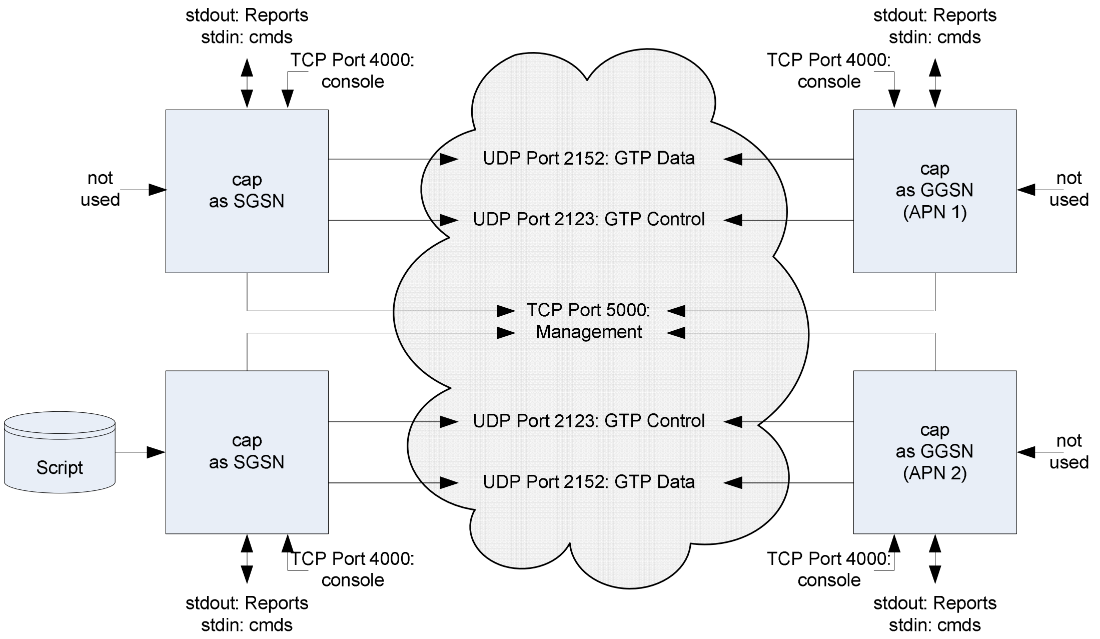
cap: A packet generator for the UMTS Gn/Gp interface

Table of Contents:

Contact author at raj@pejaver.com for executables.

Architecture



An arbitrary number of hosts running cap can be used to simulate a UMTS core network. The program can be used in any of several modes. Typically, one of the hosts will be the ‘controller’ and will execute scripts from a script file. Scripts allow GTP packets to be composed and transmitted, and allow triggers to be defined. A script can also contain commands to control the other hosts. These hosts receive commands from the controller over the ‘management’ connection (TCP port 5000, by default.) Besides being a “controller”, each instance of cap can simulate a GGSN, SGSN or CDF (Charging Function). A host can be a controller and also simultaneously simulate a device by exchanging GTP packets. However, there are some advantages to dedicating a host for just sending messages to other hosts that simulate devices.

