IMPLEMENTATION OF A TCP-BASED PROCESS MODEL IN OPNET MODELER

Karol Molnár

Department of Telecommunications, Faculty of Electronics and Communication, Brno University of Technology, Purkynova 118., 612 00 Brno, Czech Republic,
phone: +420 54114 9190, e-mail: molnar@feec.vutbr.cz

Computer-based simulation is an essential tool for the development and evaluation of communication technologies and processes. This paper introduces a way how to implement a proprietary network application and interlink the data transmission functions on the application layer with the TCP/IP protocol stack available in the simulation environment.

Keywords: OPNET Modeler, simulation, proprietary application

1. INTRODUCTION

With increasing demands on efficiency in network communications the design of new communication protocols and technologies requires a more detailed analysis and evaluation of the processes implemented. On the other hand, a detailed analysis of hardware prototypes is quite time-consuming and expensive. This is the reason why there is pressure on software simulation environments to allow building very complex and authentic simulation models.

The simple modelling of pure data flows with different traffic characteristics was overcome long ago. With increasing requirements for traffic classification and QoS support there is an extensive requirement for detailed modelling of every communication process used during data transmissions. The simulation of such details requires process modelling using finite state machines (FSM). This paper describes how to build a FSM to simulate the behaviour of a custom application and how to interlink this FSM with the TCP/IP protocol stack available in the OPNET Modeler simulation environment.

2. IMPLEMENTATION OF THE TCP/IP PROTOCOL STACK IN OPNET MODELER

The TCP/IP protocol stack represents a standard part of most network-component models available in OPNET Modeler. The implementation is based on sockets and is realized by a parent process *tcp_manager_v3* and child processes *tcp_conn_v3*.

The *tcp_manager_v3* is the manager of the whole TCP process. As a parent the *tcp_manager_v3* controls a set of TCP connections and starts the *tcp_conn_v3* process for each new request for establishing a TCP connection. The *tcp_conn_v3* process controls an individual TCP connection and manages the parameters of this connection.

The *tcp_manager_v3* process is a common part of the node model in the majority of network components in OM. It can control both active (client-side) and passive (server-side) TCP connections. According to the TCP/IP reference model the TCP node interacts with the user application and the IP node (providing the functions of the IP protocol). It receives data-units from the user-application and forwards them to the IP node. In the opposite direction it receives TCP segments and forwards them to the corresponding application. The communication with these modules is realised through interruptions. Fig. 1a) shows the process model of *tcp_manager_v3.*

The *tcp_conn_v3* process represents an individual TCP connection and is responsible for connection establishment, maintenance and termination. The process model of *tcp_conn_v3* is shown in Fig. 1b).

