

# Notes on OPNET

for

ECE 158, Data Networks, UCSD

R. L. Cruz, 4/98

## Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>The simulation kernel procedures and symbolic constants</b> | <b>1</b> |
| 1.1      | Encapsulation/De-encapsulation of Packets within Packets . .   | 4        |
| <b>2</b> | <b>Creating and Viewing Animations</b>                         | <b>5</b> |
| <b>3</b> | <b>Using the Debugger</b>                                      | <b>6</b> |

## 1 The simulation kernel procedures and symbolic constants

- Refer to OPNET SIMULATION KERNEL MANUAL (online) for details of Kernel Procedures (KPs) and symbolic constants. For your convenience, some common kernel procedures and symbolic constants that *may* be useful for this lab and later labs are briefly summarized below. (You certainly don't need them all).
- More convenient access to (abbreviated) on-line documentation for each kernel procedure can be obtained by typing "Control"-q inside an editor pad within the process editor. Typing the 'F4' key when a family of kernel procedures is selected will display information about that set of procedures. Typing the 'F4' key when a particular simulation kernel procedure is selected will insert a 'call template' into the editor pad at the current cursor position.

### Brief Description of Common KPs:

**op\_ev\_cancel(*evhandle*)** Cancels an event that has been previously scheduled.

**op\_intrpt\_code()** Returns a user defined numeric code associated with the current interrupt of the invoking process.

**op\_intrpt\_stat()** Returns the input statistic index associated with the current interrupt of the invoking process.

- op\_intrpt\_schedule\_self(*time, code*)** Schedules a self interrupt for the invoking process at the specified time, and associates it with a user defined numeric code. Returns an event handle, which can be stored in a state variable of type Evhandle (see **op\_ev\_cancel()**).
- op\_intrpt\_strm()** Returns the stream index associated with the current interrupt of the invoking process.
- op\_intrpt\_type()** Returns the type of the current interrupt which invoked the process.
- op\_pk\_copy(*pkptr*)** Creates a new packet which is an exact duplicate of the specified original packet. Returns a pointer to the newly created packet.
- op\_pk\_create\_fmt(*format\_name*)** Creates a new formatted packet with a predefined structure described by the specified packet format. Returns pointer to created packet.
- op\_pk\_destroy(*pkptr*)** Deallocates the specified packet, releasing associated memory resources for other purposes.
- op\_pk\_get(*instrm\_index*)** Returns a pointer to a packet that has arrived on an input packet stream, and removes the packet from the stream.
- op\_pk\_send(*pkptr, outstrm\_index*)** Forwards the specified packet through an output packet stream, schedules a stream interrupt at the destination module for the current simulation time (which indicates the packet's arrival) , and releases the ownership of the packet by the invoking process.
- op\_pk\_send\_forced(*pkptr, outstrm\_index*)** Forwards the specified packet through an output packet stream, releases the ownership of the packet by the invoking process, and forces a stream interrupt at the destination module and *immediate* invocation of the corresponding destination process (indicating the packet's arrival).
- op\_pk\_nfd\_set(*pkptr, fd\_name, value*)** Assigns the value of an integer, double, or packet field in the specified packet; the field-of-interest is specified by name. The value assigned to a field can be an integer, a double, or a packet pointer.

