Aurora Performance Monitoring Tool
Programmer's Guide

by Josh Kern

Goals:

Our goal in creating a performance monitoring application for Aurora was to enable users of Aurora to both be able to graphically view the behavior of a running aurora network, and to modify certain aspects of the behavior of the running network in order to tweak the network's performance. Other key goals for our performance monitoring application were that it should be visually appealing and easy to use. We wanted any novice user of aurora to be able to use and understand our application, and we wanted to our GUI to give a user a clear picture of what is really happening when an aurora network is running.

Design Decisions:

There were a number of different ways we could have approached creating our performance monitoring tool. One possibility was to integrate our performance tool into the existing aurora gui which is used to create query networks. We rejected this option because this would have added complexity to an already complex application, and required us to integrate our code with a great deal of code devoted to solving unrelated tasks. Additionally, if we'd taken this approach it would not have been very viable for multiple users at multiple sites to be monitoring the same application at once unless the entire aurora GUI were modified in order to connect to remote servers. Another option we rejected was to use a data visualization tool calle Visionary. This may be a viable alternative option in the future, but we rejected it for the time being because Visionary requires a great deal of overhead in that we would have had to set up additional database management systems and integrated them into Aurora, already a large and complex system. Also Visionary currently only runs on Windows (though this will eventually change) and all Aurora development to this point has been done on Linux. Our eventual decision was to use trolltech's C++ GUI library, QT, to do our development for the performance monitoring tool and to connect our application to the aurora runtime over the network. This solution had a number of advantages. Because our application connects to the aurora runtime through sockets, any number of users at various locations can use our performance tool concurrently to monitor the same running network. Furthermore, because our application is separate from the aurora runtime we were able to, for the most part, do our development independently of the concurrent development going on in the aurora runtime. This greatly simplified our interaction with a large complex system. We decided to use trolltech's QT instead of Java because we (Akash Parikh and I) were both more familiar with QT, and because QT's libraries provided us with considerable functionality enabling rapid GUI development.

Our Application:

In accordance with our goals I've created a GUI application in QT which allows users to manually adjust the degree of load shedding performed on the running network, to monitor the QoS (Quality of Service) the network is providing to applications with respect to latency-based utility, and to view the tuples currently present in the network as well as the drops inserted into the network by the load shedder. Our GUI uses sliders, bar graphs, and heartbeat monitor style graphs to make it easy for the user to control the load shedding and visualize the QoS provided by the runtime. The GUI also draws a picture of the network, showing how many tuples are currently in the system and what drops have been inserted by the load shedder, making it clear to a user what is really happening while an aurora query network is running. For details on using our GUI please see the Perfmon User's Guide (at the end of this document).

Class Overview:

Our performance monitoring GUI tool consists of the following classes:

main.C: Houses main(). This instantiates our DataManager, sets up the appropriate networking for the DataManager, and creates a MainWindow, passing the DataManager to it.

MainWindow: This is a subclass of QMainWindow and holds the QTabWidget containing the two tabs of our gui, the "LoadShedder" tab and the "QueryNetwork" tab. Before the MainWindow creates the LoadShedPage and the NetworkGuiPage it calls blockingHasData() on the DataManager in order to populate the gui with initial data. This class is also responsible for setting the menus (Quit is currently the only menu option) and for all interactions between the DataManager and the Gui. When the MainWindow is instantiated it sets up a timer which periodically calls the function updateGUI(). This function uses the DataManager to read the runtime stats used by the GUI and passes them along to the appropriate gui components which will display them. The only other current interaction between the GUI and DataManager is that when a user adjusts the load shed slider the MainWindow's updateLoadShed() is called which then tells the DataManager to set the level of loadshedding. This call is eventually propagated by the DataManager over the network into the aurora runtime and sets the level of loadshedding on the currently running network.

DataManager: This class is not really part of the GUI but important to mention here as it is the class which enables communication between the GUI and the Aurora runtime. It has a method called readStatsImage() which the MainWindow calls to get the runtime stats object from the aurora runtime. It is important to call releaseStatsImage() after calling readStatsImage() so that the image can be written to by the aurora runtime periodically.