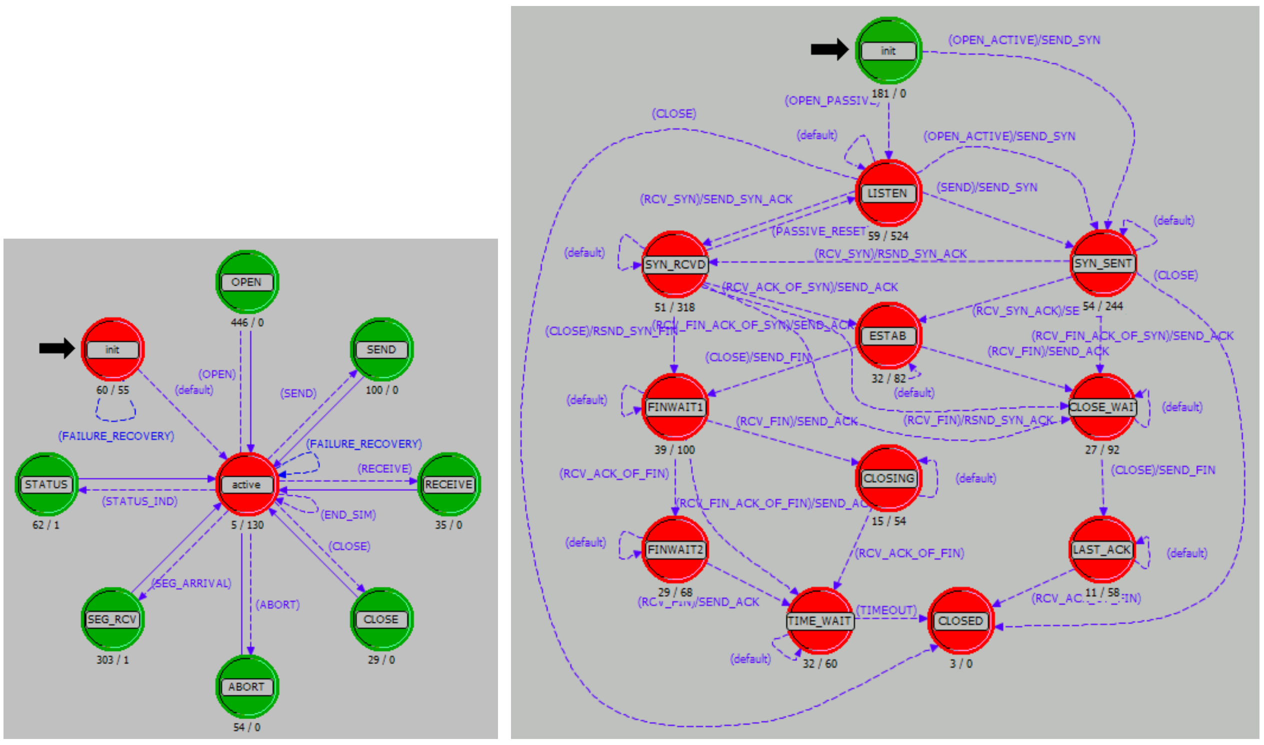

Fig. 1 a) Process model of *tcp_manager_v3* b) Process model of *tcp_conn_v3*

3. Building a Client Application

The simplest way how to implement a custom application in OM is to reuse an existing component like *ethernet_wkstn* for a client application and *ethernet_server* for a server. Do not forget to save the node model of these components under a new name to avoid the modification of the original models.

In the next step a corresponding processor node must be created in the node model, as shown in Fig. 2 a). In the figure the processor node is renamed to *Client*. The *Client* node must be connected with the *tcp* node using the Create Packet Stream command. Since the connection is bidirectional it must be applied in both directions. The same procedure is required when building the *Server* node in the modified *ethernet_server* component. Communication between the processes is realised through the ICI interface [3].

After the *Client* node has been placed into the node-model, the corresponding process model can be created in the Process Editor. The process model is represented by a finite state machine. An example of such a state machine is shown in Fig. 2b).

The initial model attributes can be configured through the *Interfaces → Model Attributes* panel. We should define attributes like Remote IP Address and Remote TCP port. The value of these attributes is then passed to the corresponding state variables (TcpT_Port rem_port and char rem_addr_str). The **init** state is responsible for the initialization of these state variables using the *op_ima_attr_get* (Objid, "attr_name", value_ptr) function. Since the IP address is stored in the attributes panel as a string for further use, it must be converted to an integer. This can be done using the *ip_address_create*(rem_addr_str) function. Additional initialization sequences can be included in this state if it is required by the application. Next, the application must register itself using the *tcp_app_register*(Objid) function. The function returns a TCP Application Handle, which is required in the following steps.

