UNIVERSITÀ DI PISA

SWEDISH ICT SICS

SEA++

USER MANUAL
DRAFT

SEA++ 0.99 INET based (pre-release)

October 3, 2016

Edited by

Francesco Racciatti[1]
Marco Tiloca[1,2]
Gianluca Dini[1]

SEA++ 0.99 INET based (pre-release)
User manual (DRAFT)

Francesco Racciatti[1] · Marco Tiloca[1,2] · Gianluca Dini[1]

AFFILIATIONS

1. Dept. of Ingegneria dell'Informazione, University of Pisa, Largo Lazzarino 1, 56100 Pisa, Italy
2. SICS Swedish ICT AB, Security Lab, Isafjordsgatan 22, SE-164 40 Kista, Sweden

CONTACTS

- racciatti.francesco@gmail.com · write to Francesco Racciatti
- marco@sics.se · write to Marco Tiloca
- gianluca.dini@ing.unipi.it · write to Gianluca Dini

Contents

List of Figures

Chapter 1

Build SEA++

SEA ++ is based on many components and makes use of multiple programming languages. In particular SEA++:

- is written in C++11 and Python 2.7.6;
- extends INET 2.6, which is based on OMNeT++ 4.x;
- requires the `libxml` library;

SEA++ is designed for Linux distributions based on Debian and it was not tested on other OSs. It is fully compatible with Ubuntu 14.04 LTS, but it is possible using it on older or newer versions by adopting several shrewdnesses.

1.1 Build SEA++ on Ubuntu 14.04 LTS

The steps to build SEA++ are the following:

1. get and install `libxml` library;
2. get and build OMNeT++ 4.x (preferibly 4.6 or greater);
3. get and build SEA++.

If errors occur during the installation of some component, it should refer to official guide:

- `libxml` - `http://www.xmlsoft.org`
- OMNeT++ - `http://omnetpp.org`
- SEA++ - `https://github.com/seapp/src/archive/v0.99.tar.gz`

1.1.1 Get and install `libxml` library

The `libxml` library can be installed by using the `apt-get`:

```
~$ sudo apt-get install libxml++2.6-dev libxml++2.6-doc
```

The `libxml` library sources are available at `http://libxmlplusplus.sourceforge.net/`.

1.1.2 Get and build OMNET++

To get OMNeT++ 4.6 open your terminal, go in your home and type:

```
~$ wget http://omnetpp.org/omnetpp/send/30-omnet-releases/2290-omnet-4-6-source-
    ide-tgz
```

When the download finishes, untar and unzip the source files in your home:

```
~$ tar xvfz 2290-omnet-4-6-source-ide-tgz
```

It creates the directory `omnetpp-4.6`.

Set environment variables (assuming you are using bash as your shell):

```
~$ export PATH=$PATH:~/omnetpp-4.6/bin
~$ export LD_LIBRARY_PATH=~/omnetpp-4.6/lib
```

Append the above commands to the `.bash_profile` file.

It is now possible to build OMNeT++:

```
~$ cd omnetpp-4.6/
~/omnetpp-4.6$ NO_TCL=1 ./configure
~/omnetpp-4.6$ make
```

1.1.3 Get and build SEA++

To get SEA++ open your terminal, go in your home and type:

```
~$ wget https://github.com/seapp/src/archive/master.zip
```

When the download finishes, unzip and rename the source directory in your home:

```
~$ unzip master.zip
~$ mv src-master seapp-0.99
```

It creates the directory `seapp-0.99`.