LoadShedPage: This is the toplevel component of the LoadShedding page in the GUI. The constructor takes the number of output applications in the running network. This is one reason it is important that the MainWindow wait until the DataManager has data before it instantiates the LoadShedPage. The LoadShedPage creates QOSMonitors for both latency and utility, based upon the number of output applications. The LoadShedPage also creates a LoadShedWidget (containing the LoadShedGraph, sliders, and some text labels). The LoadShedPage provides various functions to update the QOSMonitors with new values and to update the bars on the LoadShed graph.

LoadShedWidget: Widget created by LoadShedPage which contains QLabels, the LoadShed slider, and the LSGraph. This class connects the slider to various slots (i.e. updating the LCD display, updating the LoadShedGraph's selected bar, etc.)

LSGraph: The LSGraph is a QCanvasView which shows a canvas containing a bar graph. This class uses a vector of doubles provided by the runtime stats object (from the LoadShedder) to draw a bar graph. The vector of doubles provided by the aurora runtime is understood to be non-decreasing. Aside from storing the points of the graph, the LSGraph maintains the current position of the loadShedSlider, and the currently selected bar within the bar graph. The selected bar is shown as blue, and the number of the bar is displayed above it, whereas all the other bars are red. This makes it really clear to the user which degree of loadshedding is currently selected. The LSGraph has two primary functions, one to update the entire graph (updateGraph()), providing it with new values with which to draw bars, and one to update the position of the slider on the graph (moveSlider()). moveSlider() updates the position of the line reflecting the slider's position relative to the graph, and also resets the currently selected bar on the bar graph. It also sends a signal telling the aurora runtime that the loadshed value has been modified.

QOSMonitor: QOSMonitors are vertical layouts which contain some qlabels and a qcanvas on which to show the actual qosvalue. QOSMonitors take an integer (specifying which applications QoS the monitor is showing) and an enum QOSTYPE in their constructor. The QOSTYPE enum is defined in GuiConstants.H and is used to specify whether the QOSMonitor is displaying latency or utility. Because both monitor types work the same way functionally, I was able to easily use this class to show both types. If a third type of QOS is introduced, monitors displaying it can easily be added via the QOSTYPE enum. QOSMonitors have two important methods, and update(). The update() method is important for all QOSMonitors. It takes an integer value between 0-100 and uses this value to draw a red line showing the current QoS value. The number of lines present in each QOSMonitor is a constant in GuiConstants.H. It is currently set to 50, meaning that the QOSMonitor can display a history of at most the last 50 snapshots of the QoSValue. If all 50 lines are currently being drawn, it shifts all the previous lines backwards before drawing the newline, reflecting an updating history of QoS. The height, width, and number of lines (history length) of QOSMonitors can all be easily adjusted by modifying constants in GuiConstants.H.

NetworkGuiPage: This is the toplevel component of the "QueryNetwork" tab. It is a QCanvasView showing a canvas containing the aurora network. The NetworkGuiPage takes a string in its constructor representing the directory containing the catalog of the currently running Aurora Network. One reason the QMainWindow waits for the DataManager to be populated with data is to pass this string to NetworkGuiPage. The NetworkGuiPage's constructor creates a CatalogManager from the passed-in string and gets a pointer to the currently running Query Network from the CatalogManager. It then draws the actual network. It would be needless to redraw the Query Network every time the DataManager updates which is one reason for this design. Only the parts of the Query Network which change each time are redrawn. The primary methods of NetworkGuiPage are redrawNetwork() and updateNetwork(). redrawNetwork() is somewhat of a misnomer since it is only called once in the constructor currently, but if changes were written to the catalog while a network was running (as may be possible in future aurora implementations) it would make sense to recall this function. redrawNetwork() deletes the current network (if one exists) and then sets up maps of QueryArcs and QueryBoxes and draws the inputs, outputs, arcs, and boxes of the network. The updateNetwork() function gets passed
map of numbers of tuples per arc and a map of strings describing drops per arc. It uses these maps to modify QCanvasObjects stored in QueryArcs and update their visual appearance.

QueryBox: This class is essentially a wrapper class around Box, a class existing in the aurora runtime to represent a box operator in a query network. The QueryBox contains a pointer to the box it holds as well as pointers to gui objects representing the visualization of the box. The QueryBox also contains all the input and output ports of each box, something not store
in the actual Box class. QueryBox has a series of methods to access and set gui objects and port
which pertain to a box.

