Rapr Tutorial

1. What is RAPR?

The Real-time Application Representative (RAPR) (pronounced "wrapper") is open source software developed by the Naval Research Laboratory (NRL) PROTocol Engineering Advanced Networking (PROTEAN) group. It was designed specifically for testing application messaging behavior and performance in a MANET environment, under high packet loss conditions, but can be used to to emulate generic application performance in any IP environment using both UDP and TCP transport mechanisms.

RAPR can generate and respond to real-time network traffic in a manner closely tied to application behavior so that the network can be loaded in a variety of ways in a controlled and repeatable manner. It uses the Mgen toolkit (MGEN) to generate network traffic and supports most of the traffic shaping functionality available in Mgen Version 5.0. The network traffic can be logged for subsequent analysis and can be used to calculate performance statistics on throughput, packet loss rates, communication delay, and more.

2. What makes RAPR different from other traffic generators?

Many traffic generators are "open-loop" controllers with respect to network traffic - they push packet streams onto a network without regard to the success of the messaging transactions. RAPR is an "open-loop" and "closed-loop" controller as well, managing transaction oriented traffic between nodes. There is a causal relationship between the traffic generated and the packet loss and delay. In addition, Rapr is capabale of generating random yet repeatable traffic and behavior patterns. An xml-based "dictionary" is available that can be used to simplify scripting of system behavior.

3. How to install RAPR

- Get the lastest RAPR distribution at http://cs.itd.nrl.navy.mil/work/rapr/index.php and untar the distribution.

  You may either use the binary provided or build the source as described in the following bullets.

- Get the MGEN distribution at http://cs.itd.nrl.navy.mil/products and untar the distribution relative to the RAPR distribution.
- Get the PROTOLIB distribution at http://cs.itd.nrl.navy.mil/products and untar the protolib distribution relative to the mgen/common directory.
- Get SPRNG 2.0 or greater at http://sprng.cs.fsu.edu/ and install as directed. Modify the `/rapr/unix/Makefile.common` SPRNG_DIR, SPRNG_INC, and SPRNG_LIB directories to point to the location of the sprng lib andinclude directories.
- Get STLPORT 4.6.2+ or greater at http://www.stlport.org and install as directed. If the STLport libraries are not put in a system wide location, modify the `rapr/unix/Makefile.common` STLPORT_INC and STLPORT_LIB directories to point the the correct locations
- Type `make -f Makefile.linux` in the `/rapr/unix` directory to build the application.
- See /rapr/README.TXT for additional installation instructions if necessary.

4. How to send messages from one node to another using RAPR from the command line

The rapr "event" command can be used to send commands to RAPR via the command line. Start RAPR on the listening node and tell it to listen to a specific port:

```
./rapr event "LISTEN UDP 5000"
```

Start RAPR on the sending node and tell it to send a packet at test time 0.0 (immediately) to the listening node and enable transmission logging. (In this example and the others, replace the IP address and port numbers with those appropriate for your environment):

```
./rapr txlog event "0.0 DECLARATIVE UDP DST 192.168.1.103/5000"
```

The declarative command directs rapr to send one packet to the specified destination. The txlog command enables transmission logging. Since there are no other events to be processed, RAPR will exit after the packet is transmitted. The application will log output to STDOUT by default:

```
[debussy:]$ ./rapr txlog event "0.0 DECLARATIVE UDP DST 192.168.1.103/5000"
rapr:version 0.5.1c
mgen:version 5.01b
rapr: starting now ...
23:02:10.555132 START
19:02:10.555126 app>RAPR type>Application action>ApplicationStartUp
19:02:10.559678 app>RAPR type>Declarative action>start ubi>1744830464 \
eventSource>script_event mgenCmd>"ON 1 UDP SRC 0 DST 192.168.1.103/5000 \
PERIODIC [1 1024 ] DATA [0304C657A66A] COUNT 1"
23:02:10.559978 SEND proto>UDP flow>1 seq>1 srcPort>33594 dst>192.168.1.103\
/5000 size>1024
19:02:11.550338 app>RAPR type>Declarative action>timeout ubi>1744830464 \
0.001 OFF 1
19:02:11.552187 RAPRSTOP
23:02:11.552244 STOP
```

The listening node will log that it received the event:

```
[debussy:]$ ./rapr event "LISTEN UDP 5000"
rapr:version 0.5.1c
mgen:version 5.01b
rapr: starting now ...
23:08:52.910484 START
19:08:52.910468 app>RAPR type>Application action>ApplicationStartUp
23:08:52.911460 LISTEN proto>UDP port>5000
19:08:52.911965 app>RAPR type>Reception action>start ubi>1560281088 \
eventSource>script_event mgenCmd>"LISTEN UDP 5000,"
23:09:07.698902 RECV proto>UDP flow>1 seq>1 src>192.168.1.103/33594 \
dst>192.168.1.103/5000 sent>23:09:07.698703 size>1024 gps>INVALID,999.\
000000,999.000000,-999 data>6:03045D10AE47
```

5. What is a RAPR script, a RAPR behavior table, and a RAPR dictionary?

For more complex applications it would be impractical to generate long sequences of messages from the command line and so a RAPR input script is available. The input script can be used to schedule RAPR events over a time line, to set up application-wide defaults via "global commands" like TXLOG, and can be used to repeat and fine tune application behavior.

The RAPR Behavior Table (sometimes known as a logic table) enables RAPR to emulate interactive applications. A node can generate messages not only according to its predefined local scripts, but also in response to messages it receives over the network. Multiple events can be scripted in the table so that a single network event can result in multiple system behaviors. The local application itself can also use the behavior table to trigger behavior(s).

A RAPR dictionary capability allows for writing simpler and cleaner scripts and tables. It provides the capability to translate xml name-value pairs that can be used to give more desriptive name in scripts and tables. Short hand notation provided by the dictionary can make more sense to the user from a functional point of view.

6. How to send a message from one node to another using RAPR scripts?

The example above can be entered via an input script. Create a `rapr.input` text file to contain the commands. Add a new command to start a second event at 5 seconds into the test and tell it to have a DURATION of 5 seconds. Specify a mgen message pattern to be used rather than the default pattern of "PERIODIC [1 1024]". (See rapr-defaults for more details about attribute defaults). The second declarative event will send 10 512-byte packets at exponentially-distributed intervals for 5 seconds.

Be sure to enter a linefeed after the last entry in the file:

```
# The TXLOG enables message transmission logging
TXLOG

# The DECLARATIVE command directs rapr to send traffic to a destination:
0.0 DECLARATIVE UDP DST 192.168.1.103/5000

5.0 DURATION 5.0 DECLARATIVE UDP DST 192.168.1.103/5000 POISSON [10 512]
```

To tell RAPR to get its commands from the input script use the input command:

```
./rapr input rapr.input
```

The next example sets up multiple events scheduled to start at five seconds into the test and run for 30 seconds. (Note that you may need to remove the "\" character if you are copying and pasting these examples.) This traffic pattern might emulate a video stream:

```
# The TXLOG enables message transmission logging
TXLOG

# The DECLARATIVE command directs rapr to send traffic to a destination:
DECLARATIVE UDP DST 192.168.1.103/5000

5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 POISSON [10 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 BURST \
[REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [30 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [15 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [5 1400]
```

Notice that in the first declarative command, no DURATION time was specified. The system will provide internal defaults for certain event attributes that are not specified. The simplest DECLARATIVE sytnax that can be used is:

```
DECLARATIVE UDP DST 192.168.1.103/5000
```

It will be "translated" to the following syntax when the internal defaults are applied:

```
0.0 DURATION 0.99 UDP SRC <sytstemProvided> DST 192.168.1.103/5000 \
PERIODIC [1 1024] COUNT 1
```

If no start time is specified, the event will start immediately (t 0.0). The duration will default to 0.99 seconds, meaning the event will send as many packets as specified by the traffic pattern within this window. The SRC port will be randomly selected by the operating system. The default mgen pattern "PERIODIC [1 1024]" means send one 1024 byte packet a second at a periodic interval. The COUNT attribute specifies that only <n> messages be sent, in this case 1 message. This attribute takes precedence over the duration or mgen pattern attributes.

Another DECLARATIVE attribute that might be useful to mention here is the STOP attribute.

```
0.0 STOP 5.0 DECLARATIVE UDP DST 192.168.1.103/5000
```

This command will STOP the declarative command at 5 seconds relative to the *test time line*, not the start of the behavior event. Because the pattern will default to "PERIODIC [1 1024]", five packets will be sent at five minute intervals. See the RAPR documentation for more information on the relationship between the DURATION STOP and COUNT attributes.

7. Using the dictionary to simpify scripting

The dictionary capability can be used to simplify script writing. It translates XML name value pairs used by the application and scripts. Anything enclosed by "%" in a command will be translated into the field value in the dictionary. In a text editor, create a dictionary.xml file containing the following:

```
<RaprDictionary>
  <namespace>
     <label>DEFAULT</label>
     <item>
       <field>Video-Server</field>
       <value>192.168.1.103/5000</value>
     </item>
  </namespace>
</RaprDictionary>
```

The dictionary can also translate one keyword into multiple values. If multiple values exist, each will be used to create an independent statement. Therefore, if we add a "Video-pattern" keyword to the dictionary:

```
<RaprDictionary>
  <namespace>
   <label>DEFAULT</label>
   <item>
      <field>Video-Server</field>
      <value>192.168.1.103/5000</value>
   </item>
   <item>
      <field>Video-Pattern</field>
      <value>POISSON [10 1400]</value>
      <value>BURST [REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]</value>
      <value>PERIODIC [30 1400]</value>
      <value>PERIODIC [15 1400]</value>
      <value>PERIODIC [5 1400]</value>
   </item>
  </namespace>
</RaprDictionary>
```

we can then create the video stream behavior with a single RAPR command:

```
# Load the dictionary
LOAD_DICTIONARY dictionary.xml

# The TXLOG enables message transmission logging
TXLOG

# The DECLARATIVE command directs rapr to send traffic to a destination:
DECLARATIVE UDP DST 192.168.1.103/5000

5.0 STOP 10.0 DECLARATIVE UDP DST %Video-Server% %Video-Pattern%
```

The translation process will create a new declarative event for each pattern specified for the %Video-Pattern% keyword:

```
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 POISSON \
[10 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 BURST \
[REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [30 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [15 1400]
5.0 DURATION 10.0 DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [5 1400]
```

If the video-pattern is known to always be directed to the video-server, one could simplify the command even further:

```
<RaprDictionary>
  <namespace>
   <label>DEFAULT</label>
   <item>
      <field>Video-Message</field>
      <value>Declarative UDP DST %Video-Server% %Video-Pattern%</value>
   </item>
   <item>
      <field>Video-Pattern</field>
      <value>POISSON [10 1400]</value>
      <value>BURST [REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]</value>
      <value>PERIODIC [30 1400]</value>
      <value>PERIODIC [15 1400]</value>
      <value>PERIODIC [5 1400]</value>
   </item>
  </namespace>
</RaprDictionary>
```

Now a "video message" can be scripted that will be translated into the same behavior as the other video server examples:

```
# Load the dictionary
LOAD_DICTIONARY dictionary.xml

# The TXLOG enables message transmission logging
TXLOG

# The DECLARATIVE command directs rapr to send traffic to a destination:
DECLARATIVE UDP DST 192.168.1.103/5000

5.0 STOP 10.0 %Video-message%
```

Note that this example illustrates that dictionary entries may be nested within the dictionary and that dictionary field values are case sensitive.