It is now possible to build SEA++:

```
~$ cd seapp-0.99/
~/seapp-0.99$ make makefiles
~/seapp-0.99$ make
```

The `seapp-0.99` directory on your home should contain the following sub-directories:

`seapp/`	SEA++ root directory
`  interpreter/`	ASL interpreter
`  src/`	sources
`    actions/`	action classes
`      actionbase/`	action base class
`      change/`	change class
`      clone/`	clone class
`      create/`	create class
`      destroy/`	destroy class
`      disable/`	disable class
`      drop/`	drop class

`move/`	move class
`put/`	put class
`retrieve/`	retrieve class
`seappexpression/`	seappexpression class
`send/`	send class
`attacks/`	attack classes
`attackbase/`	attack base class
`attackentry/`	attack entry class
`physicalattack/`	physical attack class
`conditionalattack/`	conditional attack class
`unconditionalattack/`	unconditional attack class
`exmachina/`	exmachina class
`globalfilter/`	globalfilter class
`localfilter/`	localfilter class
`parser/`	parser class
`variable/`	variable class
`.../`	standard INET classes
`examples/`	examples ready to use
`.../`	standard INET classes

1.2 Build SEA++ on older Ubuntu versions

To build SEA++ on older Ubuntu versions, in addition to the steps described above you may need to:

- upgrade the Python interpreter;
- upgrade the C++ compiler;
- change SEA++ makefile;
- change OMNeT++ configure file.

1.2.1 Upgrade Python interpreter

SEA++ uses features provided by Python 2.7.6, so it is necessary to install at least this version of its interpreter.

1.2.2 Upgrade C++ compiler

SEA++ uses features provided by C++11. It is necessary to install at least the version 4.7 of gcc/g++ compilers and set them as default.

1.2.3 Link libraries in SEA++ makefile

SEA++ is built and checked for Debian Linux Distributions and not for other OSs. It is necessary to check if libraries used by SEA++ are linked to the correct paths. After downloading SEA++, in the initial makefile, all the links to the libraries are under `-l/usr/lib/x86_64-linux-gnu/*` path.

You may need to change the path to `-l/usr/lib/i386-linux-gnu/*` depending on system's architecture.

1.2.4 Change CFLAGS in OMNeT++ configure file

SEA++ is based on INET and the `opp_makemake` tool of OMNeT++ is used in order to compile the framework. The file `configure.user` is the one where all the necessary parameters (compilers, flags, etc) are declared. To enable the features provided by C++11 you have to declare the option `'-std=c++11'` in the flag CFLAGS of OMNeT++'s `configure.user` file.

Chapter 2

Summary of ASL primitives

2.1 Actions

The set of the available actions is shown below.

destroy The *destroy* action is a node action used to destroy a node. The node discards all the packets and it is present in the simulation field.

Listing 2.1: ASL destroy example

```
#primitive: destroy(nodeId, t)
destroy(5, 200)
```

Listing 2.2: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>200</start_time>
      <node>5<node>
      <action>
        <name>Destroy</name>
      </action>
    </Attack>
  </Physical>
</configuration>
```

disable The *disable* action removes the target node from the simulation field.

Listing 2.3: ASL disable example

```
#primitive: disable(nodeId, t)
destroy(5, 200)
```

Listing 2.4: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>200</start_time>
      <node>5<node>
```

```
      <action>
        <name>Disable</name>
      </action>
    </Attack>
  </Physical>
</configuration>
```

move The *move* action is a node action used to move a node.

Listing 2.5: ASL move example

```
#primitive: move(nodeId, t, x, y, z)
move(5, 100, 10, 10, 0)
```

Listing 2.6: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>100</start_time>
      <node>5<node>
      <action>
        <name>Move</name>
        <parameters>10:10:0</parameters>
      </action>
    </Attack>
  </Physical>
</configuration>
```

retrieve The *retrieve* action is a packet action used to retrieve the content of a field of a target packet and to store it into a variable.

Listing 2.7: ASL retrieve example

```
#primitive: retrieve(packetName, fieldName, variableName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
 filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
     destinationPort" == "2000")
    var destPort
    retrieve(original, "UDP.sourcePort", destPort)
}
```

Listing 2.8: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <var>
        <name>destPort</name>
        <value></value>
        <type>NONE</type>
      </var>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Retrieve</name>
```

```
      <parameters>packetName:original:field_name:UDP.sourcePort:varName:
          destPort</parameters>
    </action>
  </Attack>
 </Conditional>
</configuration>
```

In the ASF the conditional operators are stored in *reverse order* to that of the operands.

drop The *drop* action is a packet action used to discard a target packet.