QueryArc: This is a wrapper class around Arc, a class existing in the aurora runtime to represent a connection between two boxes in a query network. In addition to storing a pointer the arc it represents, a QueryArc stores pointers to gui objects displaying the number of tuples in the arc and information pertaining to drops set on arcs by the loadshedder. QueryArc has a series of methods to access and set gui objects representing the visualization of an arc.

GUIConstants.H: This is a header class holding several constants pertaining to the appearance and size of various GUI elements. These can be modified with no changes to other parts of the code to alter the appearance of the GUI.

Difficulties in Creating Our Performance Monitoring Tool:

There were a number of issues we had to confront in attempting to create a performance monitoring application for aurora. The first difficulty, as mentioned in my discussion of design decisions, was in choosing how the performance monitor would be connected with the aurora runtime, and what tools we would use to code it with. There was a great deal of internal discussion on this issue, and we spent a lot of time trying to reach a consensus, which took away from the time we had to actually spend on development of our tool. Faced with time constraints we eventually opted to use QT primarily because it enabled me to do our GUI development quickly. As I mentioned earlier, using Visionary may be a viable option for the future, (although it would be a shame to throw away all the work that has been put into the existing GUI).

Another key issue for us to face was how the GUI should look, and what information it would display. In order to facilitate this discussion, I worked on the GUI quickly, creating demos of it to show to people on the aurora project in order to hear their opinions and criticisms. Changes generated via this discussion included switching the slider from horizontal to vertical, creating a bar graph, rather than drawing lines on a graph to illustrate the load shedder curve, using tabs to separate different aspects of our performance monitoring tool, and drawing gray lines in the QOS utility monitors showing expected utility vs. computed QoS utility for the sake of comparison.

A very important issue we faced throughout the development of this project was our interaction with the aurora runtime in terms of how it passed information to us, and our dependencies on other people working to get the information we needed into the aurora runtime. We had several discussions regarding not only what information the gui needed (i.e. the string specifying which catalog to access the query network from), but also what the structures passed between the gui and the aurora runtime should be called. Also, many of the structures which we display in the gui didn't actually exist yet in the runtime while we were making the gui. To ensure that my gui code was functional when receiving the information it was not yet getting from the aurora runtime I created some objects which would continually create fake or "spoofed" data and pass it to the gui. This enabled me to test and repair my code and get aroun dependencies on information from the aurora runtime.

Drawing the aurora query network was an especially difficult problem I had to face. One of the first problems I had to deal with was that the Box class did not store any information regarding its position within the aurora network. The aurora gui, (not to be confused with our performance monitoring gui) which is used to create query networks, was written in Java and thus used a totally different interface than what we had access to, writing in C++ but it did store positions of boxes in the catalog so I knew there had to be a way to discover the positions of boxes. After spending some time delving into the system, I eventually realized that BoxRecords, from which Boxes are created, stored the positions of Boxes and thus I was able to solve this problem by adding accessors to Box and modifying the constructor. I also had to change all Box subclasses to pass the appropriate position information to their superclass. I gave all parameters added to constructors default values, so that I wouldn't accidentally interfere with anyone creating Boxes in other parts of the aurora code.

Although I now was able to position boxes as they were positioned in the aurora gui for creating networks, I was still missing information. The location of input and output ports is not stored in the catalog at all, so I had to find a way to position these in a logical manner. Also, the number of inputs and output ports per box is not stored in Box. Arcs store which ports of each box they connect to on each end. Thus my solution was to preprocess all the arcs in the network to discover which ports were on each box and then to intelligently space the ports on the box afterwards. This was one of the reasons it was necessary for me to create wrapper classes around Arcs and Boxes in trying to display a picture of the running network.

In summary there were many issues involved in trying to start a new subproject which would interact with a large system still under development. I had to, on various occasions, find clever ways to tweak the system in order to use it as I needed without interfering with the work of other developers. Also there was a great deal of internal discussion and communication about what our system should include, and how it should interact with the existing aurora system, which always tends to slow down development.

