TCL interpreter TCL Shell 8.5, the name of the script follows with the arguments of IP addresses and protocol. The number 156 represents a QoS mask with *EF* (Expedited Flow) mark. [6] The last part means routing of an error output of the script to its standard output. This feature will bring benefit later.

Calling the script from the external application is maintained by using functions *_popen* and *_pclose* (see Fig. 15).

The string *path* includes a path to TCL Shell executable and to the TCL script, and the *parameters* string contains arguments for the script shown in the Fig. 14. These two strings are concatenated together and passed as a parameter to the *_popen* function. This function invokes a new process and attaches its standard output to a file variable *output*.

```
output = _popen(strcat(path,
      parameters), "rt");

while(!feof(output))
{
if(fgets(buffer,128,output)!= NULL)
printf(buffer);
}
_pclose(output);
```

Fig. 15 Script calling from external application.

Inside the *while* cycle, the output is read and printed into the OPNET Modeler debugging console. With the error and the standard outputs chained together, the debugging console inside the OPNET Modeler is able to show errors that occurred in the script.

If the output reads *EOF* (end of file) mark, it will indicate the script has finished. The cycle is stopped, and the attachment of the output is closed by the *_pclose* function.

4.3 C API

This API is controlled via functions of the C programming language. As far as the external application is coded with C, these functions can be implemented right inside its code.

There is one disadvantage in comparison with the TCL API functions. In TCL language, all the functions include behavior for the case of an error. In C, a programmer has to implement his own solution.

Therefore, using the C API, the external application includes two internal (not exported) functions: the first one for setting up and starting the test and the second one for maintaining error correction.

To enable calling the IxChariot C API functions from the external application, it is necessary to include a header file *chrapi.h*.

4.3.1 Function for test setting up

The first function called from the C API is *CHR_api_initialize*. It is used to initialize the API and to access error information. If this function fails, it will mean the API is not initialized and the error correction would need to be maintained differently than with the other API functions (see Fig. 16).

```
rc = CHR_api_initialize
(CHR_DETAIL_LEVEL_ALL, errorInfo,
CHR_MAX_ERROR_INFO, &errorLen);

if (rc != CHR_OK)
{
printf("Initialization failed: rc =
      %d\n", rc);
  printf("Extended error info:\n%s\n",
      errorInfo);
  exit(255);
}
```

Fig. 16 API initialize.

The *rc* variable stores a return code of the function. After the function finishes, the value inside this variable is tested for errors. [2] This return code checking mechanism is implemented with every function call from the C API, but it will not be shown in the next code examples.

The following structure is similar to the TCL script. At first, a new test and pair are created. A filename is assigned to the test and the other parameters like IP addresses etc. to the pair (see Fig. 17).

```
CHR_test_new(&test);
CHR_pair_new(&pair);
CHR_test_set_filename(test, <file>,
      strlen(<file>));
CHR_pair_set_e1_addr(pair,<e1Addr>,
strlen(<e1Addr>));
CHR_pair_set_e2_addr(pair,<e2Addr>,
strlen(<e2Addr>));
CHR_pair_set_protocol(pair,
      <protocol>);
CHR_pair_use_script_filename(pair,
      <script>, strlen(<script>));
```

Fig. 17 Test and pair settings.

The situation with QoS settings is exactly the same as in the TCL script. At first is attempted to create a new template, and when the template already exists, it is modified. Then it is added to pair settings, the pair is attached to the test, and the test is started (see Fig. 18).

Now with the test running, the application waits until the test stops. In the *while* loop in the Fig. 19 are tested conditions whether the test is still running and whether the maximal wait time has been reached. The function *CHR_test_query_stop* returns a *CHR_OK* value in the case the test has stopped in the time interval defined in the *timeout* variable. If the return value is *CHR_TIMED_OUT*, timer will increase and the loop will continue. Any other return value indicates an undefined error.

```
CHR_api_new_qos_tos_template
      (CHR_QOS_TEMPLATE_TOS_BIT_MASK,
      <title>, <length>, <mask>);

if (rc == CHR_VALUE_INVALID)
CHR_api_modify_qos_tos_template
      (CHR_QOS_TEMPLATE_TOS_BIT_MASK,
      <title>, <length>, <mask>);

CHR_pair_set_qos_name(pair, <title>,
      <length>);

CHR_test_add_pair(test, pair);
CHR_test_start(test);
```

Fig. 18 QoS settings and start of the test.

Right after the loop is a condition investigating whether the test finished. If the answer is negative, an error will be thrown, telling that maximum time limit has been reached.

```
while(!isStopped && timer < maxWait)
{
rc = CHR_test_query_stop(test,
timeout);

if (rc == CHR_OK)
isStopped = CHR_TRUE;

else if (rc == CHR_TIMED_OUT)
{
timer += timeout;
printf("Waiting for test to stop...
       (%d)\n", timer);
}
else
show_error(test,rc,
       "test_query_stop");
}

if (!isStopped)
       show_error(test, CHR_TIMED_OUT,
              "test_query_stop");
```

Fig. 19 Test stop conditions.