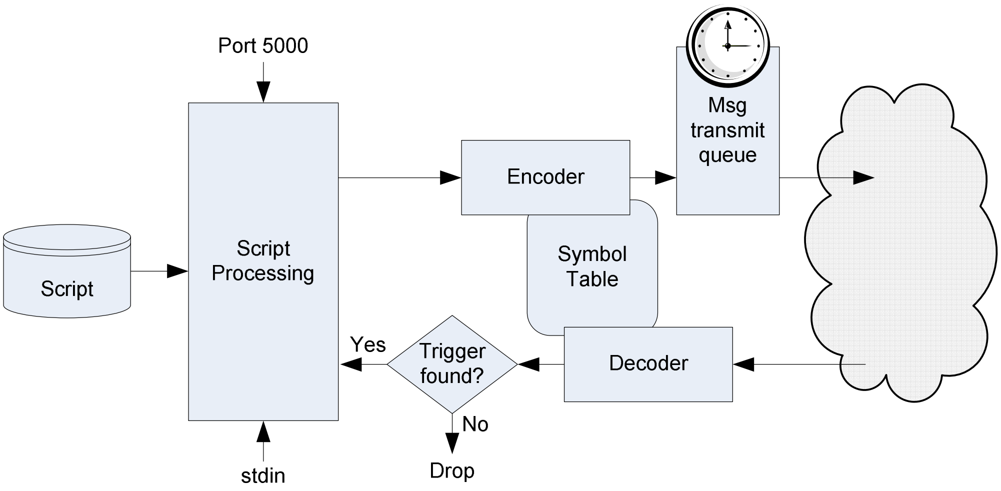
Each cap will also respond to interactive script commands from stdin. Interactive commands can also be sent to remote instances of cap using ‘nc’ to TCP port 4000. Connecting to port 5000 using ‘nc’ may seem to work, but is not recommended. GTP traffic is typically exchanged over UDP ports 2123 and 2152. GTP traffic can be sent to other ports via script options. All default port assignments can be overridden on the command line. To run cap:

```
  cap [-t mgmtp] [-s stdp] [-u ctlp] [-d datap] [cmdfile]*
```

The internal architecture of cap is shown below. GTP messages containing Gn procedures are composed and queued for transmission at a specified time. Incoming messages can trigger predefined script statements, which can then compose messages in response.

Scripts compose the messages that are to be transmitted. A transmission time is specified for each message. After they are encoded, outbound messages are added to a transmit queue and are transmitted at the specified time.

The program reads and processes the contents of the script files specified in the cap command line when it starts. After all commands have been processed, the program waits for additional commands from stdin while also listening for incoming UDP packets and incoming TCP connections. The program will terminate only when the time value specified in a $quit statement expires.



Script Execution Contexts

Messages are composed in one of three Execution Contexts:

1. The main script files specified in the cap command line. Messages composed interactively via stdin (or remote console) also fall into this category.
2. The script received from a control host due to a $remote statement. Each remote script received is an independent context.
3. The body of a trigger executed due to a matching incoming message. Each trigger invocation is an independent context. This is the most powerful envronment.

The script composes the messages that are to be transmitted. After they are encoded, outbound messages are added to the transmit queue. The exact time of transmission is controlled the `$TimeDelay` variable. It specifies the transmission time as the number of milliseconds after the start of the script environment.

The value of `$TimeDelay` is reset to 0 at the start of each of the above three situations. If the script changes the value of this variable to 300, then the next message will be transmitted no earlier than 300 ms after the start of the environment. For example, while responding to a incoming message in a trigger, setting `$TimeDelay` to 300 will cause the response to be sent 300 ms after the incoming message is received.

The script will usually increment the value of `$TimeDelay` to maintain the correct sequence of message transmission. However, the value can be reset to 0 if necessary. Note that the value of `$TimeDelay` at the end of one script (like a trigger) will not affect the timings of another script (like a trigger for the next message.)

Tunnel Endpoint IDs

Cap maintains a table of Tunnel Endpoint IDs (TEIDs). Based on the value of the `GTPtied` field in an incoming packet, it sets up the following four variables for use by a triggered script: `$PeerControl, $PeerData, $LocalControl, $LocalData`.

Writing Scripts

Syntax

Scripts are a sequence of statements. Statements can assign values to variables and fields, compose and send messages, or control the flow of the script.

All statements have a syntax like: **Name = Value**.

All characters in a line following a '#' or "//" up to the newline character are ignored. Either format can be used for comments.

White space is defined as the set of characters ' ' (space), ',' (comma), ';' semicolon, '\t'(tab), '\n' (newline) and '\r' (return). Whitespace is generally ignored, but will separate names and values, i.e., Names cannot contain whitespace characters.