Listing 2.9: ASL drop example

```
#primitive: drop(packetName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
      destinationPort" == "2000")
    drop(original)
}
```

Listing 2.10: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Drop</name>
        <parameters>packetName:original</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

clone The *clone* action is a packet action used to create a packet that is a clone of a target packet.

Listing 2.11: ASL clone example

```
#primitive: clone(packetName, clonedPacketName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
      destinationPort" == "2000")
    packet dolly
    clone(original, dolly)
}
```

Listing 2.12: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
```

```
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

The interpreter handles the variables and the packets in two different ways. In the ASL file, the user has to declare both the variables and the packets but in the ASF the declared packets do not appear, unlike the declared variables.

create The *create* action is a packet action used to create a packet ex-novo.

Listing 2.13: ASL create example

```
#primitive: create(packetName, layer5.type, value5, layer4.type, value4, layer3.
     type, value3, layer2.type, value2)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    packet exNovo
    create(exNovo, "APP.type", "0000")
}
```

Listing 2.14: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Create</name>
        <parameters>packetName:exNovo:APP.type:0000</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

change The *change* action is a packet action used to change the content of a field of a target packet.

Listing 2.15: ASL change example

```
#primitive: change(packetName, fieldName, variableName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    change(original, "UDP.destinationPort", 5000)
}
```

Listing 2.16: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <var>
        <name>5000</name>
        <value>5000</value>
        <type>NUMBER</type>
      </var>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Change</name>
        <parameters>packetName:original:field_name:UDP.destinationPort:value
             :5000</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

In the ASL file, even if the user uses variables which has not been explicitly declared, the interpreter automatically declares and initializes the variables (if it is necessary).

send Given a packet, cloned or created (and correctly fillled), which belongs to the layer L, the *send* action is used to send the packet to the bottom layer.

Listing 2.17: ASL send example

```
#primitive: send(packetName, forwardingDelay)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    packet dolly
    clone(original, dolly)
    send(dolly, 0)
}
```

Listing 2.18: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
      <action>
        <name>Send</name>
        <parameters>packetName:dolly:delay:0</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

put The *put* action is usefull to transmit packets from a node to a set of recipient nodes bypassing the communication channel.

Listing 2.19: ASL put example

```
#primitive: put(packetName, recipientNodes, direction, updateStats,
     forwardingDelay)
list targetList = {1,2,5}
list dstList = {6}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    packet dolly
    clone(original, dolly)
    put(dolly, dstList, TX, FALSE, 0)
}
```

Listing 2.20: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
      <action>
        <name>Put</name>
        <parameters>packetName:dolly:nodes:6:direction:TX:throughWC:false:delay
             :0</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

2.2 Variables and expressions

2.2.1 Variables

The ASL handles variables which can store both numbers and strings. The user must declare the variable before using it. The sintax to declare a variable is:

Listing 2.21: Sintax to declare a variable

```
var foo
```

The AS introduces the type transparency than the user has not to declare the type of the content of the variable, e.g. integer, or double, or string.

Expressions The ASL handles expressions that make possible both operations and assignments. In the table 2.1 is shown the ASL expression table.

An example of ASL expression is:

operator	**numbers**	**strings**
$=$	supported	supported
$+=$	supported	not supported
$-=$	supported	not supported
$\times =$	supported	not supported
$/=$	supported	not supported
$\div =$	supported	not supported

(a) Assignment operators

operator	**numbers**	**strings**
$+$	supported	supported
$-$	supported	not supported
$\times$	supported	not supported
$/$	supported	not supported
$\div$	supported	not supported

(b) Arithmetic operators

operator	**numbers**	**strings**
$<$	supported	supported
$>$	supported	supported
$<=$	supported	supported
$>=$	supported	supported
$==$	supported	supported
$!=$	supported	supported