8. How to make a node respond to a message it receives?

A behavior table, sometimes referred to as a "logic table", is used to define how RAPR responds to "logic ids" which are used to direct application behavior. A logic id can be embedded in a message payload to direct the target node's response. Adding a LOGICID attribute will cause the specified logic id to be embedded in all messages sent by the behavior event. Create an input script "sample-send.input" that will send one message to the destination and listen for traffic on udp port 6000:

```
# The TXLOG enables message transmission logging
TXLOG

# Listen for response traffic on port 6000
LISTEN UDP 6000

# The DECLARATIVE command directs rapr to send traffic to a destination:
DECLARATIVE UDP DST 192.168.1.103/5000 LOGICID 1
```

When a RAPR application receives a message associated with a logicid, it will look it up in its behavior table and execute the associated commands. Set up an XML logic table "logictable.xml" as follows:

```
<RaprLogicTable>
  <state>
    <value>0</value>
    <logicid>
        <id>1</id>
        <entry>DECLARATIVE DST %PACKET:SRCIP%/6000</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

When the listening application receives a packet associated with logic id 1, it will invoke the associated behavior in the logic table. Notice the %PACKET:SRCIP% keyword. It is a reserved keyword that tells rapr to replace the keyword with the information contained in the incoming packet. This example will send a single packet back to port 6000 at the originating packet's source ip address.

Load the behavior table into the RAPR listening for traffic with the LOGICTABLE_FILE command in the "sample-recv.input" input script:

```
# The TXLOG enables message transmission logging
TXLOG

# Load the behavior table
LOGICTABLE_FILE logictable.xml

# Listen for udp traffic on port 5000
LISTEN UDP 5000
```

Start the listening rapr first on the node that should respond to the message:

```
./rapr input sample-recv.input
```

Then start the sending rapr on the sending node:

```
./rapr input sample-send.input
```

The triggering node will log the reception of the response packet to standard out:

```
[debussy:]$ /home/nrl/rapr/unix/rapr input /home/nrl/rapr/unix/rapr.input
rapr:version 0.5.1c
mgen:version 5.01b
rapr: starting now ...
21:49:54.909146 START
17:49:54.909142 app>RAPR type>Application action>ApplicationStartUp
21:49:54.910105 LISTEN proto>UDP port>6000
17:49:54.910361 app>RAPR type>Reception action>start ubi>1157627904 \
eventSource>script_event mgenCmd>"LISTEN UDP 6000,"
17:49:54.913622 app>RAPR type>Declarative action>start ubi>1157627905 \
eventSource>script_event mgenCmd>"ON 1 UDP SRC 0 DST 192.168.1.103/5000 \
PERIODIC [1 1024 ] DATA [0204010000000304921D0A10] COUNT 1"
21:49:54.914248 SEND proto>UDP flow>1 seq>1 srcPort>33681 dst>192.168.1.103/\
5000 size>1024
21:49:54.923863 RECV proto>UDP flow>1 seq>1 src>192.168.1.102/33682 \
dst>192.168.1.103/6000 sent>21:49:54.923576 size>1024 gps>INVALID,999.\
000000,999.000000,-999 data>6:03043237853B
17:49:55.904756 app>RAPR type>Declarative action>timeout ubi>1157627905 \
0.001 OFF 1
```

9. Setting up a "three-way handshake"

In this example communication will take place amongst three nodes.

- Node 1 will send a packet to Node 2
- This will trigger Node 2 to send a packet to node 3 after 5 seconds
- Node 3 will ack the that it received the packet from node 2

The dictionary can be used to set up short hand notation for the node names and to give meaningful names for the "logic id's". The same dictionary can be used on all the nodes:

```
<RaprDictionary>
  <namespace>
   <label>DEFAULT</label>
   <item>
      <field>TN-1</field>
      <value>192.168.1.101</value>
   </item>
   <item>
      <field>TN-2</field>
      <value>192.168.1.102</value>
   </item>
   <item>
      <field>TN-3</field>
      <value>192.168.1.103</value>
   </item>
   <item>
      <field>Request</field>
      <value>LOGICID 1</value>
   </item>
   <item>
      <field>RequestACK</field>
      <value>LOGICID 2</value>
   </item>
   <-- Define a commonly used behavior event as "ACK" -->
   <item>
      <field>ACK</field>
      <value>DECLARATIVE DST %PACKET:SRCIP%/6000 PERIODIC [1 28]</value>
   </item>
  </namespace>
</RaprDictionary>
```

Note that we also define "ACK" as a short hand notation for defining the entire acknowledgement response behavior event definition. We have specified that the "ACK" packet (and the other packets in this example) should be 28 bytes, the minimum UDP IPv4 packet size.

Create a logic table to implement the behavior that each logic id should be associated with. Note that entries in the logic table must correspond to the numeric logicid *value* in the dictionary but that the *entry*'s may contain dictionary entries.

```
<RaprLogicTable>
  <state>
    <value>0</value>
    <logicid>
        <-- 5 seconds after processing the logicId start a -->
        <-- declarative event with a "request ack" logicid -->
        <id>1</id>
        <entry>5.0 DECLARATIVE DST %TN-3%/5000 PERIODIC [1 28] %RequestACK%</entry>
    </logicid>
    <logicid>
        <-- Send an "ack" response -->
        <id>2</id>
        <entry>%ACK%</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

The first node's (TN-1) input script directs RAPR to send a single packet to the second node (TN-2) and puts a REQUEST logic id in the payload. It loads the common dictionary used by all three nodes.

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary.xml

# Send a packet to TN-2 with a REQUEST logicid
DECLARATIVE UDP DST %TN-2%/5000 PERIODIC [1 28] %REQUEST%
```

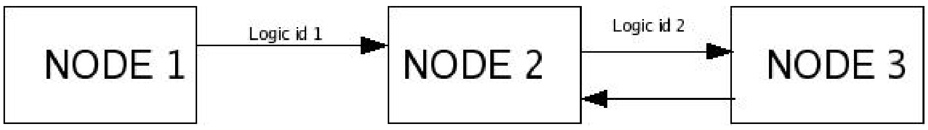
The second node loads the common dictionary, the logictable, and listens to port UDP 5000.

```
LOAD_DICTIONARY dictionary.xml
LOGICTABLE_FILE logictable.xml

LISTEN UDP 5000
```

The third node also loads the dictionary, the logictable, and listens to port UDP 5000.

```
LOAD_DICTIONARY dictionary.xml
LOGICTABLE_FILE logictable.xml

LISTEN UDP 5000
```

The RAPR's on nodes 2 and 3 should be started before the RAPR on node 1 so they are sure to be listening on UDP port 5000 when node 1 initiates the three way handshake. When node 1 sends the first packet to node 2, the "application" being emulated will behave as follows:



10. Creating random system behavior and complex responses

The next example illustrates how to:

- define multiple responses to a single message
- define probabilistic responses
- define probabilistic behaviors

Rapr system behavior can be probabilistic. The system keywords RANDOMI and RANDOMF are used to return random integers or floats within a specified range. The keywords should be referenced in the SYSTEM dictionary namespace. For example, the following command specifies that a behavior event send a packet between 2048 and 8192 bytes.

```
DECLARATIVE UDP DST 192.168.1.103/5000 PERIODIC [1 %SYSTEM:RANDOMI(2048,8192)%]
```

Create a dictionary that associates readable names with the logic id's we will use in the example:

```
<RaprDictionary>
  <namespace>
   <label>DEFAULT</label>
   <item>
      <field>SampleServer</field>
      <value>192.168.1.103/6000</value>
   </item>
   <item>
      <field>Complex-Response-Reply</field>
      <value>LOGICID 1</value>
   </item>
   <item>
      <field>Random-Reply</field>
      <value>LOGICID 2</value>
   </item>
   <item>
      <field>Short-Reply</field>
      <value>LOGICID 3</value>
   </item>
   <item>
      <field>Long-Reply</field>
      <value>LOGICID 4</value>
   </item>
   <item>
      <field>Delayed-Reply</field>
      <value>LOGICID 5</value>
   </item>
  </namespace>
</RaprDictionary>
```

To define multiple responses to a given logic id, simply create multiple entries for the id in the logic table. A logic id of "1" in this example will trigger a multi-event "video-stream" response as seen in the "video-server" example.

```
<RaprLogicTable>
  <state>
    <value>0,0</value>
    <logicid>
        <-- Complex Response Reply -->
        <id>1</id>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  POISSON [10 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/5000 \
                  BURST [REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [30 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [15 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [5 1400]</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

The optional **<percent>** logicid child element defines the probability of triggering the event(s) associated with the logicid. The value should be a float between 0 and 1, 1 meaning always do the behavior. In the following example, the declarative event defined for the "response reply" logic id has a 50% probablity of occuring:

```
<RaprLogicTable>
  <state>
    <value>0,0</value>
    <logicid>
        <-- Complex Response Reply -->
        <id>1</id>
        <percent>0.5</percent>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  POISSON [10 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/5000 \
                  BURST [REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [30 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [15 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [5 1400]</entry>
    </logicid>
    <logicid>
        <-- Random Reply -->
        <id>2</id>
        <percent>0.5</percent>
        <entry>DECLARATIVE UDP DST 192.168.1.103/6000</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

The start and duration of a behavior events can be randomized as seen in the "short reply", "long reply", and delayed reply responses:

```
<RaprLogicTable>
  <state>
    <value>0,0</value>
    <logicid>
        <-- Complex Response Reply -->
        <id>1</id>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  POISSON [10 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/5000 \
                  BURST [REGULAR 2.0 PERIODIC [40 1400] FIXED 0.1]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [30 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [15 1400]</entry>
        <entry>DURATION 5.0 DECLAR UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT% \
                  PERIODIC [5 1400]</entry>
   </logicid>
   <logicid>
        <-- Random Reply -->
        <id>2</id>
        <percent>0.5</percent>
        <entry>DECLARATIVE UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
   </logicid>
   <logicid>
        <-- Short Reply -->
        <id>3</id>
        <entry>DURATION %SYSTEM:RANDOMI(1,3)% DECLARATIVE UDP
                  DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
    <logicid>
        <-- Long Reply -->
        <id>4</id>
        <entry>DURATION %SYSTEM:RANDOMI(15,20)% DECLARATIVE UDP
                  DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
   <logicid>
        <-- Delayed Reply -->
        <id>5</id>
        <entry>%SYSTEM:RANDOMI(5,10)% DURATION 5 DECLARATIVE UDP
                 DST %PACKET:SRCIP%/%PACThenKET:SRCPORT%</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

Use the event command to tell the target node to load the dictionary and logictable and listen for traffic on port 600:

```
./rapr event "LOAD_DICTIONARY dictionary.xml" event "LOGICTABLE_FILE \
logictable.xml" event "LISTEN UDP 6000"
```

Practice triggering different system behaviors (and testing your logic table definitions) by sending the various logic id's to the listening node, e.g.:

```
./rapr event "DECLARATIVE UDP DST 192.168.1.103/6000 LOGICID 5" txlog
```

11. What is a RAPR Interrogative Behavior Object

The interrogative behavior event is used to emulate the basic elements of a transaction. It will send a message and "listen" for a response. If a response is not received from the destination within a timeout interval, the interrogative object will send another message. You can test the interrogative behavior by configuring one node to listen for data (load the logictable from the previous example so the definition of logicid "3" is available):

```
./rapr event "LISTEN UDP 5000" event "LOGICTABLE_FILE logictable.xml"
```

Then use the event command to send an interrogative event with a logic id of 3 (short-reply):

```
./rapr event "LISTEN UDP 6000" event "INTERROGATIVE UDP SRC 6000 \
DST 192.168.1.103/5000 LOGICID 3" txlog
```

As seen in the log on the listening node:

```
[debussy:]$ ./rapr event "LISTEN UDP 5000" event "LOGICTABLE_FILE \
logictable.xml" txlog
rapr:version 0.5.1c
mgen:version 5.01b
rapr: starting now ...
15:45:35.243108 START
11:45:35.243103 app>RAPR type>Application action>ApplicationStartUp
15:45:35.244232 LISTEN proto>UDP port>5000
11:45:35.244485 app>RAPR type>Reception action>start ubi>4060086272 \
eventSource>script_event mgenCmd>"LISTEN UDP 5000,"
11:45:35.244681 app>RAPR type>RaprEvent action>loading_logictable \
name>logictable.xml
15:45:40.278489 RECV proto>UDP flow>1 seq>1 src>192.168.1.102/6000 \
dst>192.168.1.103/5000 sent>15:45:40.278267 size>1024 gps>INVALID,\
999.000000,999.000000,-999 data>18:0104010000730204030000000304CBE4A304
11:45:40.281894 app>RAPR type>Declarative action>start ubi>4060086274 \
eventSource>net_event mgenCmd>"ON 1 UDP SRC 0 DST 192.168.1.102/6000 \
PERIODIC [1 1024 ] DATA [0304B0F08102040401000073]"
15:45:40.282034 SEND proto>UDP flow>1 seq>1 srcPort>33923 \
dst>192.168.1.103/6000 size>1024
15:45:41.282665 SEND proto>UDP flow>1 seq>2 srcPort>33923 \
dst>192.168.1.103/6000 size>1024
15:45:42.282470 SEND proto>UDP flow>1 seq>3 srcPort>33923 \
dst>192.168.1.103/6000 size>1024
11:45:42.282772 app>RAPR type>Declarative action>timeout ubi>4060086274 \
0.001 OFF 1
```

when the listening node receives a message with a logic id of "3" in the payload (highlighted in the data field, see the rapr user's guide for more details on interpreting payload fields), the listening rapr will look up the behavior in the logic table:

```
   <logicid>
        <-- Short Reply -->
        <id>3</id>
        <entry>DURATION %SYSTEM:RANDOMI(1,3)% DECLARATIVE UDP
                  DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
```

and send a packet stream lasting between 1 and 3 seconds to the source ip/port address of the incoming message. Notice that the declarative event logs that its event source was a "net_event" (as opposed to a script_event).