The delimiter between a Name and a Value can be '=' (equal) or ':' (colon).

Variables

Variable names consist of the characters A-Z, 0-9, $ and _. All names are case insensitive and can be up to 127 chars long. To identify a name, the script must specify enough leading characters of the name to find a unique match. For example, `$Destination` can be specified as `$dest` since no other name starts with `$dest`.

By convention, predefined system variables start with a $. System variables are shown in a table below. For example: `$destination, $repeat`. It is suggested that all user defined variables also start with a $.

IE Field names and GTP header variables do not start with a $. These names are listed in separate tables below. Example names are: `GTPteid, NSAPI`.

Assigning a value to a system variable usually causes the variable to be updated. Some values are set by cap for use by the script. For example, `SenderIP, $Repeat, $EndRepeat`. It doesn't make sense to assign values to some variables, like `$end` or `$rand`.

Assigning an IP address value to a procedure name is how message composition is started.

Assigning a value to an IE Field variable causes that IE to be inserted into the outgoing message. This is how outgoing messages are composed. Assigning a value to a GTP header variable sets those values in the GTP header of outgoing message. These variables are also used to access values of fields in incoming messages.

Values

Values can be up to 127 characters long. If a value needs to contain whitespace, then the string should be enclosed in quotes. Either ‘”’ (double quote) or “’” (single quote) can be used. For example, “this is a string”, and ‘192.168.1.100, [1111:2222::9999]’.

The @ Operator

A value can also be an expression that has been evaluated using the ‘@’ operator. For example, `@1+2+3` evaluates to 6. The available operators are listed in the table below.

Note that the expression is evaluated from left to right and all operators have equal precedence. So, `@2+2*3` evaluates to 12, not 8. There are no parenthesis. If one is needed then the expression can be evaluated in multiple statements.

If there is whitespace between the values and operators then the whole expression must be within quotes.

Numbers can be in decimal (123), hexadecimal (0xffee) or octal (0777).

If a non numeric string is used in an arithmetic operation, it is treated as a 0.

When a comparison operator is used, the result will be a 1 (true) or 0 (false). For example, @25%20>=1+1 will evaluate to 2.

Strings and numbers can be concatenated using ‘&’. FYI, MS Excel uses this operator to concatenate.

+	Add	numbers only
-	Subtract	numbers only
*	Multiply	numbers only
/	Divide	numbers only
%	Modulo	numbers only
&	Concatenate	numbers and strings
>	Compare “greater than”	numbers only
>=	Compare “greater than or equal to”	numbers only
<	Compare “less than”	numbers only
<=	Compare “less than or equal to”	numbers only
==	Compare “equal to”	numbers and strings
!=	Compare “not equal to”	numbers and strings

Variable Substitution

The current value of a variable is substituted during expression evaluation. For example: `$TimeDelay = @$TimeDelay+300` causes the system variable to be incremented by 300. `NSAPI = “@$repeat % 11”` causes an NSAPI IE to be added to the outgoing msg with the computed value. Note that the second example had quotes only because it contains space characters for readability. It is equivalent to: `NSAPI = @$repeat%11`.

IE value Substitution

Referencing an IE or GTP header field during expression evaluation while responding to an incoming message has a special meaning. It causes the corresponding value to be extracted from the incoming message and inserted into the expression. For example, `NSAPI = @NSAPI+1` causes an IE to be added to the outgoing message with a value that is 1 more than the NSAPI value in the incoming message.

IE value substitution happens only for Trigger execution contexts. This is the only context where there is a message to be read.

