Getting Started with QUENAS

Alejandro Marcu

October 13, 2006

Contents

1 Introduction

QUENAS (QUeued Network Automatic Simulation) is a simulator created to analyze the Hypercube Protocol, but designed with the idea of being capable of running any other protocol.

The simulator takes an input file with commands and produces an output file with the result of the simulation.

The input file contains commands in a simple interpreted language, enabling the user to build a network, set its parameters, run applications on the nodes, query the different objects and so on.

This guide has the objective of helping the new user to get their first simulations working, and provide a brief tutorial about the simulator commands.

2 Getting Started

After downloading and unpacking QUENAS, the first step is to compile it to produce the executable file. Then, some internal tests can be run to check that all is working as expected.

Once done with that, the simulator is ready to run. A first detailed example is presented in this section to show how a simulation is run.

2.1 Compiling and running QUENAS

After getting and unpacking the distribution file, go to `quenas` directory and run at the shell:

```
$ make quenas
```

This will completely build an executable for your platform. Then, you are ready to run simulations. You can first run the simulator in test mode to check that everything is in order:

```
$ quenas -test
```

Running a simulation is very easy; you just need to provide an input file (usually with `sim` extension), and the name for an XML output file:

```
$ quenas example.sim ouput.xml
```

The output file can be seen with any text viewer or with a browser. The simulations can optionally use XSLT templates to provide a nice view of the output.

2.2 Hello World! Simulation

As usual, a "Hello World!" example is built in this section for running your first simulation.

Open your favorite text editor to create the file `helloworld.sim` and type:

```
setAddressLength(4)

newNode(hello)
newNode(world)

newConnection(hello, world, 2048 Kbps, 1 ms)

[1 s] node(hello).joinNetwork()
[10 s] node (world).joinNetwork()

[15 s] allNodes.query()
```

The first line sets the address length in the network, in that case it will be using 4 bits. The next two lines create nodes named "hello" and "world", and a connection between them is created right after that.

Notice that those commands will run as soon as the simulation starts, and even if nodes are created and physically connected, they won't connect to the network until they are told to do so.

The following line starts with `[1 s]`, which instructs the simulator that the command in that line (and all the lines below until a new time indication is given), will run at the specified time, and make the "hello" node join the network, which is initially empty. At 10 seconds, the "world" node joins the network.

Finally, the last line makes the simulator query all the nodes, printing the results in the output file.

To run this simulation, type:

```
$ quenas helloworld.sim helloworld.xml
```

It will produce an output like:

```
Loading file helloworld.sim...
Running simulation. Output being sent to helloworld.xml...
Simulated time: 15.0001 s
```

If an error message is displayed, check that the simulation file was correctly introduced.

To see the result from the simulation, open `helloworld.xml` in a text editor. The output should be:

```
<simulation>
<Notification id="simulator.exec.query">
    <time>15 s</time>
    <command>allNodes.query</command>
    <Network>
        <Node id="hello">
            <universalAddress>hello</universalAddress>
            <primaryAddress>0000/1</primaryAddress>
            <MACAddress>D2:18:E9:05:00:00</MACAddress>
        </Node>
        <Node id="world">
            <universalAddress>world</universalAddress>
            <primaryAddress>1000/1</primaryAddress>
            <MACAddress>92:1B:C1:06:00:00</MACAddress>
        </Node>
    </Network>
</Notification>
</simulation>
```

This output shows the addresses of the nodes at 15 s.

The simulator ends just after running the last command in the file. Some commands, like `joinNetwork`, initiate a connection process that take some simulated time, so if they're run as the last command, they won't have time to complete, and the user may wonder why the last command didn't work. This is why its recommended to end the simulation with a query that runs some time after the last command that takes some simulated time, in order to give it time to finish.

3 Writing Simulations

To run a simulation, a file must be created with commands that will be interpreted by the simulator, describing the network topology and the actions that it must take.