The triggering node's log file:

```
[debussy:]$ ./rapr event "LISTEN UDP 6000" event "INTERROGATIVE UDP SRC 6000 \
DST 192.168.1.103/5000 LOGICID 3" txlog
rapr:version 0.5.1c
```

```
mgen:version 5.01b
rapr: starting now ...
15:45:40.272944 START
11:45:40.272938 app>RAPR type>Application action>ApplicationStartUp
15:45:40.273908 LISTEN proto>UDP port>6000
11:45:40.274153 app>RAPR type>Reception action>start ubi>1929379840 \
eventSource>script_event mgenCmd>"LISTEN UDP 6000,"
11:45:40.277483 app>RAPR type>Interrogative action>start ubi>1929379841 \
eventSource>script_event cnt>0 mgenCmd>"ON 1 UDP SRC 6000 DST \
192.168.1.103/5000 PERIODIC [1 1024 ] DATA [010401000073020403000000\
0304CBE4A304]"
11:45:40.277982 app>RAPR type>Interrogative action>off_event ubi>1929379841\
eventSource>script_event cnt>0 mgenCmd>"0.990000 OFF 1"
15:45:40.278267 SEND proto>UDP flow>1 seq>1 srcPort>6000 \
dst>192.168.1.103/5000 size>1024
15:45:40.284519 RECV proto>UDP flow>1 seq>1 src>192.168.1.103/33923 \
dst>192.168.1.102/6000 sent>15:45:40.282034 size>1024 gps>INVALID,\
999.000000,999.000000,-999 data>12:0304B0F08102040401000073
11:45:40.285008 app>RAPR type>Interrogative action>success ubi>1929379841
15:45:41.282968 RECV proto>UDP flow>1 seq>2 src>192.168.1.103/33923 \
dst>192.168.1.102/6000 sent>15:45:41.282665 size>1024 gps>INVALID,999.\
000000,999.000000,-999 data>12:0304B0F08102040401000073
15:45:42.282938 RECV proto>UDP flow>1 q>3 src>192.168.1.103/33923 \
dst>192.168.1.102/6000 sent>15:45:42.282470 size>1024 gps>INVALID,999.\
000000,999.000000,-999 data>12:0304B0F08102040401000073
```

indicates that the triggering node sent a single packet to the listening node with logic id "3". Upon receipt of the first response message (flow>1 seq>1) from the target node, the transaction is considered "successful" and no further triggers are sent.

How the interrogative event associates the incoming message with it's original query is discussed in the next section "What is a UBI?"

12. What is a UBI?

Notice the UBI field in the logged interrogative event:

```
11:45:40.277483 app>RAPR type>Interrogative action>start ubi>1929379841 \
eventSource>script_event cnt>0 mgenCmd>"ON 1 UDP SRC 6000 DST \
192.168.1.103/5000 PERIODIC [1 1024 ] DATA [010401000073020403000000\
0304CBE4A304]"
```

Each behavior event created by RAPR is associated with a Unique Behavior Identifier (UBI). It is embedded in the payload of messages sent by interrogative events. (In the above example 0104-**01000073** is the hex equivelent of ubi 1929379841). RAPR embeds this "foreign ubi" in any responses it makes to the message:

```
15:45:40.284519 RECV proto>UDP flow>1 seq>1 src>192.168.1.103/33923 \
dst>192.168.1.102/6000 sent>15:45:40.282034 size>1024 gps>INVALID,\
999.000000,999.000000,-999 data>12:0304B0F08102040401000073
```

The RAPR receiving this response packet then looks for the interrogative object with this "foreign ubi" that is waiting for a reply. If it finds one, the "transaction" is considered successful and no further messages are sent.

Use of the UBI field is discussed in more detail in the section "How to keep track of state information".

13. How to make a node retransmit a request message if it fails to receive a reply?

As noted earlier, when an interrogative object does not receive a response from it's target, it will send another message. The default behavior of the interrogative object is to send 3 retries at 10 second intervals if a reply is not received. Additional attributes are available to the interrogative object to override these defaults. Create a new logictable that defines logic id "1" as follows (after 5 seconds send a declarative message back to the packet source ip/port):

```
<RaprLogicTable>
  <state>
    <value>0</value>
    <logicid>
        <id>1</id>
        <entry>5.0 DECLARATIVE DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

Start rapr listening to udp port 5000 using the new logic table:

```
./rapr event "LISTEN UDP 5000" event "LOGICTABLE_FILE logictable.xml"
```

Create a new input script for this example. Start an interrogative event that changes the default interrogative object behavior. Use the RETRYINTERVAL attribute. It specifies that if a response is not received within 3 seconds send another message. The NUMRETRIES attribute specifies that 5 retries be made.

```
# The TXLOG enables message transmission logging
TXLOG

# Listen to UDP port 6000
LISTEN UDP 6000

# Start an interrogative object
INTERROGATIVE RETRYINTERVAL 3 NUMRETRIES 5 UDP SRC 6000 DST 192.168.1.103/5000\
 LOGICID 1
```

Because the response message is sent after a 5 second delay, the 3 second retry interval of the interrogative object expires before any response is received and a second message is sent. Notice that the interrogative message succeeds after receiving the first message. Notice also that the second message it sent ALSO triggered a response message from the target node. Because the interrogative messages has already succeeded, the second message has no effect on the system behavior.

14. What is "state" in the logic table?

As seen in previous examples of the logic table, all logic id's are enclosed within a "state" block:

```
  <state>
    <value>0</value>
    <logicid>
        <id>1</id>
        <entry>5.0 DECLARATIVE DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
  </state>
```

System state can be used to change the overall behavior of the system. For example an "application" in a "starting up" state might not respond to any network requests. In this state there would be no entries in the "starting up" state block. The dictionary can be used to give meaningful names to system state:

```
   <item>
      <field>STARTING_UP</field>
      <value>0</value>
   </item>

  <state>
    <value>%STARTING_UP%</value>
    <logicid>
        <id>1</id>
        <entry>5.0 DECLARATIVE DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
  </state>
```

By default the system starts in state "0" so behavior that should occur in the default system state should be defined in that state block.

To practice changing system state, set up a logictable and dictionary as follows:

```
<RaprDictionary>
  <namespace>
   <label>DEFAULT</label>
   <item>
      <field>DO_NOTHING</field>
      <value>0</value>
   </item>
   <item>
      <field>RESPOND</field>
      <value>1</value>
   </item>
  </namespace>
</RaprDictionary>

<RaprLogicTable>
  <state>
    <-- DO_NOTHING is defined as "0" in a dictionary -->
    <value>%DO_NOTHING%</value>
    <logicid>
        <id>1</id>
            <-- Use the CHANGE_STATE rapr event to change system state -->
            <entry>CHANGE_STATE 1</entry>
    </logicid>
  </state>
  <state>
   <-- RESPOND is defined as "1" in a dictionary -->
   <value>%RESPOND%</value>
   <logicid>
        <-- Reply -->
        <id>1</id>
        <entry>DECLARATIVE UDP DST %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

Start rapr listening to udp port 5000 using the new logic table and dictionary:

```
./rapr event "LISTEN UDP 5000" event "LOGICTABLE_FILE logictable.xml" \
event "LOAD_DICTIONARY dictionary.xml"
```

Load the following input script into RAPR using the event command:

```
# The TXLOG enables message transmission logging
TXLOG

# Listen to UDP port 6000
LISTEN UDP 6000

# Start an interrogative object
INTERROGATIVE RETRYINTERVAL 3 NUMRETRIES 5 UDP SRC 6000 DST 192.168.1.103/5000\
 LOGICID 1
```

Notice that the first message does not illicit a reply. This is because the system was in the default system state of "DONT_REPLY" and the behavior for logic id "1" is a CHANGE_STATE rapr event, not a "response" behavior event. Because no response was received by the interrogative event, the request message is sent again. This time the system state has been changed to "RESPONSE" and the behavior for logic id "1" in this system state is a response message.

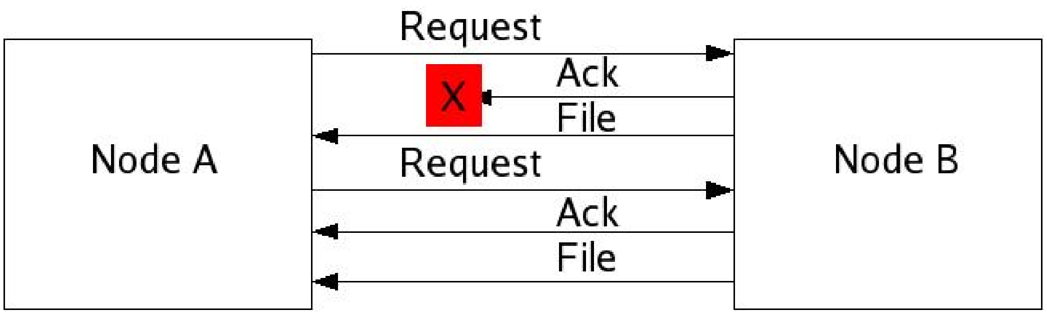
15. How to emulate "transaction based" behavior using UBI state.

Emulating more complex system behaviors sometimes requires maintaining "transaction based" state, as in the case where what is considered a single "transaction" may require multiple network messages that should be treated in a coherent and reliable way independent of other transactions. In this case, the system needs to maintain "transaction state". In RAPR, this is accomplished by using UBI state.

As discussed above, each behavior event is associated with a "unique behavior identifier" or UBI that is embedded in the payload of interrogative messages and associated response messages. The system can be directed to change its behavior as it receives messages associated with a given UBI. An example may best illustrate this capability. Let's say we want to emulate the following behavior:

- Node A should send a request to Node B requesting a TCP file
- Node B should ACK the first request, and schedule that a large TCP file be sent within 3 seconds.
- Multiple requests from Node A should not result in more than one TCP file being sent. Node B should simply ACK any subsequent requests.

The third point is an important consideration in a high packet loss environment. As illustrated below, if Node A does not receive the first response acknowledgement, it will send another request to Node B. Without using transaction state that causes Node B to "remember" that it has already received the first request, Node B will schedule the transmission of two TCP files if it receives two requests.