(c) Comparison operators

operator	**numbers**	**strings**
AND	supported	supported
OR	supported	supported

(d) Logical operators

Table 2.1: ASL expression table

Listing 2.22: Sintax expression example

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
```

As specified above, the ASL introduces the type transparency and a variable can store both numbers and strings. This feature makes possible to initialize a variable with a number and after assign a string to it:

Listing 2.23: Legal expressions

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
result = "Hello, world!"  # legal expression
```

However the user has got the responsibility to ensure the consinstency of the expressions:

Listing 2.24: Illegal expressions

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
result = "Hello"    # legal expression
result += ", world!"    # legal expression
var operand3 = 5
resutl += operand3    # illegal expression
```

Chapter 3

Set up and run a new SEA++ scenario

SEA++ is distributed with a complete set of examples that are ready to use. However, the process to build a new SEA++ scenario is substantially identical to that of INET.

In the following example is shown how to set up a new simple scenario in which there are a client and a server.

1st step - make the folder As in INET, the 1st step is to make the folder that will contain all the new files, e.g. scenario.

```
~/seapp-0.99/inet/examples/ mkdir scenario
~/seapp-0.99/inet/examples/ cd scenario
```

2nd step - network description As in INET, the 2nd step is to edit the `.ned` file that contains the network description, e.g. `scenario.ned`.

Listing 3.1: scenario.ned

```
package inet.examples.inet.scenario;

import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.nodes.inet.StandardHost;
import inet.globalfilter.GlobalFilter;

network scenario
{
  parameters:
    string attackConfigurationFile = default("none");
    double n;

  submodules:
    globalFilter: GlobalFilter;
    client: StandardHost;
    server: StandardHost;
    configurator: IPv4NetworkConfigurator;

  connections allowunconnected:
    client.pppg++ <--> { datarate = 10Mbps; } <--> server.pppg++;
```

```
    globalFilter.nodes++ <--> client.global_filter;
    globalFilter.nodes++ <--> server.global_filter;

}
```

In this step, is fundamental to:

- add the the string parameter `attackConfigurationFile` to the network;
- import the GlobalFilter class, declare a GlobalFilter submodule and connect it to all the other nodes.

In case of **disable** primitive it is necessary to import and declare the submodule 'ExMachina' in the .ned file.

3^rd^ step - edit omnetpp.ini As in INET, the 3^rd^ step is to edit the `omnetpp.ini` file. In this step is fundamental to bind the configuration(s) with the ACF(s), by overwriting the name of the network parameter `attackConfigurationFile` with the name of a particular ACF.

```
[General]
network = scenario
sim-time-limit = 600s

// General settings ...

// Config(s) specific settings
[Config attack-example]
**.attackConfigurationFile = attack-example.xml

[Config attack-example2]
**.attackConfigurationFile = attack-example2.xml

// ...
```

4^th^ step - add the ACF The 4^th^ and last step is to add the ACF(s) in the folder.

Run the simulation The simulation is ready to run. In the terminal, call `run_inet` that is in the `src` folder:

```
~/seapp-0.99/inet/examples/scenario ../../src/run_inet $*
```

It will start the simulation supported by the GUI, as shown in figure 3.1. Be carefull to specify well the path from the current folder (i.e. the simulation folder) to the `src` folder.

To run the simulation in express mode without use the GUI, type:

```
~/seapp-0.99/inet/examples/scenario ../../src/run_inet -u Cmdenv -f omnetpp.ini
```

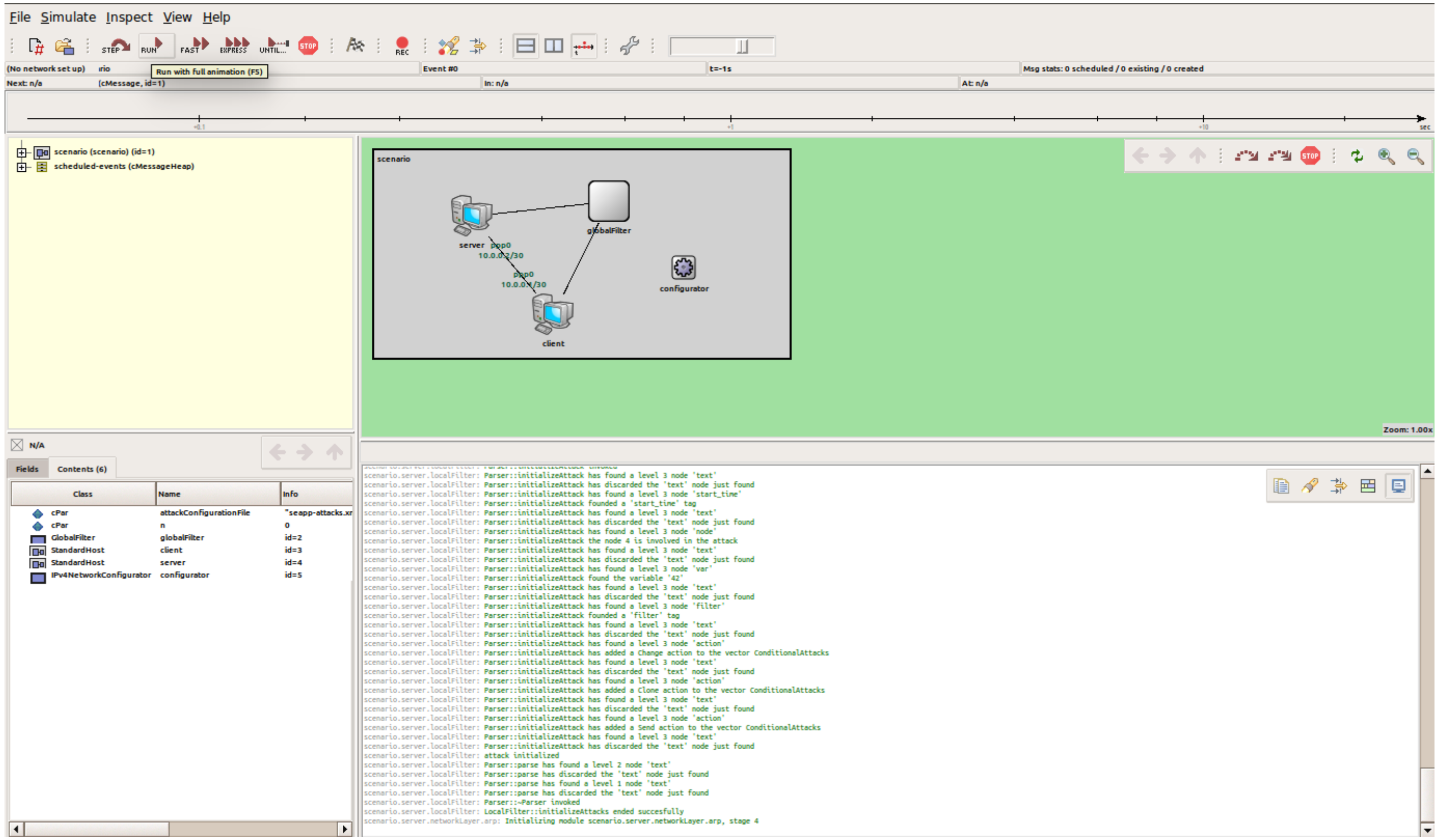

Figure 3.1: Simulation scenario

Chapter 4

Create and fill packets

This chapter shows informations that may help the user to create and fill a packet properly. To build a packet from scratch that belongs to a certain layer, the user has to know its structure and the protocol that runs on the layer below or above. The user has also to know the output gate of the local filter (or local filters) to which forward the packet.

4.1 Handle ControlInfo object

After the creation of a packet, the user has to fill its header by using the action change. In some cases the user has also to fill the fields contained in the `ControlInfo` object appended to packets. The `ControlInfo` object contains commands and informations that are used by the recipient layer to handle properly the incoming packets.

The tables below show the packets that the user can create. Some packets are associated with the related ControlInfo object by default.

Example If the user wants to create a generic packet of layer 5 and send it to the bottom layer, he must know the protocol that runs on the layer 4, for example UDP. By analyzing the table 4.8, which specifies the structure of the ControlInfo object, the user finds the field which has to fill: `sockId`, `destAddr`, `destPort`, `srcAddr`, `interfaceId`.

4.2 Handle output gate

When the user creates a new packet, it has to specify the output gate in the local filter by using the action change and the keyword `sending.outputGate`. For example, if the user creates an application packet that flows in recepit direction, it has to specify the gate `app_udp_sup$o[0]`.

Listing 4.1: Handle output gate example