Future Possibilities:

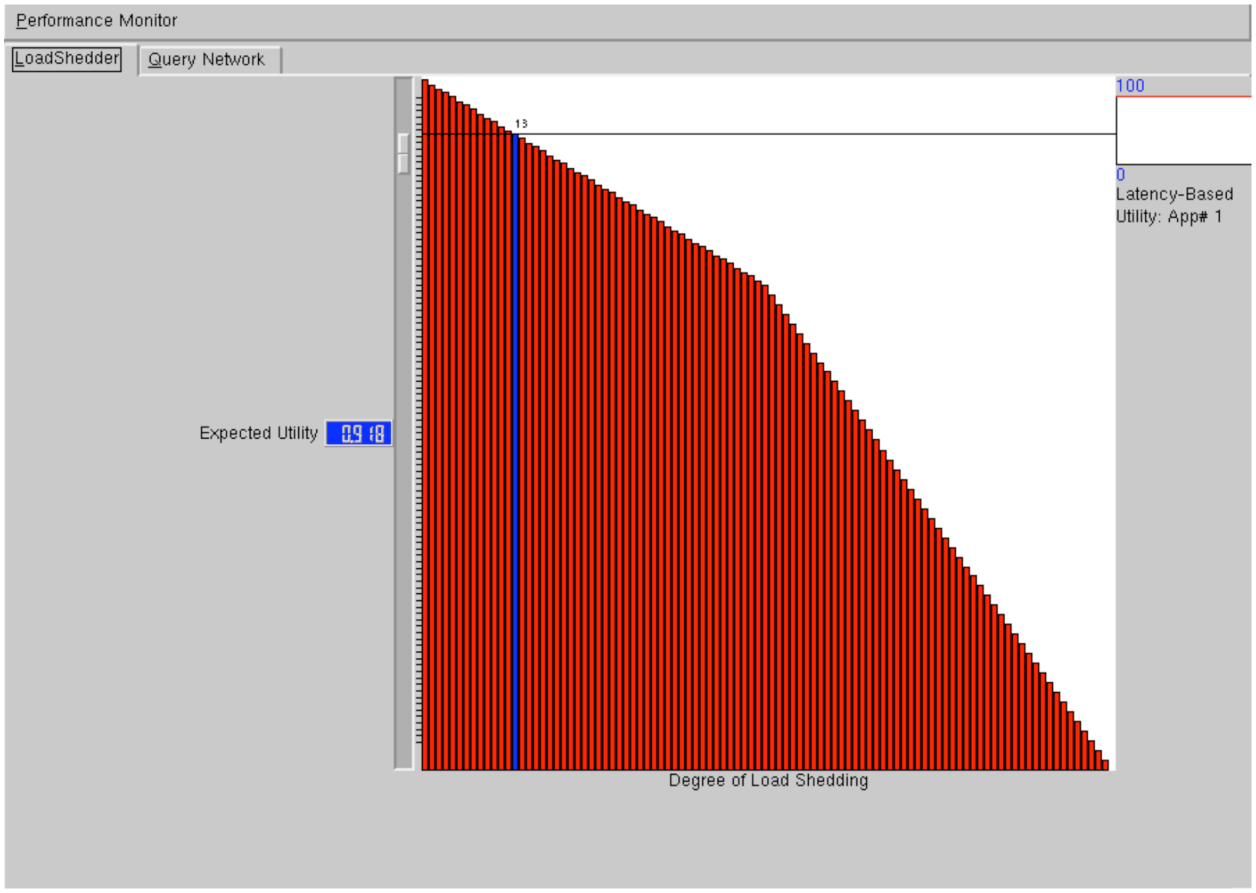
There are many things which are not presently part of our Performance Monitor Tool which we would have liked to add if we had had more time. Currently the “Query Network” tab shows the current state of the network, but there are additional things we could show in the future. For instance, there are boxstats produced by the aurora runtime such as the cost or selectivity of each box which we aren’t currently showing in the Query Network. It would be nice to show these for each box when a user mouses over a particular box in the Query Network. Currently we aren’t actually passing the boxstats from the runtime to the gui, because a) we are not using them, and b) there seems to be a problem with the way we currently attempt to serialize these statistics. If the serialization were fixed, and the networking code adjusted to send these boxstats over we could display them in the query network. To do this, one would nee to either convert QueryArc and QueryBox into subclasses of QCanvasItem and display them directly on the canvas, or one would need to not use a canvas at all to show the query network, and convert QueryArc and QueryBox to widgets. The second method is probably more work but enables more flexible and powerful features. The reason I originally decided to use a QCanvas was that QCanvases draw very quickly and I didn’t realize I was going to want to allow the user to interact with the network.