Note that a null string is substituted if the incoming message does not contain the referenced field. For example, if the incoming message does not contain the NSAPI IE, “`NSAPI = @NSAPI+1`” will evaluate to “`NSAPI = @+1`”, which will cause an error. The $If statement can be used to verify existence of a IE in the incoming message. For example,

```
    $if = @NSAPI
        NSAPI = @NSAPI+1
    $endif = 1
```

Composing and Sending Messages

Messages are composed by listing the IEs between the ProcedureName statement and a “$end” statement. The ProcedureName statement lists the Name and the target IP address and UDP port. For example: `CreatePDPContextRequest = 192.168.2.22:2123`. If the port number is not specified, the default port is 2123 (or whatever was specified in the –u option.) IE statements made outside the scope of a message composition will be ignored (i.e., before a ProcedureName statement or after the $end statement). Examples of composing and sending a message are:

```
    CreatePDPContextRequest = 192.168.2.22
        IMSI = "4045612345678
        Recovery = 1
        TEIDData = 0x12345678
        TEIDControl = 0x12345679
        NSAPI = 10
        GSNAddress = 10.40.30.41
        GSNAddress = 10.40.30.42
        EndUserAddress = 10.22.33.44
        AccessPointName = cuegroup.com
        MSISDN = 12122215151
        RATType = 4
    $end = send
```

and

```
      GPDU = 192.168.1.104:2152
            GTPTEID = @TEIDData
            GTPSeq = @GTPSeq
            gpdudata = "this is a time for all good men to eat
cake"
            $timedelay = @$timedelay+200
            $print = "Sending GPDU to GGSN"
      $end=send
```

Script Control

There are several script control constructs available.

`$repeat = nnn`
`    statement`
`    statement`
`    ….`
`$endrepeat = 1`
	The block of embedded statements will be repeated nnn times. Two system variables are automatically updated during each iteration:
	• $repeat will contain the iteration count in the range 1 .. nnn
	• $endrepeat will contain the repeat count (nnn). The value specified for $endrepeat is ignored (specified as 1 in the example to the left.)

`$remote = ipaddr [:prt]`
`    statement`
`    statement`
`    ….`
`$endremote = 1`
	The block of embedded statements will be sent to the remote host. This allows commands and trigger definitions to be sent to the remote host, thereby avoiding the need for script files at that end. The default value for the destination port is 5000. The value specified for $endremote is ignored (specified as 1 in the example to the left.)

`$trigger = Proc[:tid]`
`    statement`
`    statement`
`    ….`
`$endtrigger = 1`
	The block of embedded statements will be executed when an incoming Procedure is seen. If a tunnel ID (tid) is specified, then the GTPteid field of the incoming message must match it. Multiple triggers may be defined. Trigger definitions persist. Redefinitions will cause the latest definition to be used. The value specified for $endtrigger is ignored (specified as 1 in the example to the left.)

`$if = condvalue`
`    statement`
`    statement`
`    ….`
`$endif = 1`
	The block of embedded statements will be executed if the condvalue is a nonzero numeric value. The condvalue will typically be an expression starting with ‘@’. The value specified for $endif is ignored. For example:
	`$if = @GTPsuspendrequest`
	`      GTPsuspendresponse = 1`
	`$endif = 1`

`$include = filename`	The block of statements in the specified file will be processed. Processing will continue after the `$include` statement when all the lines in the file have been processed. Nested includes are allowed. It is sometimes useful to type in a `$include` command interactively on the console to issue a sequence of commands.

`$quit = timedelay`	The program will stop after the specified number of milliseconds. A zero value will cause the program to terminate immediately, which may be a bad idea if there are unsent messages in the transmit queue. It is possible to type in a `$quit` command interactively on the console to cause the program to display the final report and exit.
`$sleep = timedelay`	The program will sleep for the specified number of milliseconds. The program will not send out queued messages or respond to incoming messages while it is sleeping. Most scripts will increment the value of the `$TimeDelay` variable instead of using `$sleep`. This construct is useful mostly when a controller is sending commands to other SGSN and GGSN simulators with `$remote` statements.
`$define = varname`	This allows a new user defined variable to be created. Redefining a previously defined user defined variable will erase its previous value.
`$print = string`	The specified string is printed when the statement is executed. Note that, depending on the value of `$TimeDelay`, the message containing the `$print` statement may be transmitted much later than when the string is displayed.