This section explains the commands that are needed to write such file. For a complete reference of the commands, see Appendix A.

3.1 Creating a Network

The simulation file will usually start by defining a network structure by creating nodes and connections between them.

The address length of the network should be specified, or a default value of 8 will be used. This is done with the command `setAddressLength`, for example:

```
setAddressLength(12)
```

After that, the nodes can be created, giving them a name that is used as its universal address:

```
newNode(a)
newNode(b)
newNode(c)
newNode(foo)
```

To create physical connections between the nodes, use `newConnection` command, whose first two parameters are the mandatory node names, the third and fourth parameter are optional for specifying the bandwidth and delay:

```
newConnection(a,b) # ideal connection: infinite bandwidth, 0 delay
newConnection(b,c, 1 Mbps) # 0 delay
newConnection(a,foo, 256 Kbps, 1 ms)
```

Notice that "`#`" is used for comments, and everything after that sign until the end of the line is not processed.

Both nodes and connection can be retrieved after being created to run commands on them:

```
node(a).query() # will give information about node a

connection(b,c).setDelay(1 ms)

connection(a,b).setDelay(100 us).setBandwidth(1 Gbps)
```

In the last example, methods are chained, which is possible because `setDelay` method returns the connection after setting the delay.

Also notice that blank lines can be entered in the simulation file without any effect, but for the only purpose of making the simulation easier to read.

Connections and nodes can be created and modified not only at the begining of the simulation, but also in any simulation time.

The simulator has some functions that return a set of objects; in that case the following functions will be executed over all the objects:

```
allNodes.query()
```

The function `allNodes` returns all the nodes in the network, and `query` is executed on each of them, generating that way a report of all the nodes of the network.

All the node connections can also be retrieved:

```
node(a).allConnections.query()

node(b).allConnections.setBandwidth(1 Mbps)
```

This kind of functions can be chained:

```
allNodes.allConnections.setDelay(1 us)
```

The above example can be very usefull when building a network where all the connections have the same parameter values.

3.2 Joining and Leaving the network

Nodes are logically disconnected when created, so you must make them join the network to have something to simulate.

```
[10 s] node(a).joinNetwork()
[12 s] node(b).joinNetwork()

[15 s] allNodes.query()

[20 s] node(c).joinNetwork()

[25 s] allNodes.query()
```

The queries are added in the example in order to see how are nodes acquiring addresses. It can be verified that the addresses obtained are the expected ones by using assertions in primary or secondary addresses:

```
node(a).assertPrimaryAddress('0100/3')
node(b).assertPrimaryAddress('0110/4')
node(b).assertSecondaryAddresses('0111/4')
```

If the address obtained is different than the expected, the simulator will stop with an error message. This is usefull to build test cases, since the results don't need to be verified by hand; a succesful ending of the simulator will mean that the result was the expected.

Nodes can be forced to leave the network with `leaveNetwork` command:

```
[30 s] node(a).leaveNetwork()

[32 s] allNodes.query()

[35 s] node(foo).joinNetwork()

[38 s] node(a).joinNetwork

[40 s] allNodes.query()
```

Nodes can join and leave the network as many times as desired.

3.3 Tracing Routes

Once nodes are connected, routes can be traced. For that purpose, a Trace Route application is built into the simulator, which sends a packet with a special mark to make nodes in the path add an optional header to the packet with its address. In order to do a trace route, a command like the following is needed:

```
node(a).traceRoute.trace('0101')
```

A route will be traced from node `a` to the node having hypercube address `0101`. When the packet arrives to destination or finds out that the destination doesn't exist, a notification in the simulator is raised. However, by default the notifications are not written to the output file, so to be able to see the route traced you need to enable this notification adding at the very begining of the simulation file:

```
simulator.notifFilter.accept('node.routing.trace')
```

Trace Route application will produce in the output file a fragment like:

```
<Notification id="node.routing.trace">
    <time>100.02 s</time>
    <TraceRoute id="Trace Route at 100 s from 0000 to 0101">
        <distance>2</distance>
        <hop>
            <node>0100</node>
            <nodeName>g</nodeName>
        </hop>
        <hop>
            <node>0101</node>
            <nodeName>z</nodeName>
        </hop>
    </TraceRoute>
</Notification>
```

This fragment shows the route that the packet has followed and the distance it travelled.