- op\_pk\_nfd\_get(*pkptr, fd\_name, value\_ptr*)** Obtains the value of a field in the specified packet; the field-of-interest is specified by name. *value\_ptr* is a *pointer* to a variable to be filled with the fields value.
- op\_sim\_time()** Returns the current simulation time.
- op\_stat\_reg(*name, index, type*)** Registers a statistic with name *name* (a string) and returns a value of type *Stathandle* for use with kernel procedures such as **op\_stat\_write** (). The string *name* should match with that of an entry in the “Declare Local Statistic” or “Declare Global Statistic” table within the process editor. The integer argument *index* acts as a “subscript” for dimensioned statistics. For undimensioned statistics (the usual case), the value **OPC\_STAT\_INDEX\_NONE** should be used for *index*. The integer argument *type* should be **OPC\_STAT\_LOCAL** for local statistics and **OPC\_STAT\_GLOBAL** for global statistics.
- op\_stat\_write(*stathandle, value*)** Writes the argument *value* (of type double) to the statistic with handle *stathandle*.
- op\_stat\_local\_write(*outstat\_index, value*)** Writes a numeric value to an output statistic of the surrounding processor or queue; the statistic value is tagged with the current simulation time.
- op\_stat\_local\_read(*instat\_index*)** Returns current value of specified input statistic.
- op\_subq\_empty(*subq\_index*)** Returns code indicating if the specified subqueue is empty of packets.
- op\_subq\_pk\_access(*subq\_index, pos\_index*)** Returns a pointer to a packet stored in the specified subqueue, at the specified subqueue position. (does not remove packet from subqueue)
- op\_subq\_pk\_insert(*subq\_index, pkptr, pos\_index*)** Inserts the specified packet into the specified subqueue, at the specified position. Returns Boolean value indicating if insertion was completed successfully.
- op\_subq\_pk\_remove(*subq\_index, pos\_index*)** Returns a pointer to a packet stored in the specified subqueue at the specified subqueue position, and removes the packet from the subqueue.

**op\_subq\_stat**(*subq\_index*, *stat\_index*) Returns the value of a statistic which is automatically computed by the surrounding queue; the statistic pertains to the specified subqueue, rather than the queue as a whole.

**OPC\_INTRPT\_SELF** self interrupt (possible returned value of **op\_intrpt\_type()**)

**OPC\_INTRPT\_STRM** stream interrupt (possible returned value of **op\_intrpt\_type()**)

**OPC\_INTRPT\_STAT** statistic interrupt (possible returned value of **op\_intrpt\_type()**)

**OPC\_QSTAT\_PKSIZE** index used for the current number of packets in the queue/subqueue (e.g. the second argument of **op\_subq\_stat()** )

**OPC\_QPOS\_TAIL** tail position of subqueue (e.g. the second argument of **op\_subq\_pk\_access()** or **op\_subq\_pk\_remove()**)

**OPC\_QPOS\_HEAD** head position of subqueue (e.g. the second argument of **op\_subq\_pk\_access()** or **op\_subq\_pk\_remove()**)

**op\_pk\_stamp**(*pkptr*) Stores information in the specified packet which records the current location(i.e., module) and simulation time, for later reference.

**op\_pk\_stamp\_time\_get**(*pkptr*) Returns the simulation time (of type double) when the specified packet was last stamped.

## 1.1 Encapsulation/De-encapsulation of Packets within Packets

The example below illustrates how packet encapsulation is accomplished with OPNET. The example first creates a “lower layer” packet with format “my\_format”, which is defined in the parameter editor. ( The format “my\_format” was constructed in the parameter editor to contain a field named “data\_field”, which is of type “packet”. By setting the length of the field “data\_field” to -1, this indicates that the size of the field will be determined by the size of the packet that is written into it. )

In the example code below, an “upper layer” packet is encapsulated into this newly created “lower layer” packet.

```

/* create lower layer packet with format "my_format" */
lower_layer_pkptr = op_pk_create_fmt ("my_format");

/* encapsulate upper layer packet within lower layer packet */
/* put upper layer packet in field "data_field" defined in format
   "my_format"*/
op_pk_nfd_set( lower_layer_pkptr , "data_field" , upper_layer_pkptr);

```

Note that after the upper layer packet is encapsulated, the process doing the encapsulation loses “ownership” of the encapsulated packet, and in some sense “ownership” is transferred to the lower layer packet. Thus, the process doing the encapsulation will no longer be able to access fields within the encapsulated packet.

The example below illustrates how the encapsulated packet can be removed from the lower layer packet which contains it (de-encapsulation). After de-encapsulation, the process acquires ownership of the packet that was encapsulated.

```

op_pk_nfd_get( lower_layer_pkptr , "data_field" , &upper_layer_pkptr);

```

## 2 Creating and Viewing Animations

We describe here a simple way of creating and viewing animations. See the on-line documentation to use more powerful features.