System Variables

`$timedelay = value`	This variable defines when a queued message is actually transmitted. It is detailed in its own section.
`$timedistribution = n`	This option delays the message by a random time in the range 0..n-1 milliseconds. It does not change the value of the `$TimeDelay` variable.
`$dropPercentage`	Must be a value in the range 0 to 100. It specifies the percentage of packets awaiting transmission that will be randomly dropped.
`$rand`	Random value, a 32 bit integer. A different value is returned each time the variable is accessed. For example: `$var1 = @$rand`
`$peerData`	The Data TEID of the peer. For example, while sending a GPDU, `GTPteid = @$peerData`
`$peerControl`	The control TEID of the peer. For example, while sending a GPDU, `GTPteid = @$peerControl`
`$localData`	The Data TEID of the receiving side associated with `$peerData` and `$peerControl`.
`$localControl`	The Control TEID of the receiving side associated with `$peerData` and $peerControl.
`$var1`	Can be used in scripts by those too lazy to use `$define`
`$destination`	Can be used in scripts by those too lazy to use `$define`

Data packets

The content of a transmitted GPDU is specified by setting the `gpdudata` field. The contents of an incoming data packet can be obtained by reading this field. For example:

`gpdudata = “Hi Mom”`	Encapsulates and sends a static string
`gpdudata = @$var1`	Encapsulates and sends contents of $var1

Note that Wireshark sometimes flags data packets as being malformed because it expects gpdu data to start with an IP layer header.

PCAP Streams

The `$definepcap` keyword can be used to read packets from a PCAP file and insert them into GPDU packets. The first 14 bytes of each packet read is to be an Ethernet header and is skipped.

`$definepcap = $strname`	This defines a new stream variable called `$strname`. The variable is not yet associated with any PCAP file. There can be more than one stream open at a time. The stream file will be closed at EOF.
`$strname = “filename”`	This construct opens a PCAP file and associates it with the stream `$strname`. An error message will be printed if the file cannot be opened.
`gpdudata = @$strname`	This reads the next packet from the stream and inserts it into `gpdudata`. Packet data cannot be moved to any variable other than `gpdudata`. This construct would typically be placed inside a loop. If no packet is read, then `gpdudata` will contain 0 bytes.

Example:

```
$definepcap = $stream               // define variable $stream
$stream = "test/test.pcap"          // open pcap file

$var1 = 1                           // first sequence number
$repeat = 10                        // send 10 packets
      GPDU = 192.168.1.60:2152      // construct a GPDU
            GTPTEID = 1234567       // something…
            GTPSequence = @$var1

            gpdudata = @$stream     // read next packet from pcap
      $end = GPDU                   // end GPDU definition

      $var1 = @$var1+1              // increment sequence number
$endrepeat = 10
```

Supported Procedures

All the procedures in 3GPP 29.060 are supported.

ID	ProcedureName
1	EchoRequest

2	EchoResponse
3	VersionNotSupported
16	CreatePDPContextRequest
17	CreatePDPContextResponse
18	UpdatePDPContextRequest
19	UpdatePDPContextResponse
20	DeletePDPContextRequest
21	DeletePDPContextResponse
22	InitiatePDPContextActivationRequest
23	InitiatePDPContextActivationResponse
26	ErrorIndication
27	PDUNotificationRequest
28	PDUNotificationResponse
29	PDUNotificationRejectRequest
30	PDUNotificationRejectResponse
48	IdentificationRequest
49	IdentificationResponse
50	SGSNContextRequest
51	SGSNContextResponse
52	SGSNContextAcknowledge
53	ForwardRelocationRequest
54	ForwardRelocationResponse
55	ForwardRelocationComplete
56	RelocationCancelRequest
57	RelocationCancelResponse
58	ForwardSRNSContext
59	ForwardRelocationCompleteAcknowledge
60	ForwardSRNSContextAcknowledge
254	EndMarker
255	GPDU

Supported IE fields

The following Information Elements defined in 3GPP 29.060 are supported.