It is also possible to use the simulator to assert that the route followed is the expected one. In that case, the simulator will give an error message if the route is not the expected. For example, for the above situation:

```
node(a).traceRoute.assert('0101', 'g;z')
```

will check that the packet arrived to destination going to `g` and then to `z`. You must not leave any blank spaces before or after node names. In order to assert that there is no route to host, use an empty string.

The routing tables can be queried in any moment, using the command:

```
node(a).routing.table.query()
```

The `routing` function brings the routing algorithm object for the node, whose table is asked with the `table` command, and the query is applied over this last one.

As expected, the routing tables of all the nodes can be queried at once using:

```
allNodes.routing.table.query()
```

3.4 Rendez Vous Server

Each node has a Rendez Vous Server application, who is in charge for maintaining a table with the primary addresses of some universal addresses. This application can be queried to see its table:

```
node(a).rendezVousServer.query
```

This produces a fragment like:

```
<Notification id="simulator.exec.query">
    <time>125 s</time>
    <command>node(a).rendezVousServer.query</command>
    <RendezVousServer>
        <Entry>
            <node>b</node>
            <address>1000</address>
        </Entry>
        <Entry>
            <node>d</node>
            <address>1010</address>
```

```
        </Entry>
        <Entry>
            <node>f</node>
            <address>1111</address>
        </Entry>
    </RendezVousServer>
</Notification>
```

3.5 Using the Test Application

Each node has a "Test Application" that is built over the hypercube protocol to send messages to Test Applications in other nodes, measuring departure and arrival time. To send a packet with this application, the command is:

```
node(a).testApplication.send(b)
```

Notice that, contrary to the Trace Route application, it takes an universal address as its parameter, so it may need to solve it to an hypercube address using the Rendez Vous application.

The application will send a notification when the packet arrives:

```
<Notification id="node.testApplication.received">
    <time>126.01 s</time>
    <Data>
        <source>a</source>
        <destination>b</destination>
        <sentTime>126 s</sentTime>
        <receivedTime>126.01 s</receivedTime>
        <elapsedTime>10 ms</elapsedTime>
    </Data>
</Notification>
```

3.6 Filtering the output

The simulator generates many notifications when event occurs; however by default most of them are not written to the output file because they would polute the output file with unnecesary information, making it much harder to read.

When a notification is raised, the Notification Filter checks whether the notification should be written or not. The Notification Filter can be configured from the simulation file.

To make all the notifications to be written, use the command:

```
simulator.notifFilter.setDefault(true)
```

This will instruct the filter to accept every notification that is not explicitely denied.

Still some notifications can be individually filtered:

```
simulator.notifFilter.deny('node.statemachine')
```

In the above example, all the notifications starting with `node.statemachine` won't be shown, so the notifications `node.statemachine.main.waitpap.exit` or `node.statemachine.hbl.listenhb.enter` will be filtered.

When the filter runs, it finds the largest (counting the number of dots) matching possible and obeys that rule. This feature can be demonstrated with:

```
simulator.notifFilter.deny('node.statemachine')
simulator.notifFilter.accept('node.statemachine.main.waitpap')
```

In that case, `node.statemachine.main.waitpap.exit` and `node.statemachine.main.waitpap.enter` are accepted, but not `node.statemachine.hbl.listenhb.enter`.

Using filters is useful to obtain a desired view of the output to analyze different aspects.

3.7 Using XSLT templates

Altough XML files can be directly read, sometimes it's desirable to have a more friendly format. XSLT is a language for transforming XML documents (see http://www.w3.org/TR/xslt). Using this technology, the output files can be transformed to a more convenient format, like plain text or html.