- Create a probe file within the Probe Editor. After loading the network model, create an “Automatic Animation Probe” by pushing the appropriate button and dragging it into the work area. After selecting the probe and clicking the right mouse button on it to view its attributes, enter “top” (without the quotes) into the “subnet” attribute of the probe. This indicates that the entire network will be animated. You may also specify the start and stop times for which animation is to occur, if you do not wish to animate the entire simulation. For the “representation” attribute, select “packet flows”, which will cause packet movement to be animated. Save the probe file.
- At simulation time, within the simulation editor, set up the simulation as usual. Make sure it refers to the appropriate probe file with the animation probe you created. Within the attribute dialog box of a

simulation object, open the dialog box for “Animation Attributes”. Select “Send animation to history file”, and specify a file that will be used to store the animation created during simulation. Run the simulation as usual.

- After the simulation has run, open a new shell window (or suspend the execution of `opnet` in the current window (Control-z). Type the command `m3_vuanim`, which will launch the animation viewer application. After loading the animation file you created earlier, start the animation by pushing the “play button.” Note: you may create a postscript file of a screen image by using the “capture screen bitmap” button at the bottom, and then the “send page to printer” button. This file can then be sent to the printer. In order to do this, you will need to set up an environment variable prior to starting the `m3_vuanim` program (otherwise TIFF files will be created instead of postscript). Specifically, use the command  
`setenv screen_bitmap_type epsi`  
before running `m3_vuanim`.

### 3 Using the Debugger

The debugger provides an interactive environment for tracing the execution of a simulation. To run a simulation through the debugger, set up the simulation within the Simulation Editor as usual. Within the attribute dialog box of a simulation object, open the dialog box for “Environment Files”, and select the “debug” entry so that it is “included”. After starting the simulation, refer to the shell window that is running `opnet`. After some initialization, you should get the “odb” prompt within the window.

- You can type `help` to get a list of debugger commands. Briefly, these commands allow you to select and de-select various tracing modes, and control the execution of the simulation by setting stopping points. Some common commands:

**cont** This command will cause the simulation to continue until the next breakpoint, or until the end of the simulation if there is no breakpoint. Usually this command is used after turning on or off some tracing mode.

- next** Causes simulation of next event, and returns to debugger after simulation of this event (“single step”).
- evstop** The command **evstop nnn** causes a breakpoint to be set for event number **nnn**. Event numbers are integers, and are usually referenced within OPNET error messages.
- tstop** The command **tstop nnn.nn** causes a breakpoint to be set for the specified simulation time **nnn.nn**. Simulation times are real numbers, and are usually referenced within OPNET error messages.
- fulltrace** This toggles the fulltrace mode. If fulltrace is on, all events will be traced as the simulation is run. Usually there is a lot of uninteresting information that is generated, so it is common to turn the mode on slightly ahead of a known time in the simulation where there is a problem. For example, if the simulation aborts due to an error, one might note the event number where the simulation “bombs.” Then one would use the debugger and set a breakpoint for a few events before the “problem event.” Once this breakpoint is reached, fulltrace can be turned on, and the simulation continued. The output could then be used to figure out why the problem is occurring.
- status** Lists breakpoints and traces (and IDs).
- ltrace** The command **ltrace string** activates the labelled trace identified with the label **string**, where **string** is a character string. When a labelled trace is active, then when the KP procedure *op\_prg\_odb\_ltrace\_active()* is executed during a simulation of an event (within some process model), with an argument of “**string**”, it will return the value *OPC\_TRUE*. This KP is often used to enable conditional execution of printf statements, for debugging.
- susptrace** The command **susptrace nnn** suspends (de-activateates) the trace with ID **nnn**, where **nnn** is an integer. Trace IDs can be obtained with the **status** command.
- quit** Terminate simulation and generate an output vector file.