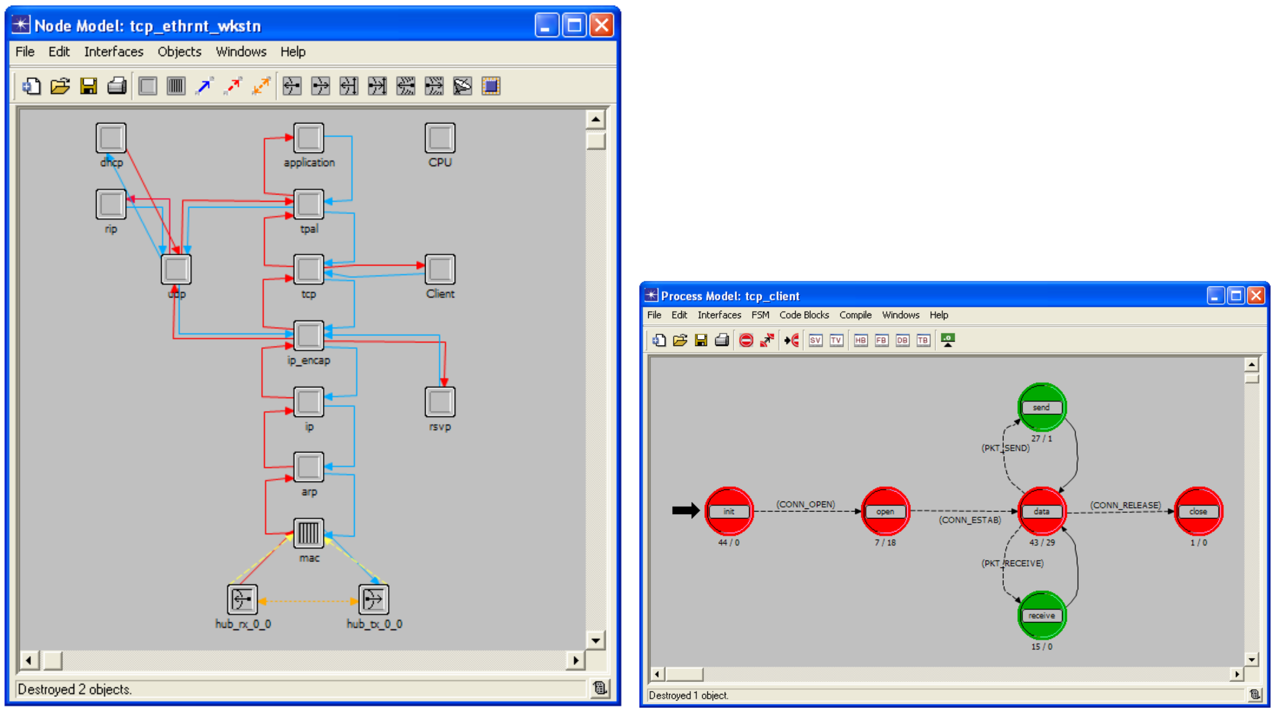

Fig 3 a) Example of a node model, b) Process model of a simple application

The **open** state contains only one function, which establishes connection with the server: *tcp_conn_open* (ApiT_Tcp_App_Handle*, remote_address, remote_port, local_port, command, ip_tos). The first parameter represents the TCP Application Handle obtained in the **init** state. The *command* parameter specifies the required TCP command. On the client side it is the TCPC_COMMAND_OPEN_ACTIVE command. The transition from the **open** to the **data** state is initiated by the OPC_INTPRT_REMOTE remote interruption and the TCPC_IND_ESTAB return value.

The **data** state contains the main application logic. In the simplest case this state processes three interruptions: one indicating an incoming data-unit, one when there is a data-unit ready to be sent and one when the connection is to be released. If any of these three interruptions occurs, the corresponding state is activated.

The **send** state creates and forwards data-units to the *tcp* node. To create a packet of 1024 bytes the *send_packet* = op_pk_create (1024); function can be used. Since the value of the *send_packet* variable changes for each packet, this temporary variable should be declared in the temporary variable (TV) block as Packet* *send_packet*;.

If required, an application data can be inserted into the *send_paket* using the op_pk_fd_set (*send_packet*, 0, OPC_FIELD_TYPE_STRUCT, *datapacket*, 512, op_prg_mem_copy_create, op_prg_mem_free, sizeof (TCPPACKET)); function, where TCPPACKET represents a custom data-structure and *datapacket* is a concrete instance of this type.