Some samples are provided in directory `examples/templates`. They produce HTML from the XML, focusing on different aspects of the simulation.

To make things easier, the simulator can write in the output file a link to the XSLT desired, using a command in the first line of the simulation file:

```
simulator.formatter.setStylesheet('templates/traceroute.xsl')
```

This will produce the line:

```
<?xml-stylesheet type="text/xsl" href="templates/traceroute.xsl"?>
```

That way, by simply opening the XML file with a browser that supports XSLT (like Firefox), the output will be converted to HTML and displayed in a more friendly way.

A Function Reference

In this appendix, a list of the functions available for each object is presented:

A.1 Connection

A Connection between two nodes. It can have a delay and a bandwidth. If those parameters are not set, zero delay and infinite bandwidth are used.

`query()`	Output information about the connection.
`setDelay(t)`	Set the delay of the connection.
`setBandwidth(bw)`	Set the bandwidth of the connection.

A.2 HypercubeNetwork

The functions written in the simulation file are executed on this object. It handles the nodes and connections, and provides access to the simulator object.

`simulator()`	Get the Simulator object.
`query()`	Output information about the Network.
`setAddressLength(n)`	Set the address length (Dimension of Hypercube).
`newNode(name)`	Create a new node with the specified name.
`node(name)`	Return the node with the specified name.
`allNodes()`	Return all the nodes in the network.
`newConnection(node1, node2, [bw], [delay])`	Create a new connection between the specified nodes, optionally setting the bandwidth and delay.
`connection(node1, node2)`	Return the connection between the specified nodes.

A.3 HypercubeNode

A node in the Hypercube Network.

`query()`	Output information about the Node.
`query(neighbours)`	Output information about the Node and the detail of its neighbours
`assertPrimaryAddress(paddr)`	Verify that the node's primary address is the specified.
`assertSecondaryAddresses (saddr)`	Verify that the node's secondary address is the specified.
`allConnections()`	Return all the connections of the node.
`joinNetwork()`	The node will try to join the Network.
`leaveNetwork()`	The node will leave the Network.
`traceRoute()`	Get the TraceRoute object.
`rendezVousServer()`	Get the RendezVousServer object.
`testApplication()`	Get the TestApplication object.
`routing()`	Get the ReactiveRouting object.

A.4 NotifFilter

The Notification Filter selects which notifications are written to the output file.

`accept(notif)`	Makes the filter accept the specified notification.
`deny(notif)`	Makes the filter to not accept the specified notification.
`setDefault(accept)`	Makes the filter accept/deny a notification that was not explicitely set.

A.5 ReactiveRouting

The Reactive Routing algorithm object.

`table()`	Get the routing table.

A.6 RendezVousServer

The Rendez Vous Server is the application running in each node to manage Rendez Vous tables.

`query()`	Output information about the Rendez Vous Server.

A.7 RoutingTable

Each ReactiveRouting object has a routing table to save information about the routes to different destinations.

`query()`	Output information about the Routing Table.

A.8 Simulator

The simulator object just provides access to the Notification Filter and the formatter objects.

`notifFilter()`	Get the Notification Filter object.
`formatter()`	Get the Formatter object.

A.9 TestApplication

Hypercube Application to test sending packets to other nodes by using their universal addresses.

`send(uaddr)`	Send a packet to the specified universal address.

A.10 TraceRoute

Hypercube Application to trace routes between nodes.

`trace(paddr)`	Trace the route to the node with the specified primary address.
`assert(paddr, route)`	Verifies that the route to the node with the specified primary address is the expected, passed in route. For example, route could be: 'g;h;f;e'. If an empty string is passed, a No Route to Host is expected.

A.11 XMLFormatter

This object does the formatting of the notifications output as XML.

`setStylesheet(fileName)`	Makes the XML output include the specified XSLT stylesheet.