"Transaction state" or "ubi state" is implemented by using a UBI state block. As illustrated already "system state" is defined by the value of the state xml tag. UBI state adds another dimension to the state value. Note that UBI state definitions should be defined in a separate state block.

```
<RaprLogicTable>
  <state>
     <value>0</value>   <- System state 0
  </state>

  <state>
     <value>0,1</value> <- System state 0, UBI state 1
  </state>
</RaprLogicTable>
```

Changing UBI state works alot like changing system state. A CHANGE_UBI_STATE command will cause RAPR to look up subsequent logicid's in a separate part of the state table that is associated with the packet's UBI. In this example, the first packet will be looked up in the the default system state block (**System state 0**). Logic id "1" in this state tells RAPR to remember the UBI and change the state for that ubi to **UBI state 1**. A second packet with the same UBI and logic id 1 will then be looked up in **UBI state 1** which changes the state again to **UBI state 2**. The third packet with the same UBI and logic id 1 will result in a response:

```
<RaprLogicTable>
  <state>
     <value>0</value>   <- System state 0
     <logicid>
```

```
        <id>1</id>
        <entry>CHANGE_UBI_STATE %PACKET:UBI% 1</entry>
     </logicid>
  </state>
  <state>
     <value>0,1</value> <- UBI state 1
     <logicid>
        <id>1</id>
        <entry>CHANGE_UBI_STATE %PACKET:UBI% 2</entry>
     </logicid>
  </state>
  <state>
     <value>0,2</value> <- UBI state 2
     <logicid>
        <id>1</id>
        <entry>DECLARATIVE UDP %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
     </logicid>
  </state>
</RaprLogicTable>
```

If multiple system states are defined, the UBI state will be looked up in the UBI state definition for the current system state. E.g. If the system is in state "1" a CHANGE_UBI_STATE command will lookup the UBI state defined for state **<value>1,1</value>**.

```
<RaprLogicTable>
  <state>
    <value>0</value>
  </state>
  <state>
    <value>1</value>
    <logicid>
      <id>1</id>
      <entry>CHANGE_UBI_STATE %PACKET:UBI% 1</entry>
    </logicid>
  </state>
  <state>
     <value>1,1</value> <- System state 1, UBI state 1
     <logicid>
        <id>1</id>
        <entry>DECLARATIVE UDP %PACKET:SRCIP%/%PACKET:SRCPORT%</entry>
     </logicid>
  </state>
</RaprLogicTable>
```

To implement the example [15] set up a logic table as follows:

```
<RaprLogicTable>
  <state>
    <!-- Default state -->
    <value>0</value>
    <logicid>
        <id>1</id>
            <!-- Change the state for this "transaction" to "event queued" -->
            <entry>CHANGE_UBI_STATE %PACKET:UBI% %EVENT_QUEUED%</entry>
            <!-- Ack the request -->
            <entry>%ACK%</entry>
            <!-- Schedule the transmission a 1 MB "TCP file" -->
            <entry>%BIG-TCP-FILE%</entry>
    </logicid>
  </state>
  <state>
    <!-- Default state, UBI state EVENT_QUEUED -->
    <value>0,1</value>
    <logicid>
        <!-- Event already queued, Only ACK retry -->
        <id>1</id>
              <entry>%ACK%</entry>
    </logicid>
  </state>
</RaprLogicTable>
```

The corresponding dictionary is:

```
<RaprDictionary>
  <namespace>
    <label>DEFAULT</label>
    <item>
      <field>SERVER</field>
      <value>192.168.1.102</value>
   </item>
   <item>
      <field>SERVER-PORT</field>
      <value>7000</value>
   </item>
   <item>
      <field>TCP-PORT</field>
      <value>6000</value>
   </item>
   <item>
      <field>ACK-PORT</field>
      <value>5000</value>
   </item>
   <item>
      <field>ACK</field>
      <value>DECLARATIVE UDP DST %PACKET:SRCIP%/%ACK-PORT%</value>
   </item>
   <item>
      <field>EVENT_QUEUED</field>
      <value>1</value>
   </item>
   <item>
      <field>BIG-TCP-FILE</field>
      <value>5.0 DECLARATIVE TCP DST %PACKET:SRCIP%/%TCP-PORT% \
               PERIODIC [1 1048576] COUNT 1</value>
   </item>
  </namespace>
</RaprDictionary>
```

On the "server" start a RAPR with the following input script:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary.xml
LOGICTABLE_FILE logictable.xml

LISTEN UDP %SERVER-PORT%
```

On the "client " start a RAPR with the following input script:

```
# The TXLOG enables message transmission logging
TXLOG
LOGICTABLE logictable.xml
LOAD_DICTIONARY dictionary.xml

# Listen to the ACK port
LISTEN UDP %ACK-PORT%
# Listen to the TCP port
LISTEN TCP %TCP-PORT%

# Start an interrogative object
INTERROGATIVE UDP DST %SERVER%/%SERVER-PORT% LOGICID 1
```

Notice that the "server" changes ubi state, "acks" the message, and schedules the transmission of a TCP file. When then client gets the "ack" it will not send any further requests however. To "fake" a packet loss condition, add a RETRYINTERVAL of 2 to the interrogative object. Modify the "ACK" behavior event to send to a port the client is not listening on. Notice that only 1 large tcp message is sent, while every request is "acked".

16. What are "namespaces" in the dictionary used for?

So far we have used dictionaries where all the definitions have been in the "DEFAULT" xml namespace. For more complex applications it might be useful to specify multiple namespaces. This will allow us to use the same dictionary field names that have different values defined in different namespaces. For example we might want to define common identifying attributes like "IP" for all system nodes, but be able to differentiate the specific value depending on the node role (e.g. client, server). If we set up a "client" namespace and a "server" namespace we could specify %SERVER:IP% or %CLIENT:IP% depending on the node's role. For example:

```
<RaprDictionary>
   <namespace>
      <label>SERVER</label>
      <!-- Entries Associated with a Server -->
      <item>
      <!-- Server IP Address -->
         <field>IP</field>
         <value>192.168.1.102</value>
      </item>
   </namespace>
   <namespace>
      <label>CLIENT</label>
      <!-- Entries Associated with a Client -->
      <item>
      <!-- Client IP Address-->
         <field>IP</field>
         <value>192.168.1.103</value>
      </item>
   </namespace>
</RaprDictionary>
```

17. How can I emulate an HTTP-like client server application?

This example will walk through how to set up rapr to emulate simplistic HTTP-like client server behavior, including:

- An HTTP web server that will respond to client requests
- HTTP clients that will request "web pages" from the HTTP server
- The client requests will be sent at exponentially distributed 10 minute intervals
- Client requests will be statistically varied
- The server responses will be statistically varied
- The server will be "shutdown" for a period in which it will not respond to any client requests

In this example we'll be using dictionary "namespaces". Create the following dictionary entries specifying a server namespace:

```
<RaprDictionary>
   <namespace>
   <label>DEFAULT</label>
   <item>
       <field>HTTP_PORT</field>
       <value>8000</value>
   </item>
   </namespace>
   <namespace>
   <label>SERVER</label>
       <item>
           <field>IP</field>
           <value>192.168.1.103</value>
       </item>
   </namespace>
</RaprDictionary>
```

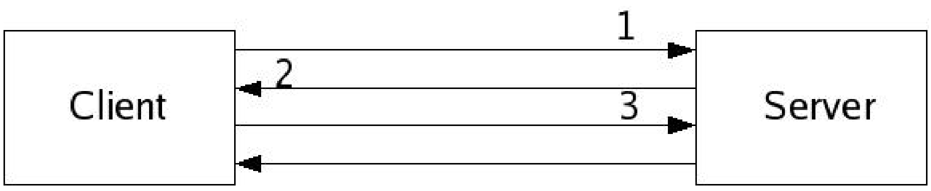
The HTTP server input script is fairly straight-forward since it is simply listening for requests. Notice the new keywords "OVERWRITE_MGENLOG" and "OVERWRITE_RAPRLOG". These directives cause RAPR to direct logging data to the specified files rather than sending it to STDOUT. The mgen log file will log mgen messaging data, the rapr log file logs rapr event data. Notice that file names may be either fully qualified or relative to the RAPR binary directory.

```
# The TXLOG enables message transmission logging
TXLOG

OVERWRITE_MGENLOG mgen-http.log
OVERWRITE_RAPRLOG rapr-http.log

LOAD_DICTIONARY dictionary-http.xml
LOGICTABLE_FILE logictable-http.xml

# Listen for HTTP requests
LISTEN TCP %HTTP_PORT%
```

Now create a client script that sends a request to the server. Notice that we commented out creating the log files for now:

```
# The TXLOG enables message transmission logging
TXLOG

#OVERWRITE_MGENLOG mgen-http.log
#OVERWRITE_RAPRLOG rapr-http.log

LOAD_DICTIONARY dictionary-http.xml
LOGICTABLE_FILE logictable-http.xml

# Listen for HTTP requests
LISTEN TCP %HTTP_PORT%

DECLARATIVE TCP DST %SERVER:IP%/%HTTP_PORT% LOGICID 1
```

Finally, create a logic table to define the system's behavior. When building more complicated applications it is easier set things up incrementally, so to begin we'll set up a typical HTTP "transaction". The client will send a request (e.g. a URL request) to the server. The server will respond with a status line, such as "HTTP/1.1 200 OK" and the body of the file requested by the client. The client will then make another request (for a gif file, for example) that the server will return. We'll use logic id's 1 through 3 to implement this behavior:



In it's simplest form, this behavior can be implemented as follows:

```
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>
      <logicid>
          <id>1</id>
          <entry>DECLARATIVE TCP SRC 5001 DST %PACKET:SRCIP%/
                 %HTTP_PORT% LOGICID 2</entry>
      </logicid>
      <logicid>
          <id>2</id>
          <entry>DECLARATIVE TCP SRC 5001 DST %PACKET:SRCIP%/
                 %HTTP_PORT% LOGICID 3</entry>
      </logicid>
      <logicid>
          <id>3</id>
          <entry>DECLARATIVE TCP SRC 5001 DST %PACKET:SRCIP%/
                 %HTTP_PORT%</entry>
      </logicid>
   </state>
</RaprLogicTable>
```

Test the application behavior and verify that each client sends and receives two messages. Next modify the messages to reflect more realistic traffic patterns. Make the first request message vary in size from 76 - 2048 bytes. Add the COUNT attribute to ensure that just one request message is sent:

```
# The TXLOG enables message transmission logging
TXLOG

#OVERWRITE_MGENLOG mgen-http.log
#OVERWRITE_RAPRLOG rapr-http.log

LOAD_DICTIONARY dictionary-http.xml
LOGICTABLE_FILE logictable-http.xml

# Listen for HTTP requests
LISTEN TCP %HTTP_PORT%

DECLARATIVE TCP DST SRC 5000 %SERVER:IP%/%HTTP_PORT% PERIODIC \
[1 %SYSTEM:RANDOMI(76,2048)%] LOGICID 1 COUNT 1
```

Make the logic table events more realistic as well by adding randomness to the response patterns. The behavior to emulate is as follows:

- the server will reply to the "uri request" with a message between 76 and 2048 bytes
- the client has a 50 percent probability of requesting additional files, gifs or jpegs for example
- if the client responds it will request between 1 and 3 additional files. The requests will be from 76 and 2048 bytes
- the server will return a randomly sized tcp message in response to each request

It is the <**percent**> attribute that configures the client to have a 50 percent probability of sending additional requests. Specifying the **%SYSTEM:RANDOMI(1,3)%** as the pattern interval will cause the client to request from 1 to 3 additional files from the server, emulating a client requesting additional graphic file. When the server receives these requests it will reply to each message.