ID	IEName	Type
1	Cause	char
2	IMSI	TBCD
3	RAI	TBCD
4	TLLI	int
5	PacketTMSI	int
8	ReorderingRequired	char
11	MAPCause	char
13	MSValidated	char
14	Recovery	char
15	SelectionMode	char
16	TEIDData	int
17	TEIDControl	int
19	TeardownInd	char

20	NSAPI	char
21	RANAPCause	char
23	RadioPrioritySMS	char
24	RadioPriority	char
25	PacketFlowId	short
26	ChargingCharacteristics	short
27	TraceReference	short
28	TraceType	short
29	MSNotReachableReason	char
127	ChargingID	int
128	EndUserAddress	EUAddr
131	AccessPointName	string
132	ProtocolConfigurationOptions	string
133	GSNAddress	IP
134	MSISDN	TBCD
138	TargetIdentification	string
139	UTRANTransparentContainer	string
141	ExtensionHeaderTypeList	string
142	TriggerId	string
143	OMCIdentity	string
144	RANTransparentContainer	string
145	PDPContextPrioritization	string
147	SGSNNumber	string
148	CommonFlags	char
149	APNRestriction	char
150	RadioPriorityLCS	char
151	RATType	char
163	HopCounter	char
165	MBMSSessionIdentifier	char
166	MBMS2G3GIndicator	char
167	EnhancedNSAPI	char
170	MBMSSessionRepetitionNumber	char
171	MBMSTimeToDataTransfer	char
173	BSSContainer	string
176	BSSGPCause	char
177	RequiredMBMSbearercapabilities	string
178	RIMRoutingAddressDiscriminator	char
181	MSInfoChangeReportingAction	char
183	CorrelationID	char
184	BearerControlMode	char
185	MBMSFlowIdentifier	string
187	MBMSDistributionAcknowledgement	char
188	ReliableINTERRATHANDOVERINFO	char
190	FQDN	string
191	EvolvedAllocationRetentionPriority1	char
197	CSGMembershipInformation	char
202	GGSNBackOffTime	char

GTP Header control variables

The following field variables can be read and written. While responding to an incoming message, these variables can be accessed to read values in the incoming message. Setting the values will set GTP header values in the outgoing message being composed. For example,

```
GTPSequence = @GTPSequence // copy value from incoming msg
GTPteid = $PeerControl     // peer's tunnel ID
```

GTP Header	Description
GTPteid	Tunnel Endpoint Identifier (32 bit integer)
GTPsequence	Sequence Number (16 bit integer)
GTPnpdu	N-PDU Number (16 bit integer)
GTPpdcppdunum	PDCP PDU Number
GTPsuspendrequest	Suspend Request (0 or 1)
GTPsuspendresponse	Suspend Request (0 or 1)
GTPmbmssupport	(0 or 1)
GTPmsinfochange	(0 or 1)

The following field is extracted from the incoming UDP packet.

senderip	IP address of the sender of the incoming packet. It is used while composing a response in triggers. For example: UpdatePDPContextResponse = @senderIP

Report

A report is generated at the end of a run, when $quit fires. The report contains:

- Name and number of procedures processed, incoming and outgoing.
- A table of local and remote TEIDs (data & control), IMSI and NSAPI. This can be useful while using the associated PCAP.

Appendix A: Unimplemented IEs

The following Information Elements defined in 3GPP 29.060 are not supported.