```
create(newPacket, ...)
...
change(newPacket, "sending.outputGate", "app_udp_sup$o[0]")
```

Table 4.1: ControlInfo object structure

From layer 5 to layer 4			
ASL type	**Packet type**	**ControlInfo object**	**fields**
APP.0000	cPacket	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId
APP.0100	cPacket	TCPSendCommand	connId userId
APP.0201	TimingReport	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId
APP.0301	TimingCommand	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId

Table 4.2: ControlInfo object structure

From layer 4 to layer 5			
ASL type	**Packet type**	**ControlInfo object**	**fields**
APP.0000	cPacket	UDPDataIndication	sockId srcAddr destAddr srcPort destPort ttl interfaceId typeOfService
APP.0100	cPacket	TCPCommand	connId userId

Table 4.3: ControlInfo object structure

<table>
<tr><th colspan="4">From layer 4 to layer 3</th></tr>
<tr><th>ASL type</th><th>Packet type</th><th>ControlInfo object</th><th>fields</th></tr>
<tr><td>TRA.0000</td><td>UDPPacket</td><td>IPv4ControlInfo</td><td>destAddr
srcAddr
interfaceId
multicastLoop
protocol
typeOfService
timeToLive
dontFragments
nextHopAddr
moreFragments
macSrc
macDest
diffServCodePoint
explicitCongestionNotification</td></tr>
<tr><td>TRA.0010</td><td>TCPSegment</td><td>IPv4ControlInfo</td><td>destAddr
srcAddr
interfaceId
multicastLoop
protocol
typeOfService
timeToLive
dontFragments
nextHopAddr
moreFragments
macSrc
macDest
diffServCodePoint
explicitCongestionNotification</td></tr>
</table>

Table 4.4: ControlInfo object structure

From layer 3 to layer 4			
ASL type	**Packet type**	**ControlInfo object**	**fields**
TRA.0000	UDPPacket	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification
TRA.0010	TCPSegment	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification

Table 4.5: ControlInfo object structure

From layer 3 to layer 2			
ASL type	**Packet type**	**ControlInfo object**	**fields**
NET.0000	IPv4Datagram	none	none
NET.0010	IPv4Datagram	Ieee802Ctrl	src dest etherType interfaceId switchPort ssap dsap pauseUnits

Table 4.6: ControlInfo object structure

From layer 2 to layer 3			
ASL type	**Packet type**	**ControlInfo object**	**fields**
NET.0000	IPv4Datagram	-	-
NET.0010	IPv4Datagram	Ieee802Ctrl	src dest etherType interfaceId switchPort ssap dsap pauseUnits

Table 4.7: ControlInfo object structure

From layer 2 to layer 1			
ASL type	**Packet type**	**ControlInfo object**	**fields**
`MAC.0000`	`PPPFrame`	-	-
`MAC.0010`	`EthernetFrame`	-	-
`MAC.0020`	`IdealAirFrame`	-	-
`MAC.0030`	`AirFrame`	-	-

Table 4.8: ControlInfo object structure

From layer 1 to layer 2			
ASL type	**Packet type**	**ControlInfo object**	**fields**
`MAC.0000`	`PPPFrame`	-	-
`MAC.0010`	`EthernetFrame`	-	-
`MAC.0020`	`IdealAirFrame`	-	-
`MAC.0030`	`AirFrame`	-	-