If the packet is, ready it can be forwarded to the *tcp* node. For this purpose the tcp_data_send (*tcp_app_handle*, *send_packet*); function can be used.

The **receive** state is responsible for the reception of incoming user data. In our simple example the incoming data units are simply destroyed. In the case of more complex applications the data received will be further processed. To receive a packet the corresponding ICI interface must be obtained and then the packet can be read:

//Obtainning the ICI pointer
ici_ptr = op_intrpt_ici();
//Reading the packet
receive_packet = op_pk_get (op_intrpt_strm ());

The data structure encapsulated in the packet can be extracted into the variable *datapacket* by the op_pk_fd_get (*receive_packet*, 0 , *&datapacket*); function. In the next step the application informs by the function tcp_receive_command_send (*tcp_app_handle*, 1); the *tcp* node that it is ready to receive the next data unit.

Before leaving the **receive** state the packet and the ICI interface must be destroyed:

op_pk_destroy (*receive_packet*);
op_ici_destroy (op_intrpt_ici ());

The **close** state is the last state of our application. Its function is to close the TCP session. It is realized by the tcp_connection_close(*tcp_handle*); function, where the *tcp_handle* variable represents the TCP Application Handle obtained at the registration of our application.

4. Building a Server Application

The construction of the server application is very similar to the client application described in the previous chapter. Hence only the differences will be mentioned here.

In the **init** state the remote address and the port number are different.

In the case of an application server a passive connection should be opened which listens for incoming requests from the clients. In the **open** state the connection type must therefore be TCPC_COMMAND_OPEN_PASSIVE. Next the *tcp* node should be informed that our application is ready to receive data. This is realized by the tcp_receive_command_send (*tcp_app_handle*, 1); function.

Usually the application logic in the **data**, **receive** and **send** states is also different at the server and the client sides. Since our simple example just transmits dummy TCP packets, which are destroyed on the opposite side of the connection, the same functions can be implemented. It can be useful to implement an additional indication of the received packets.

5. SIMULATION RESULTS

To evaluate the desired behaviour of our simulation model we can start the simulation and select some TCP-related statistics to log. Some examples of simulation results are shown in Fig. 4, 5, 6.