```
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>
      <logicid>
          <id>1</id>
          <entry>DECLARATIVE TCP SRC 5001 DST %PACKET:SRCIP%/
                 %HTTP_PORT% PERIODIC
                 [1 %SYSTEM:RANDOMI(76,2048)%] LOGICID 2</entry>
      </logicid>
      <logicid>
          <id>2</id>
          <percent>0.5</percent>
          <entry>DECLARATIVE TCP SRC 5001 DST
                 %PACKET:SRCIP%/%HTTP_PORT%
                 POISSON [1 %SYSTEM:RANDOMI(76,2048)%]
                 LOGICID 3 COUNT %SYSTEM:RANDOMI(1,3)%</entry>
      </logicid>
      <logicid>
          <id>3</id>
          <entry>DECLARATIVE TCP SRC 5001 DST %PACKET:SRCIP%/
                 %HTTP_PORT% POISSON
                 [1 %SYSTEM:RANDOMI(512,38192)%] COUNT 1</entry>
      </logicid>
   </state>
</RaprLogicTable>
```

To make this behavior occur at regularly schedule intervals, use the PERIODIC object type. This will cause the behavior to be repeated at specified intervals. In this example, the declarative event discussed above will be sent every 2 to 5 minutes, triggering the randomized request response behavior. The behavior will be repeated for an hour, as defined by the DURATION 3600 attribute.

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-http.xml
LOGICTABLE_FILE logictable-http.xml
#OVERWRITE_MGENLOG mgen-http.log
#OVERWRITE_RAPRLOG rapr-http.log

# Listen for HTTP requests
LISTEN TCP %HTTP_PORT%

DURATION 3600 PERIODIC INTERVAL %SYSTEM:RANDOMI(150,300)% \
DECLARATIVE TCP DST %SERVER:IP%/%HTTP_PORT% PERIODIC \
[1 %SYSTEM:RANDOMI(76,2048)%] LOGICID 1 COUNT 1
```

To emulate the server "shutting down", we'll use the CHANGE_STATE command. Add the following change state commands to the server's input script:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary.xml

LOAD_DICTIONARY dictionary-http.xml
LOGICTABLE_FILE logictable-http.xml

#OVERWRITE_MGENLOG mgen-http.log
#OVERWRITE_RAPRLOG rapr-http.log

# Listen for HTTP requests
LISTEN TCP %HTTP_PORT%

# Shutdown the server 10 minutes into the test
600.0 CHANGE_STATE %SERVER:STOPPED%

# Start the server back up 15 minutes later, at 25 minutes into the test
1500.0 CHANGE_STATE %SERVER:STARTED%
```

Define STOPPED and STARTED in the server namespace section of the dictionary:

```
<RaprDictionary>
   <namespace>
   <label>DEFAULT</label>
   <item>
       <field>HTTP_PORT</field>
       <value>8000</value>
   </item>
   </namespace>
   <namespace>
   <label>SERVER</label>
       <item>
           <field>IP</field>
           <value>192.168.1.103</value>
       </item>
       <item>
           <field>STARTED</field>
           <value>0</value>
       </item>
       <item>
           <field>STOPPED</field>
           <value>1</value>
       </item>
   </namespace>
</RaprDictionary>
```

Now set up the associated states in the logic table. We've already defined the "STARTED" state block, the default state of 0. Add a new state block for the "STOPPED" system state. Note that a null logicid should be defined if there are no other logicid entries in the state block.

```
<RaprLogicTable>
   <state>
       [snip]
   </state>
   <state>
   <value>1</value>
      <logicid>
          <id>0</id>
          <entry></entry>
      </logicid>
   </state>
</RaprLogicTable>
```

Notice that the server will not respond to any received messages when it changes to the "STOPPED" state ten minutes into the test. (You may wish to shorten the timeout intervals of the periodic object and state changes while testing).

Note

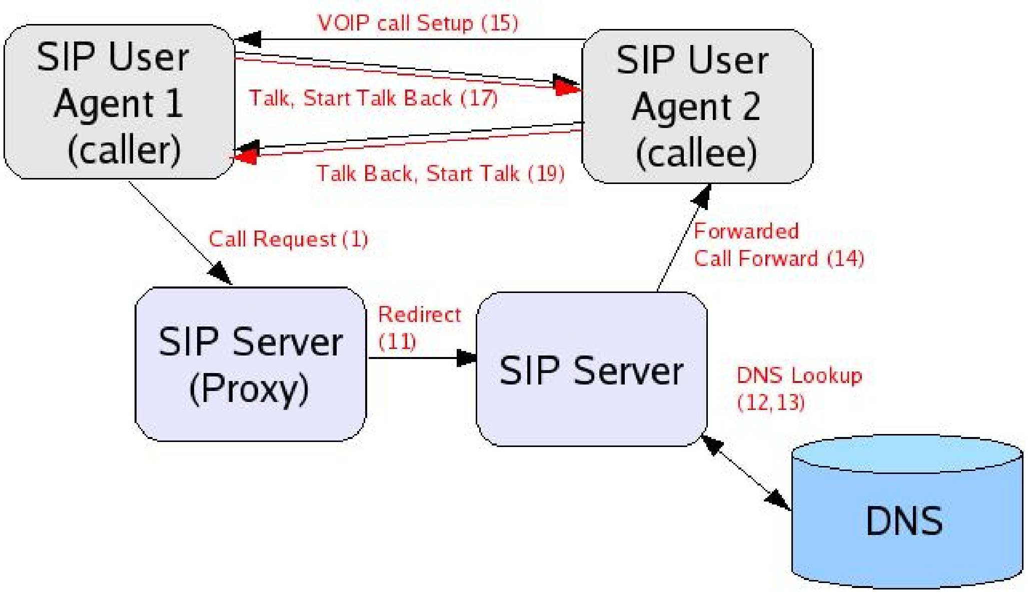
Scripts that implement this example are located in the rapr distribution in rapr/docs/samples/http-example.

18. How can I emulate SIP and VOIP?

This example will illustrate how to emulate a VOIP call using a simplification of the Session Initiation Protocol (SIP). The modeled system components and behavior include:

- 2 SIP user agents
- a SIP network server acting as a proxy
- a SIP network server
- a DNS server
- to initiate a session, the caller (or User Agent) sends a request to the SIP network server with the SIP URL of the called party.
- the SIP server will act as a proxy and redirect the call to another SIP network server that will ask DNS for the URL translation and forward the request to the callee (another SIP user agent).
- a VOIP conversation begins between the caller and callee

The logic id's we will use to trigger the messaging to model the behavior are in parenthesis in the following diagram:


SIP User Agent 1 (caller)
VOIP call Setup (15)
Talk, Start Talk Back (17)
Talk Back, Start Talk (19)
SIP User Agent 2 (callee)
Call Request (1)
Forwarded Call Forward (14)
SIP Server (Proxy)
Redirect (11)
SIP Server
DNS Lookup (12,13)
DNS

Create dictionary entries that model our system elements and logic ids, as well as for the SignallingPort and VoipPort:

```
<RaprDictionary>
   <namespace>
   <label>DEFAULT</label>
   <item>
       <field>SIP-UserAgent-1</field>
       <value>192.168.1.101</value>
   </item>
   <item>
       <field>SIP-Server-Proxy</field>
       <value>192.168.1.103</value>
   </item>
   <item>
       <field>SIP-Server</field>
       <value>192.168.1.104</value>
   </item>
   <item>
       <field>DNS-Server</field>
       <value>192.168.1.105</value>
   </item>
   <item>
       <field>SIP-UserAgent-2</field>
       <value>192.168.1.102</value>
   </item>
   <item>
       <field>CallRequest</field>
       <value>1</value>
   </item>
   <item>
       <field>CallRedirect</field>
       <value>11</value>
   </item>
   <item>
       <field>DNS-Lookup</field>
       <value>12</value>
   </item>
   <item>
       <field>DNS-Lookup-Success</field>
       <value>13</value>
   </item>
   <item>
       <field>CallForward</field>
       <value>14</value>
   </item>
   <item>
       <field>VOIPCall-Setup</field>
       <value>15</value>
   </item>
   <item>
       <field>Talk</field>
       <value>16</value>
   </item>
   <item>
       <field>StartTalkBack</field>
       <value>17</value>
   </item>
   <item>
       <field>TalkBack</field>
       <value>18</value>
   </item>
   <item>
       <field>StartTalk</field>
       <value>19</value>
   </item>
   <item>
       <field>SignallingPort</field>
       <value>6000</value>
   </item>
   <item>
       <field>VOIPPort</field>
       <value>8000</value>
   </item>
   </namespace>
</RaprDictionary>
```

Now let's model the behavior we defined in the above VOIP diagram in the logic table, and implement the messaging to be associated with the logic ids we have chosen.

[Notice that mgen patterns are not defined for some of the behavior events. When no pattern is defined, the default pattern of "PERIODIC [1 1024]" will be used. See the RAPR User's Guide for more information on system defaults. Notice also that we are not specifying a src port. This will cause the operating system to provide one. ljt remove any spurious "\"s thruout the doc and make sure we've introduced default]

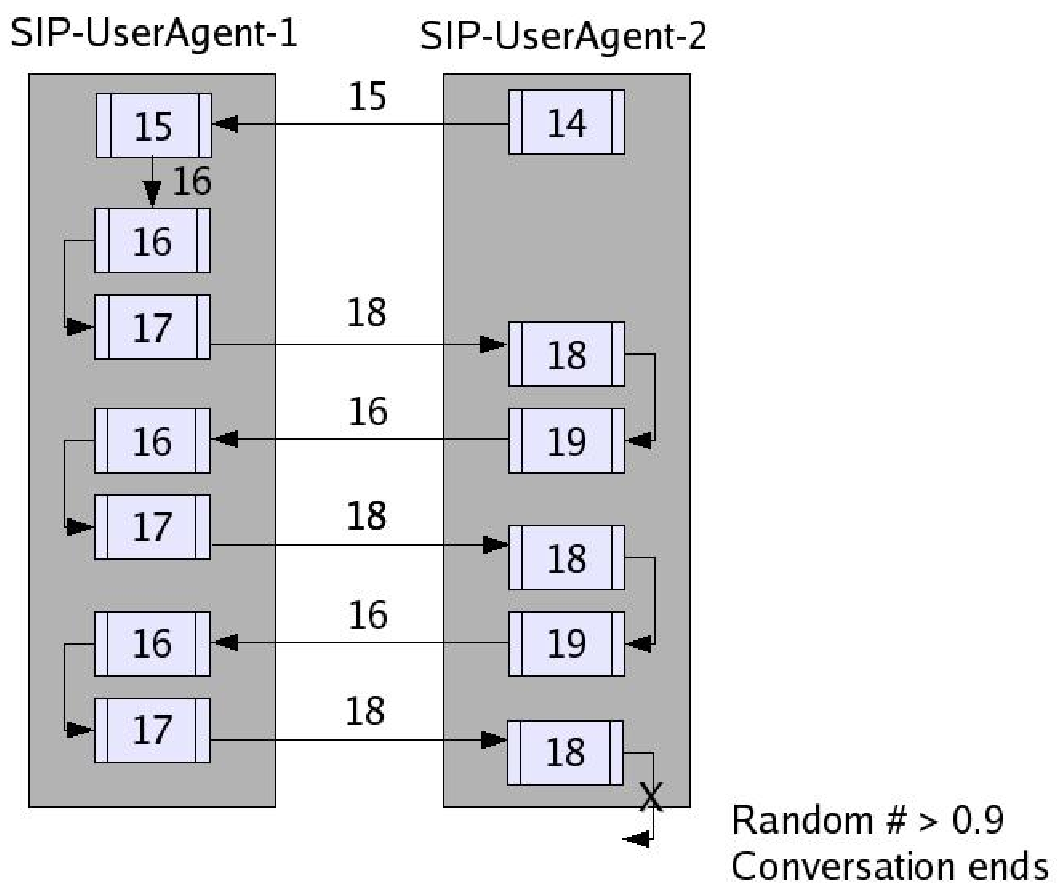
Modelling the call setup messaging is fairly straight forward for this simplified SIP example. We simply define DECLARATIVE messages to send logic id's to trigger the appropriate behavior at the appropriate node. E.g. when SIP-UserAgent-1 sends a CallRequest (a single message with logic id "1" in the payload) to the SIP-Server-Proxy, the SIP-Server-Proxy will create a DECLARATIVE message and send a CallRedirect message to the SIP-Server. See logic id's 1, 11-16 for call-setup messaging.

If all messages are successfully received, eventually SIP-UserAgent-1 will receive a Talk message that will cause it to start a DECLARATIVE behavior event within 0-2 seconds that emulates a VOIP traffic pattern that will last between 1 and 4 seconds. Other pattern attributes will be chosen randomly as well, for example, the event may send messages between 512 and 1024 bytes long.

Notice the new "SUCCESS" attribute associated with the "talk" and "talkback" behavior event(s). This attribute tells rapr to perform the behavior associated with the logicID in *its **own** logic table* when the behavior completes sucessfully. (A declarative behavior such as we are using to emulate a voice "talkspurt" will almost always complete successfully. See the Rapr User's Guide for more information on different logic id types). The success logic id will cause rapr to send a message (StartTalkBack) to SIP-UserAgent-2 directing it to "reply", e.g. start it's own VOIP emulation behavior event. When this behavior completes, it will invoke a "SUCCESS" logic id (StartTalk) to direct SIP-UserAgent-1 to "reply" , and so on.

Each node then, will "talk" for a random amount of time and then tell the other node that it is it's turn to "talk". Notice the <percent> attribute of logic id 19, StartTalk. The conversation will end when a random number draw is greater than 0.9 when this logic id is being processed. At this point the declarative message to trigger the next "talk" event will not be sent and the "conversation" will end.

The logic id's passed between the two user agents that direct the voip call are as follows:



The following logic table implements the behavior discussed above:

```
<RaprLogicTable>
  <state>
     <!-- Default state -->
     <value>0</value>
     <logicid>
         <!-- Redirect the request to the SIP-Server -->
         <id>1</id> <!-- CallRequest -->
         <entry>DECLARATIVE UDP
                DST %SIP-Server%/%SignallingPort%
                LOGICID %CallRedirect%</entry>
     </logicid>
     <logicid>
         <!-- Send a "dns-query" to the dns-server -->
         <id>11</id>  <!-- CallRedirect -->
         <percent>1</percent>
         <entry>DECLARATIVE UDP
                DST %DNS-Server%/%SignallingPort%
                LOGICID %DNS-Lookup%</entry>
     </logicid>
     <logicid>
         <!-- Return successful DNS response -->
         <id>12</id> <!-- DNS-Lookup -->
         <entry>DECLARATIVE UDP
                DST %PACKET:SRCIP%/%Signalling
                Port% LOGICID %DNS-Lookup-Success%</entry>
     </logicid>
     <logicid>
         <!-- Forward request to SIP-UserAgent-2 -->
         <id>13</id> <!-- DNS-Lookup-Success -->
         <entry>DECLARATIVE UDP
                DST %SIP-UserAgent-2%/%SignallingPort%
                LOGICID %CallForward%</entry>
```

```
      </logicid>
      <logicid>
          <!-- Setup VOIP call with SIP-UserAgent-1 -->
          <id>14</id> <!-- CallForward -->
          <entry>DECLARATIVE UDP
                 DST %SIP-UserAgent-1%/
                 %VOIPPort% LOGICID %VoipCall-Setup%</entry>
      </logicid>
       <logicid>
          <!-- %VoipCall-Setup% -->
          <id>15</id> <!-- Start talking -->
          <entry>DECLARATIVE UDP SRC 5001
                 DST %SIP-UserAgent-2%/%SignallingPort%
                 LOGICID %Talk%</entry>
      </logicid>
      <logicid>
          <!-- %Talk% -->
          <id>16</id> <!-- Start talking, when done  -->
                      <!-- invoke logicID 18         -->
          <entry>%SYSTEM:RANDOMF(0.0,2.0)% DURATION
                 %SYSTEM:RANDOMF(1,4)% DECLARATIVE UDP
                 SRC 5000 DST %SIP-UserAgent-1%/%VOIPPort% BURST
                 [RANDOM %SYSTEM:RANDOMI(5,10)%
                 POISSON [%SYSTEM:RANDOMI(5,10)%
                 %SYSTEM:RANDOMI(512,1024)%] EXP 5.0]
                 SUCCESS %StartTalkBack%</entry>
      </logicid>
      <logicid>
          <!-- %StartTalkBack% -->
          <id>17</id> <!-- Tell SIP-UserAgent-2 to start talking -->
          <entry>DECLARATIVE UDP SRC 5001 DST %SIP-UserAgent-1%/
                  %SignallingPort% LOGICID %TalkBack%</entry>
      </logicid>
      <logicid>
          <!-- %TalkBack% -->
           <id>18</id> <!-- Start talking, when done -->
                      <!-- invoke logicID 19        -->
          <entry>%SYSTEM:RANDOMF(0.0,2.0)% DURATION
                 %SYSTEM:RANDOMF(1,10)% DECLARATIVE UDP SRC
                 5000 DST %SIP-UserAgent-2%/%VOIPPort% BURST
                 [RANDOM %SYSTEM:RANDOMI(5,10)%
                 POISSON [%SYSTEM:RANDOMI(5,10)%
                 %SYSTEM:RANDOMI(512,1024)%] EXP 5.0]
                 SUCCESS %StartTalk%</entry>
      </logicid>
      <logicid>
          <!-- %StartTalk% -->
          <id>19</id> <!-- -->
          <percent>0.9</percent>
          <entry>DECLARATIVE UDP SRC 5001 DST %SIP-UserAgent-2%/
                 %SignallingPort% LOGICID %Talk%</entry>
      </logicid>
  </state>
</RaprLogicTable>
```

To test the application, create an input script for SIP-UserAgent-1 to initiate the VOIP call. Direct the mgen output to a log file:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-voip.xml
LOGICTABLE_FILE logictable-voip.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-voip.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-voip.log

# Listen for HTTP requests
LISTEN UDP %SignallingPort%
LISTEN UDP %VOIPPort%
```

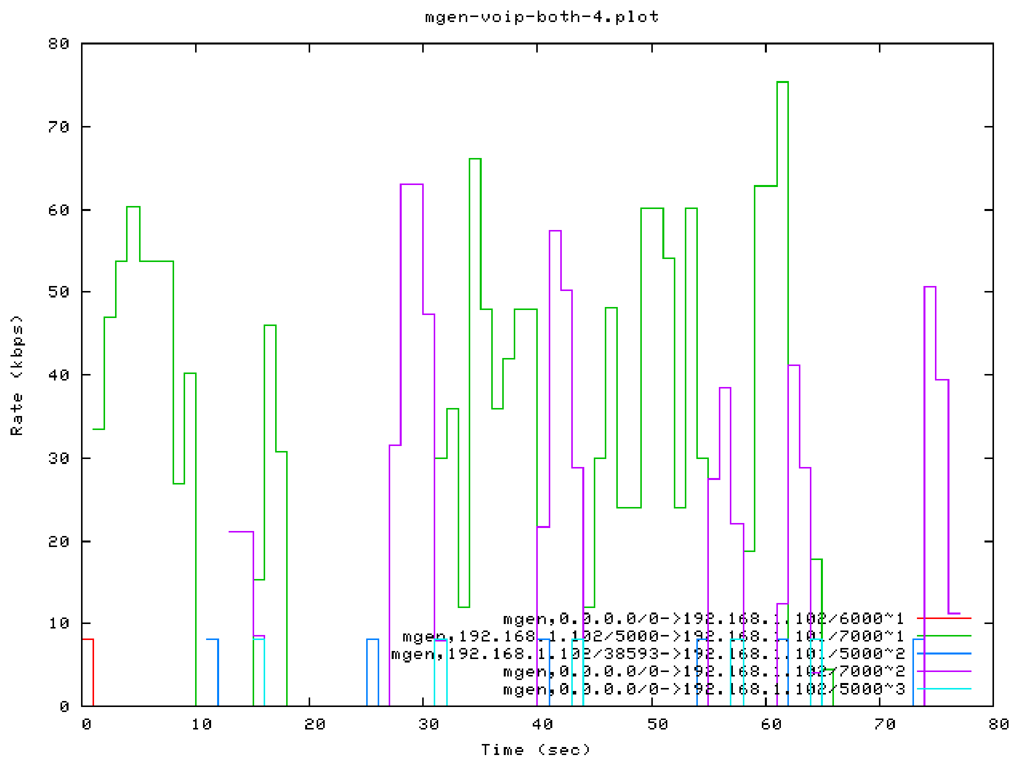
```
DECLARATIVE UDP DST %SIP-Server-Proxy%/%SignallingPort% \
LOGICID %VOIP-CallSetup% COUNT 1
```

The other node simply needs to listen for data traffic and load the voip dictionary and logic tables:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-voip.xml
LOGICTABLE_FILE logictable-voip.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-voip.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-voip.log

# Listen for HTTP requests
LISTEN UDP %SignallingPort%
LISTEN UDP %VOIPPort%
```

The diagram below was generated by a program called TRPR. TRPR reads mgen files and plots the data in a variety of ways. This diagram simply plots rate over time for sent and received VOIP (not setup) messages in the mgen log file. TRPR can be obtained at NRL's protean forge website.

Note that the "talk spurts" from SIP-UserAgent-1 last between 1 and 10 seconds, the "talk" spurts from SIP-User-Agent-2 last between 1 and 4 seconds. The "talk spurts" start 0 to 2 seconds after receipt of a triggering logic id.



19. How can the Periodic behavior event help simplify my scripts?

Periodic behavior events can be used to spawn other behavior event types (Declarative and Interrogative only) at regular intervals. These generated behavior events behave as independent events and have no relationship to one another.

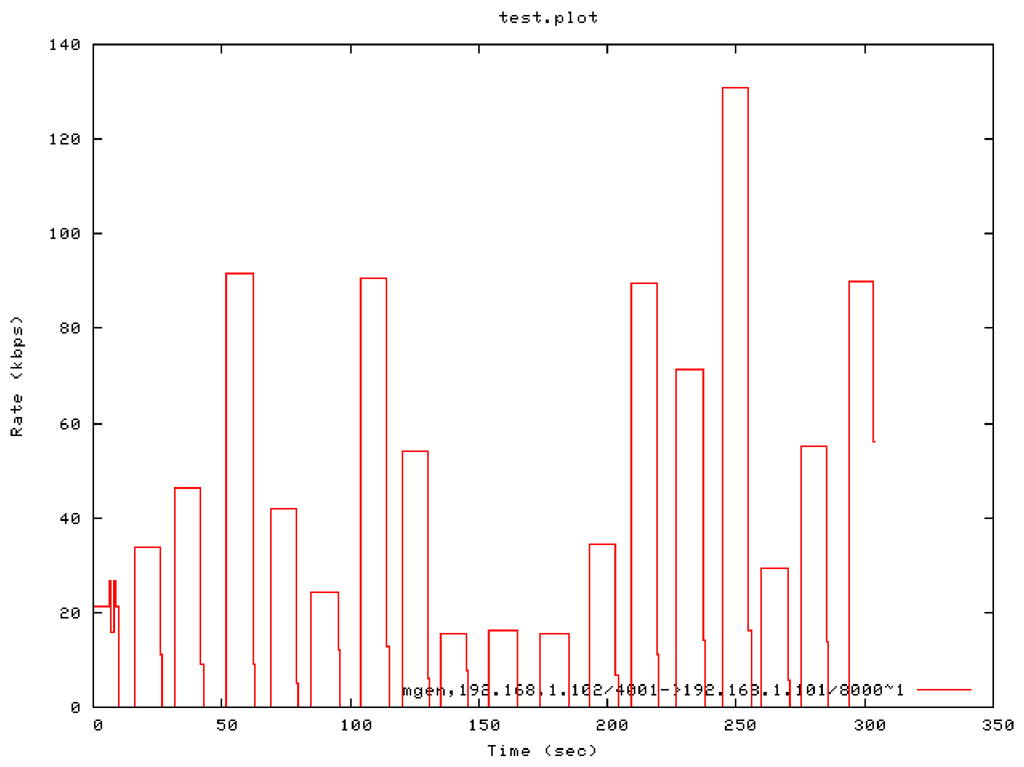
In the following example, the first periodic event will send a single 1024 byte udp message every 5 seconds for 30 seconds; the second periodic event will start an interrogative event every 60 seconds for 5 minutes:

```
DURATION 30.0 PERIODIC INTERVAL 5 DECLARATIVE UDP \
DST 192.168.1.102/5001 PERIODIC [1 1024] COUNT 1


DURATION 300 PERIODIC INTERVAL 60 INTERROGATIVE UDP \
DST 192.168.1.101/8000"
```

Each time a periodic event is restarted, the attributes will be retranslated. Thus a single script event line such as the following:

```
DURATION 300.0 PERIODIC INTERVAL %SYSTEM:RANDOMI(15,20)% \
DURATION 10.0 DECLARATIVE UDP SRC 4001 DST 192.168.1.101/8000 \
PERIODIC [%SYSTEM:RANDOMI(1,10)% %SYSTEM:RANDOMI(512,2048)%]
```

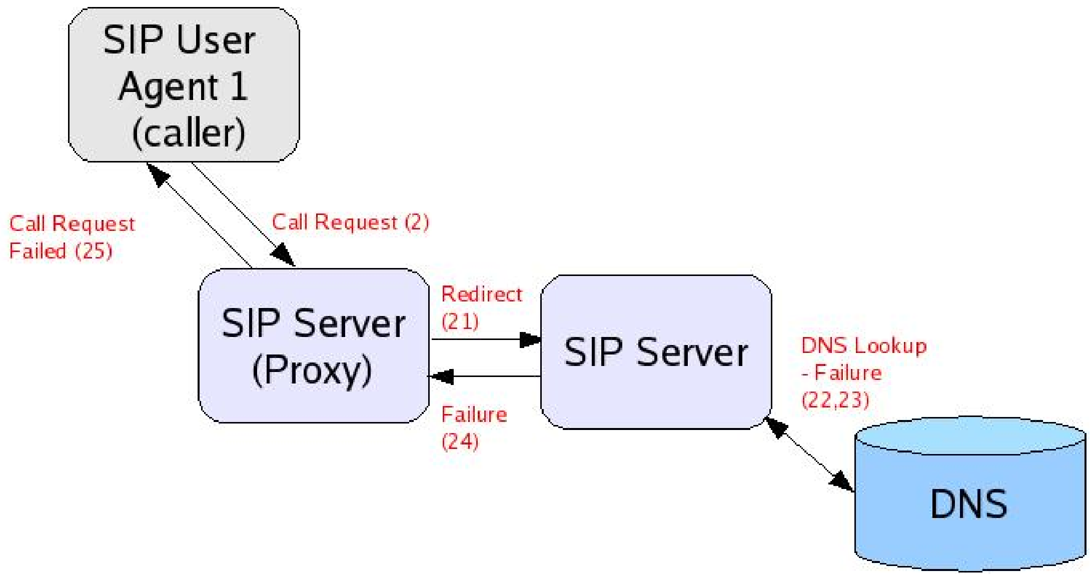
will result in bursts of packets of varying sizes and frequency being sent every 15 to 20 seconds to the destination. A distinct behavior event for each "traffic burst" need not be scripted:



To see how using the periodic object can be used to expand upon application behavior patterns add the following logic codes to the SIP logictable discussed in the previous section.

```
 <RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>
      <logicid>
          <!-- Redirect the request to the SIP-Server -->
          <id>2</id> <!-- CallRequest -->
          <entry>DECLARATIVE UDP
                 DST %SIP-Server%/%SignallingPort%
                 LOGICID %CallRedirect%</entry>
      </logicid>
      <logicid>
          <!-- Send a "dns-query" to the dns-server -->
          <id>21</id>  <!-- CallRedirect -->
          <percent>1</percent>
          <entry>DECLARATIVE UDP
                 DST %DNS-Server%/%SignallingPort%
                 LOGICID 22</entry>
      </logicid>
      <logicid>
          <!-- Return unsuccessful DNS response -->
          <id>22</id> <!-- DNS-Lookup -->
          <entry>DECLARATIVE UDP
                 DST %PACKET:SRCIP%/%Signalling
                 Port% LOGICID 23</entry>
      </logicid>
      <logicid>
          <!-- Failure -->
          <id>23</id> <!-- DNS-Lookup-Failure -->
          <entry>DECLARATIVE UDP DST %SIP-Server-Proxy%
                 %SignallingPort% LOGICID 24</entry>
                 <!-- No response -->
      </logicid>
      <logicid>
          <!-- Failure no response -->
          <id>24</id> <!-- DNS-Lookup-Failure -->
          <entry>DECLARATIVE UDP DST %SIP-UserAgent-1%/
                 %SignallingPort% LOGICID 25</entry>
                 <!-- No response -->
      </logicid>
  </state>
</RaprLogicTable>
```

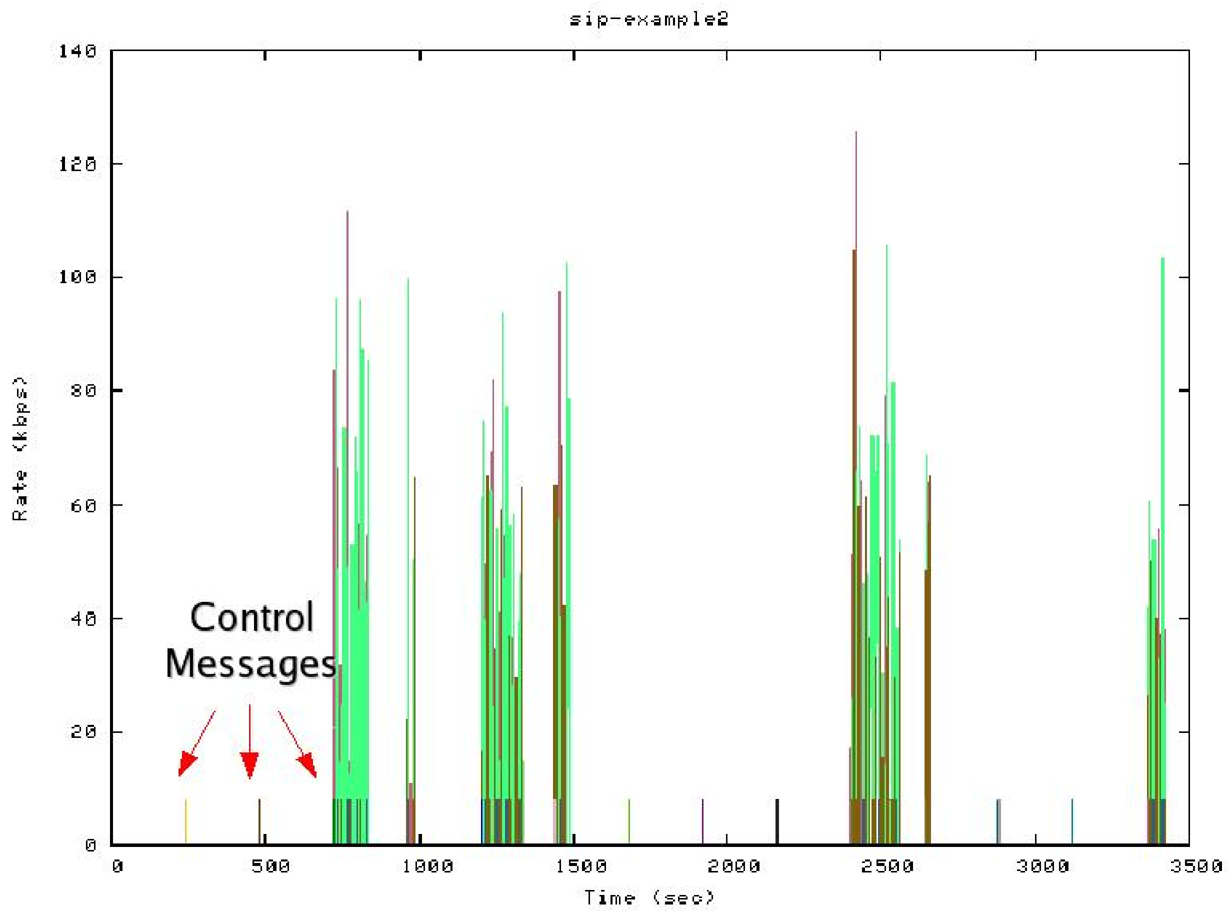
The behavior modeled by this application "pattern" is a SIP call setup that fails due to dns failure and can be visualized as follows:



Now use a PERIODIC object to initiate a "SIP Call setup" every 4 minutes. The periodic object will randomly choose between logic id 1 (which results in a VOIP call) or logic id 2 (which results in no call setup due to dns failure). Each time the periodic interval elapses, one behavior or another will be randomly chosen.

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-sip.xml
LOGICTABLE_FILE logictable-sip.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-sip.log
#OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-sip.log

# Listen for HTTP requests
LISTEN UDP %SignallingPort%
LISTEN UDP %VOIPPort%

0.0 DURATION 3600.0 PERIODIC INTERVAL 240.0 \
DECLARATIVE UDP DST %SIP-Server-Proxy%/%SignallingPort% \
LOGICID %SYSTEM:RANDOMI(1,2)%
```

A plot generated by TRPR shows the modified "application" behavior. Every 4 minutes, the periodic object sends "control messages" (messages with logic codes 1 or 2 in the payload) that result in either a "VOIP conversation" between the two nodes (logic code 1) or in a "SIP Call setup failure" (logic code 2).



20. What is the Stream behavior event?

Stream behavior events can be used to emulate VOIP traffic with much less scripting than can be achieved using other behavior event types. For example, we can implement the "VOIP" conversation discussed in the previous SIP example with just one logic table entry and one script entry. (Note that we used declarative behavior events in the SIP example because STREAM event processing cannot be initiated by a logic id at this time. STREAM processing **must** be initiated from an input script.)

STREAM events can also be used to more easily emulate a "group chat" conversation amongst multiple nodes.

To set up VOIP emulation using STREAM processing, an initial STREAM behavior event should be defined in the input script of the triggering node and be associated with a logic id. This logic id will direct other nodes who are to participate in the VOIP emulation to start associated STREAM events. A STREAM id is also embedded in the payload of STREAM messages that is used to coordinate the "conversation" amongst the participating nodes.

Each STREAM event should be assigned a "response probability". This attribute defines the probability that the STREAM event should be the next event (or node) in the conversation to "respond". For example, to coordinate a conversation between 2 nodes, one node should be associated with a response probability of between 0 and 50, the second with a probability between 51 and 100. Upon receipt of the first STREAM message, both nodes will draw a random number between 1 and 100. Stream messages also embed a SEED in the payload which is used to seed a random number generator and therefore all nodes in the conversation will draw the same number. Thus, whichever node draws a number within their response probability range will be the next node to reply.

The following table summarizes in more detail the attributes of the STREAM behavior event:

RESPPROB	Stream messages are sent with a seed that is used by the participating nodes to determine whether they should be the next node to respond to the "conversation". This incoming seed is used by all participating nodes to seed a random number generator. If the next random number draw falls within the response probability range (RESPPROB <lowRange> <highRange>) defined for a given node, the node will be the next to "speak" in the "conversation". Note that a triggering node may well be elected "next to respond". The response ranges should be equally distributed amongst all nodes participating in the "conversation" such that only one node can be elected to respond. For example, to set up a group conversation among four nodes each with an equal probability of reply, setup the RESPPROB attribute as follows ... STREAM RESPPROB 0 25 (node A) ... STREAM RESPPROB 26 50 (node B) ... STREAM RESPPROB 51 75 (node C) ... STREAM RESPPROB 76 100 (node D)
BURSTDURATION	The BURSTDURATION attribute specifies the length in seconds of the node's reply to a "conversation". A reply can be thought of as a "burst". This setting may be superceded by the BURSTRANGE attribute.
BURSTCOUNT	The BURSTCOUNT attribute defines the length of the "conversation". A triggering stream with a burst count of 5 will result in 5 "bursts" of conversation from any of the participating nodes.
BURSTDELAY	The BURSTDELAY attribute defines the range to be used to determine the number of seconds to wait before a node will respond to a "conversation" (if so elected based on its response probability)
BURSTRANGE	The BURSTRANGE attribute defines the range to be used to determine the length of the next burst response in number of seconds (if so elected based on its response probability). Note that BURSTDURATION can also be used to specify the length of burst responses. For example: ... STREAM BURSTRANGE 5 10 ... will cause the node to send a burst response between 5 and 10 seconds.
TIMEOUTINTERVAL	The TIMEOUTINTERVAL attribute defines the length of time (in seconds) that the stream event will wait for response traffic. If no response is received from ANY node participating in the conversation (including intself) within this timeout interval, the stream object will "timeout" and stop the stream event, and not respond to any further traffic associated with the stopped "conversation". For example: ... STREAM BURSTRANGE 5 10 ... will cause the node to send a burst response between 5 and 10 seconds.
BURSTPRIORITY	A BURSTPRIORITY can be assigned to STREAM events that will cause the "conversation" to take precedence over any other ongoing conversation. For example, if a conversation of default priority (0) is ongoing, and a second "conversation" is initiated at a higher priority, the higher priority conversation will be "served" before the lower priority conversation.

In the following STREAM example, we'll use a multicast address 224.225.1.2 for the "voip" traffic. We'll set up a STREAM converstaion between two nodes. Each node should "JOIN" the multicast group in the input scripts.