In the opposite case is the test saved, and program returns into the callback function.

4.3.2 Function for error correction

The function for error correction is named *show_error*. Its main purpose is to gather information about an error and print it into the OPNET Modeler console. Its concept has been taken from examples in IxChariot SDK (Software Development Kit) folder.

To gather basic error information, a function *CHR_api_get_return_msg* is used. If this function succeeds, the console will show the information (see Fig. 20).

```
rc = CHR_api_get_return_msg(code, msg,
       CHR_MAX_RETURN_MSG, &msgLen);

if (rc != CHR_OK)
{
printf("%s failed\n", where);
printf("Unable to get message
       for return code %d, rc = %d\n",
       code, rc);
}
else
{
printf("%s failed: rc = %d
       (%s)\n", where, code, msg);
}
```

Fig. 20 Basic error information.

In specific cases when the return code carries a value *CHR_OPERATION_FAILED* or *CHR_OBJECT_INVALID*, extended information is available. This information is written into a log file (see Fig. 21).

```
fd = fopen(logFile, "a+");

rc = CHR_common_error_get_info(handle,
       CHR_DETAIL_LEVEL_ALL, errorInfo,
       CHR_MAX_ERROR_INFO, &errorLen);

if (rc == CHR_OK)
{
if (fd != NULL)
{
fprintf(fd, "%s %s failed\n",
       timestamp, where);
fprintf(fd, "%s %s\n", timestamp,
       errorInfo);

fclose(fd);
}
}
```

Fig. 21 Extended error information.

5. Conclusion

This article describes two possible approaches to create a middleware to interconnect an OPNET Modeler simulation with the IxChariot network operation generator. The approaches differ in an API that is used to maintain the connection.

Using the TCL API requires additional software (TCL Shell). Also, the middleware is divided between the external application and the TCL script, which means less consistency of the source code.

The C API, on the other hand, requires more programming effort during a development phase. Nevertheless, increased consistency and lesser software requirements in comparison with the TCL API make the C API right choice when interconnecting the OPNET Modeler with the IxChariot.

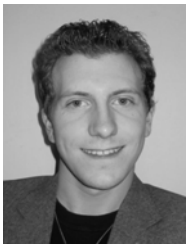
The TCL API could bring benefits if the application that generates data (OPNET Modeler for this case) was based on the TCL language. Such an application is the network simulator NS-2 for instance.

Acknowledgments

This paper has been supported by the Grant Agency of the Czech Republic (Grants No. GA102/09/1130 and No. 102/07/1012) and the Ministry of Education of the Czech Republic (Project No. MSM0021630513).

References

[1] IXIA. IxChariot User Guide, Release 7.0. 913-0843 Rev. *A*. July 2009.
[2] IXIA. IxChariot API Guide, Release 7.10. 913-0954-02 Rev. A. June 2010.
[3] OPNET TECHNOLOGIES. OPNET Modeler Documentation Set, Release 16.0. OPNET Technologies Inc. July 2010.
[4] HOŠEK, J.; RŮČKA, L.; MOLNÁR, K.; BARTL, M.; MATOCHA, T. Integration of Real Network Components into OPNET Modeler Co-simulation Process. WSEAS TRANSACTIONS on COMMUNICATIONS, 2010, volume 9, issue 9, p. 553-562. ISSN: 1109- 2742.
[5] BARTL, M. *Aplikace zpracovávající reálný síťový provoz v prostředí OPNET Modeler*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 31 s.
[6] BEDNÁRIK, J. Modelovanie komunikácie proprietárnym protokolom, určeným pre výmenu informácií s podporovanou technológiou QoS, v prostredí Opnet Modeler, Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2007. 35 s.



Milan Bartl is currently a 2nd-year master student at the Department of Telecommunications of the Faculty of Electrical Engineering and Communication, BUT. His master thesis is focused on the issue of cooperation between external systems and simulation environmet OPNET Modeler and its utilization in QoS assurance area.

Karol Molnar received his MSc. degree in Electronics and Communications (1997) and Ph.D. degree in Teleinformatics (2002) at Brno University of Technology (BUT), Czech Republic. He is with the Dept. of Telecommunications of the Faculty of Electrical Engineering and Computer Science, BUT as Assistant Professors (2002-2007) and Associate Professor (2008 – up-to-now). In his scientific work he focuses on modern network technologies, especially on topics of QoS support in both fixed and mobile network technologies. During the last several years he actively participates in theoretical and research works closely related to the technology of Differentiated Services.

Jiri Hosek received the B.S. and M.S. degrees in Electrical Engineering from Faculty of Electrical Engineering and Communication at the Brno University of Technology in 2005 and 2007, respectively. Recently he is studying for the Ph.D. degree. He is currently an assistant at the Department of Telecommunications of the Faculty of Electrical Engineering and Communication at the same university. His research work has been concentrated on the design of new methods for Quality of Service (QoS) assurance in data networks.