9	AuthenticationTriplet
12	PTMSISignature
18	TEIDData2
22	RABContext
129	MMContext
130	PDPContext
135	QOSProfile
136	AuthenticationQuintuplet
137	TFT
140	RABSetupInformation
146	AdditionalRABSetupInformation
152	UserLocationInformation
153	MSTimeZone
154	IMEISV
155	CAMELChargingInformationContainer
156	MBMSUEContext
157	TMGI
158	RIMRoutingAddress
159	MBMSProtocolConfigurationOptions
160	MBMSServiceArea
161	SourceRNCPDCPcontextinfo
162	AdditionalTraceInfo
164	SelectedPLMNID
168	MBMSSessionDuration
169	AdditionalMBMSTraceInfo
174	CellIdentification
175	PDUNumbers
179	ListofsetupPFCs
180	PSHandoverXIDParameters
182	DirectTunnelFlags
186	MBMSIPMulticastDistribution
189	RFSPIndex
192	EvolvedAllocationRetentionPriority2
193	ExtendedCommonFlags
194	UCI
195	CSGInformationReportingAction
196	CSGID
198	AMBR
199	UENetworkCapability
200	UEAMBR
201	APNAMBRwithNSAPI
203	SignallingPriorityIndication
204	SignallingPriorityIndicationwithNSAPI
251	ChargingGatewayAddress

Appendix B: Sample Scripts

```
// Example: create PDP Context and exchange GPDUs
$remote = 192.168.1.104                 # send script to remote cap
       $trigger = CreatePDPContextRequest      # define a trigger on remote cap

              $var1 = @$rand%100000*2     # random even number in range 0..199998

              // Send response when a CreatePDPContextRequest is received
              CreatePDPContextResponse = @senderIP   # reply
                     GTPTEID = @TEIDControl          # from incoming PDU
                     cause = @$rand%4+128            # random number
                     IMSI = @imsi                    # copy from req pkt
                     TEIDData = @$var1               # set random teid
                     TEIDControl = @$var1+1
                     NSAPI = @nsapi                  # copy from req pkt
                     EndUserAddress = 20.30.40.50
                     GSNAddress = 10.20.30.41
                     GSNAddress = 10.20.30.42
                     $timedelay = 30                 # delay before response
                     $print = "Reply with CreatePDPContextResponse"
              $end = CreatePDPContextResponse

              // send a few GPDUs to SGSN
              $timedelay=@$timedelay+1000            # wait 1 sec after prev
              $repeat = 2
                     GPDU = @senderip:2152           # back to sender
                            GTPTEID = @TEIDdata
                            GTPSequence = $repeat
                            gpdudata = "Wicked winged wabbits"
                            $timedelay=@$timedelay+30
                            $print = "Sent: Wicket winged wabbits"
                     $end = GPDU
              $endrepeat = 0

       $endtrigger = CreatePDPContextRequest
$endremote = 192.168.1.104              # end of script sent to remote cap

// main
$repeat = 10                            # create 10 PDP contexts
       $var1 = @$rand%100000*2          # random even number in range 0..199998

       # packet definition, starts a new transaction
       CreatePDPContextRequest = 192.168.1.104       # destination GGSN
              GTPTEID = 0                            # always 0 for CreatePDP
              IMSI = "@12345678 & $var1"             # concatenate
              Recovery = 1                           # whatever
              TEIDData = @$var1                      # use random values
              TEIDControl = @$var1+1                 #
              NSAPI = @$repeat%11                    # $repeat=iteration count
              GSNAddress = 10.40.30.41
              GSNAddress = 10.40.30.42
              EndUserAddress = 10.22.33.44
              AccessPointName = cuegroup.com
              MSISDN = 12122215151
              RATType = 4
              $timedelay=@$timedelay+300       # 300 ms between requests
              $print = "Sending CreatePDPContextRequest to GGSN"
       $end = CreatePDPContextRequest          #
$endrepeat = 0
$quit = 60000              # auto stop after 1 min, wait for responses to come
```