As previously mentioned, STREAM events must be initiated from an input script. The STREAM definition in the following input script specifies that a random number draw must be between 51 and 100 before it will send a "burst" of messages. When it does send traffic, it will wait 0-2 seconds before responding (as defined by the BURSTDELAY attribute) with a burst between 4 and 10 seconds (as defined by the BURSTRANGE). The BURSTCOUNT attribute specifies that the conversation will consist of 20 talks spurts, or "traffic bursts". Again, we are using the same behavior pattern as specified in the SIP example.

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-stream.xml
LOGICTABLE_FILE logictable-stream.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-stream.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-stream.log

# Listen for HTTP requests
LISTEN UDP %VOIPPort%
JOIN 224.225.1.2

STREAM RESPPROB 51 100 BURSTDELAY 0 2 BURSTRANGE 4 10 \
BURSTCOUNT 20 UDP SRC 5000 DST 224.225.1.2/%VOIPPort% \
BURST [RANDOM %SYSTEM:RANDOMI(5,10)% POISSON \
[%SYSTEM:RANDOMI(5,10)% %SYSTEM:RANDOMI(512,1024)%] EXP 5.0] \
LOGICID 99
```

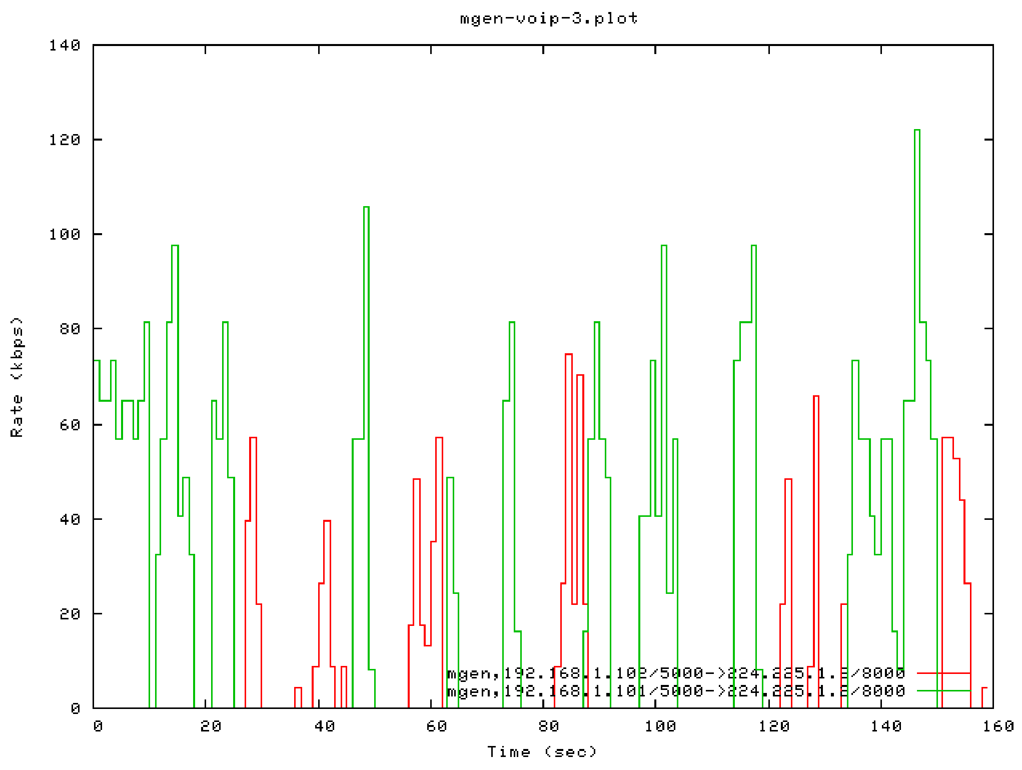
Logic id "99" in the listening node should also define a STREAM event. In this example, the STREAM event will respond if the random number draw is between 0 and 50. The stream event will wait between 0 to 2 seconds before responding with a burst length between 4 and 10 seconds. The same pattern we defined in the previous SIP example is used.

```
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>
      <logicid>
         <id>99</id>
         <entry>STREAM RESPPROB 0 50 BURSTDELAY 0 2 BURSTRANGE 4 10 UDP SRC 5000 DST %PACKET:DSTIP%/%VOIP
                 [RANDOM %SYSTEM:RANDOMI(5,10)%
                 POISSON [%SYSTEM:RANDOMI(5,10)%
                 %SYSTEM:RANDOMI(512,1024)%] EXP 5.0] </entry>
      </logicid>
  </state>
</RaprLogicTable>
```

The listening node simply needs to list to the VOIP port and JOIN the multicast group:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY /home/nrl/rapr/unix/dictionary-stream.xml
LOGICTABLE_FILE /home/nrl/rapr/unix/logictable-stream.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-stream.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-stream.log

# Listen for HTTP requests
LISTEN UDP %VOIPPort%
JOIN 224.225.1.2
```

As shown in the following TRPR output, the STREAM objects will emulate a VOIP conversation with much less "scripting". Notice that the "triggering" node, (192.168.1.102) was more likely to respond in this example:



To add other nodes to the conversation we simply need to define the logic id in their logic tables (in this case 99), listen to the multicast address, and make sure the response probabilities for the behavior event are set appropriately. For example, to define a "group chat" conversation between 4 nodes that have an equal probability of "chatting", change the STREAM definition for the initiating node to have a response probability of 0 to 25:

```
# Node A initiating node's input script
#
#
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY dictionary-sip.xml
LOGICTABLE_FILE logictable-sip.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-voip-3.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-voip-3.log

# Listen for HTTP requests
LISTEN UDP %SignallingPort%
LISTEN UDP %VOIPPort%
JOIN 224.225.1.2

STREAM RESPPROB 51 100 BURSTDELAY 0 2 BURSTRANGE 4 10 \
BURSTCOUNT 20 UDP SRC 5000 DST 224.225.1.2/%VOIPPort% \
BURST [RANDOM %SYSTEM:RANDOMI(5,10)% POISSON \
[%SYSTEM:RANDOMI(5,10)% %SYSTEM:RANDOMI(512,1024)%] EXP 5.0]\
LOGICID 99
```

Next create separate logic tables for the three other nodes, ensuring that the response probabilities for all participating nodes do not overlap and add up to 100:

```
<!-- Node B -->
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>

      <logicid>
         <id>99</id>
         <entry>STREAM RESPPROB 26 50
```

```
                BURSTDELAY 0 2 BURSTRANGE 4 10
                UDP SRC 5000 DST %PACKET:DSTIP%/%VOIPPort%
                BURST [RANDOM %SYSTEM:RANDOMI(5,10)% POISSON
                %SYSTEM:RANDOMI(5,10)% %SYSTEM:RANDOMI(512,1024)%]
                EXP 5.0] </entry>
     </logicid>
  </state>
</RaprLogicTable>
```

```
<!-- Node C -->
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>

      <logicid>
         <id>99</id>
         <entry>STREAM RESPPROB 51 75
                BURSTDELAY 0 2 BURSTRANGE 4 10
                UDP SRC 5000 DST %PACKET:DSTIP%/%VOIPPort%
                BURST [RANDOM %SYSTEM:RANDOMI(5,10)% POISSON
                %SYSTEM:RANDOMI(5,10)% %SYSTEM:RANDOMI(512,1024)%]
                EXP 5.0] </entry>
     </logicid>
  </state>
</RaprLogicTable>
```

```
<!-- Node B -->
<RaprLogicTable>
   <state>
      <!-- Default state -->
      <value>0</value>

      <logicid>
         <id>99</id>
         <entry>STREAM RESPPROB 76 100
                BURSTDELAY 0 2 BURSTRANGE 4 10
                UDP SRC 5000 DST %PACKET:DSTIP%/%VOIPPort%
                BURST [RANDOM %SYSTEM:RANDOMI(5,10)% POISSON
                %SYSTEM:RANDOMI(5,10)% %SYSTEM:RANDOMI(512,1024)%]
                EXP 5.0] </entry>
     </logicid>
  </state>
</RaprLogicTable>
```

Nodes B, C, and D simply need to join the multicast group and load the appropriate logic table:

```
# The TXLOG enables message transmission logging
TXLOG
LOAD_DICTIONARY /home/nrl/rapr/unix/dictionary-stream.xml
LOGICTABLE_FILE /home/nrl/rapr/unix/logictable-stream.xml
OVERWRITE_MGENLOG /home/nrl/rapr/log/mgen-stream.log
OVERWRITE_RAPRLOG /home/nrl/rapr/log/rapr-stream.log

# Listen for HTTP requests
LISTEN UDP %VOIPPort%
JOIN 224.225.1.2
```

21. What is the remote control interface?

Rapr provides a "remote control interface" that can be used to send commands to a running rapr application. To enable the interface, start rapr by providing an instance name:

```
RAPR instance RAPR1
```

Subsequent invocations of RAPR specifying the same instance name will pass provided commands to the first instance and then exit: (Note that the quotes around the event definition are required.)

```
RAPR instance RAPR1 event "0.0 DECLARATIVE UDP DST 127.0.0.1/5000 \
PERIODIC [1 1024]"
```

This is the log from the first RAPR instance that creates the DECLARATIVE event:

```
[debussy:]$ /home/nrl/rapr/unix/rapr  instance rapr-ljt
rapr:version 0.6.0
mgen:version 5.01b
rapr: starting now ...
21:46:15.499981 START
17:46:15.499965 app>RAPR type>Application action>Application\
StartUp
17:47:09.003846 app>RAPR type>Declarative action>start \
ubi>1694498816 eventSource>rti_event mgenCmd>"ON 1 UDP SRC \
0 DST 192.168.1.100/6000 PERIODIC [1 1024 ] DATA [03040597B627] \
 COUNT 1"
17:47:09.003923 app>RTI_EVENT TYPE>Command event arg: 0.0 \
DECLARATIVE UDP DST 192.168.1.100/6000 PERIODIC [1 1024]
17:47:09.994299 app>RAPR type>Declarative action>timeout \
ubi>1694498816 0.001 OFF 1
```

22. How to synchronize scenario scripts across a network?

To assist in starting application or "scenario" scripts across a network, RAPR provides a START command that designates an absolute start time. This <hour:min:sec> field corresponds to the relative script time of 0.0 seconds. All transmission and reception events will be scheduled relative to this absolute start time. The optional GMT suffix (no white space after the time) indicates that the clock time given is Greenwich Mean Time (GMT) rather than the operating systems local time zone. If no START command is given, RAPR schedules transmission and reception events relative to program startup. For example:

```
#Start RAPR exactly at 1:30PM local time
START 13:30:00

#Start RAPR at 30 seconds past 8:30
START 8:30:30GMT
```

When specifying an absolute start time, it is important that the system clocks of network nodes be syncrhonized via a time protocol such as NTP or GPS.

23. What are RAPR defaults?

The RAPR application provides certain application wide default values. In addition, a "default" dictionary is provided in the RAPR distribution that can be used to override these application wide default values. If no default dictionary is loaded into a running rapr application via a "LOAD_DICTIONARY" command RAPR will default to using the values as specified in the table below. The "dictionary Field" columns lists the dictionary field names that must be used in the DEFAULT system name space. The "system value" lists the internal application wide default that can be overridden in the default dictionary.

The The default dictionary is included in the rapr distribution in the unix directory. It is called "raprDictionary.xml".

Table 1. Default Dictionary

Dictionary Field	System Value	Comments
RETRYINTERVAL	10	The default retry interval used by an interrogative object if no RETRYINTERVAL attribute is specified.
NUMRETRIES	3	The default number of retries used by an interrogative object if no NUMRETRIES attribute is specified.
PATTERN	PERIODIC [1 1024]	The default mgen flow pattern that is used for all behavior events when no PATTERN is specified.
STREAMDURATION	.99	The default stream duration that is used by the interrogative object for each message query or as the default duration for declarative objects. See the object definitions for more details on how this interval is used/calculated.
PROTOCOL	UDP*	The default protocol to be used by all behavior events. Note that this default can *only* be set in the default dictionary. No internal default PROTOCOL is set system wide. If no entry is set specified in the default dictionary, the PROTOCOL *must* be set on the command line for each behavior event.