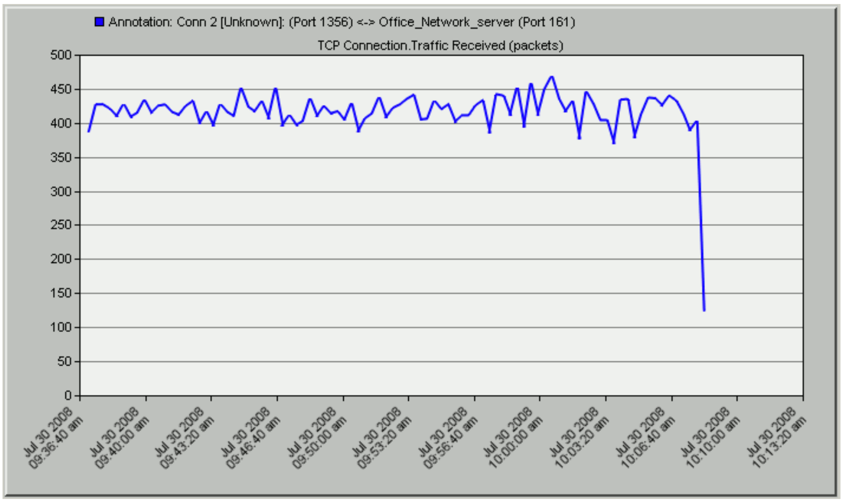

Fig 4 Received TCP traffic in the simulation model

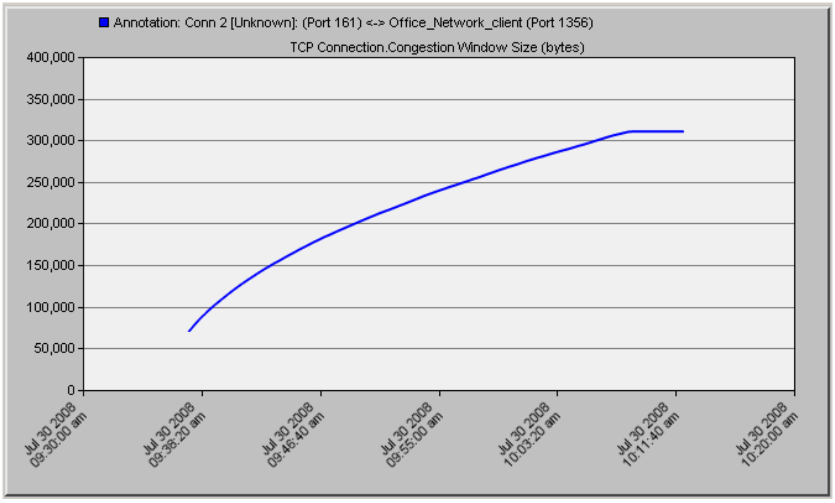

Fig 5 Congestion window size

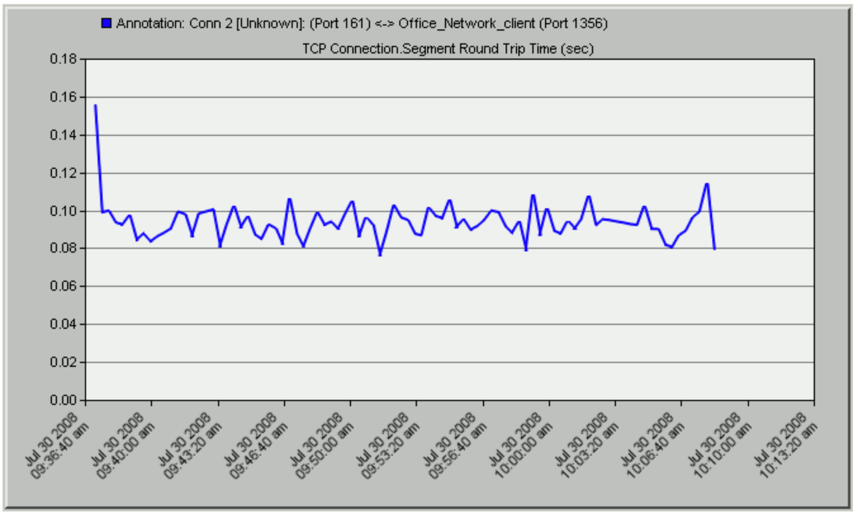

Fig 6 Round Trip Time

As it is evident from the previous charts the TCP transmission was realized in an ideal network which did not introduced any losses or significant transmission delays. Of course, such events also can be modelled in OPNET Modeler.

6. Conclusion

From the previous chapters it should be evident that building a simulation model of a complex networking application is quite easy and straightforward. On the other hand, it allows us to implement sophisticated application logic, which can very authentically simulate the behaviour of real network services. Such a level of modelling and simulation gives results which are comparable with those obtained from measurements of real systems.

I hope that this short paper will prove to be motivating and helps other interested persons to build more complex and refined simulation models.

Acknowledgement

This paper has been supported by the Grant Agency of the Czech Republic (Grant No. 102/06/1569) and the Ministry of Education of the Czech Republic (project No. MSM0021630513).

7. References

[1] OPNET Technologies, Inc., OPNET Modeler Release 14 Product documentation, 2008.
[2] Tirinda V. Využití protokolu TCP v simulačním prostředí OPNET Modeler, VUT, Brno, 2008 (in Czech).
[3] Zeman O. Implementace simulačního modelu zjednodušené databáze DiffServ□MIB, VUT, Brno, 2008 (in Czech).