Another nice thing to allow the user to do with query network would be to adjust the inputRates of tuples in each box. One could do this by attaching a Widget to each box, containin some kind of slider controlling the inputRate. This would allow the user to really experiment with the network, tweaking its performance beyond just adjusting the loadShedder. Also it would be nice if we could use splines to draw the network arcs as the aurora gui does, rather than straight lines. Unfortunately I don’t believe any information regarding the splines used in the aurora gui is written to the database so we can’t replicate the gui’s network picture precisely. This is also true of the inputs and outputs of the query network, which should store positioning information but don’t currently.

It might be nice to have some control of other things in the performance monitoring tool as well. For instance, it might be really useful if in the performance monitor there was some way to adjust the scheduler. One might be able to switch between round-robin and whatever other scheduling algorithms are available to see which scheduler produces the best performance for a particular network. It also might be nice if in the future if one could control the workload generator from the performance monitor, rather than from within the gui where it currently resides. It might make more sense to have it within the performance monitor, because then one could adjust it and immediately see how the network handled increased (or diminished load) and how it might respond to gradually changing loads. The aurora gui presently creates networks, but doesn’t run them, so it is perhaps more logical if something so intimately connected to running networks as the workload generator could be controlled in an application which actually shows a running network.

Aurora Performance Monitoring Tool
A User's Guide

by Josh Kern



(Performance Monitoring Gui Overview)

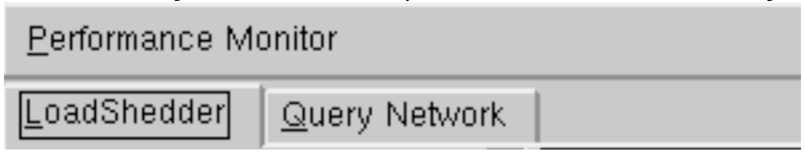
Functionality:
-Choose a degree of load shedding by specifying expected aggregate utility for a running network
-View the current QOS levels of each application output with respect to latency-based utility
-View the state of the currently running aurora network in a visually appealing manner
-View the drops inserted into the running network by the load shedder

Running the performance monitor:
Type "$AURORA_SRC_DIR/perf_mon/bin/perf_mon_gui <hostname> [updateSpeed]" to start the performance monitor.
<hostname> is the name of the machine on which there is a currently running aurora query network.
[updateSpeed] is an optional parameter specifying how often in milliseconds the gui should refresh itself with the latest information. If new information is not available the gui just redisplays its current information. For the most part, this value should be left at its default (2000 or 2 seconds). If the update period is too small, one will needlessly

refresh the data, and also make the qosmonitor values repeat meaning less of the qos history will be maintained. If the update period is too great, one runs the risk of missing data and making the qos values seem less consistent than they actually are.

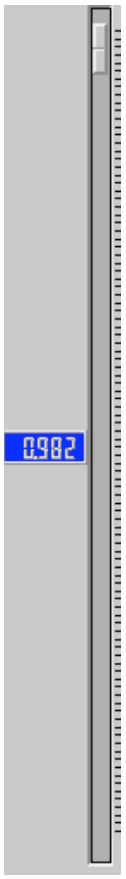
* (You may run as many performance monitors concurrently as you like, it is possible to monitor a network from distributed sites)

The gui will start up after about 10 seconds (it waits to be populated with initial data before launching). When it comes up you will see two tabs at the top of the screen, “LoadShedder” and “Query Network”. “LoadShedder is the initially selected tab. In the "LoadShedder" tab you will see a slider, a graph, and several QOSMonitors (labeled as ‘App Output # <num>’). In the Query Network tab you can view the performance of the currently running network.



(Performance Monitor Tabs, LoadShedder tab initially selected)

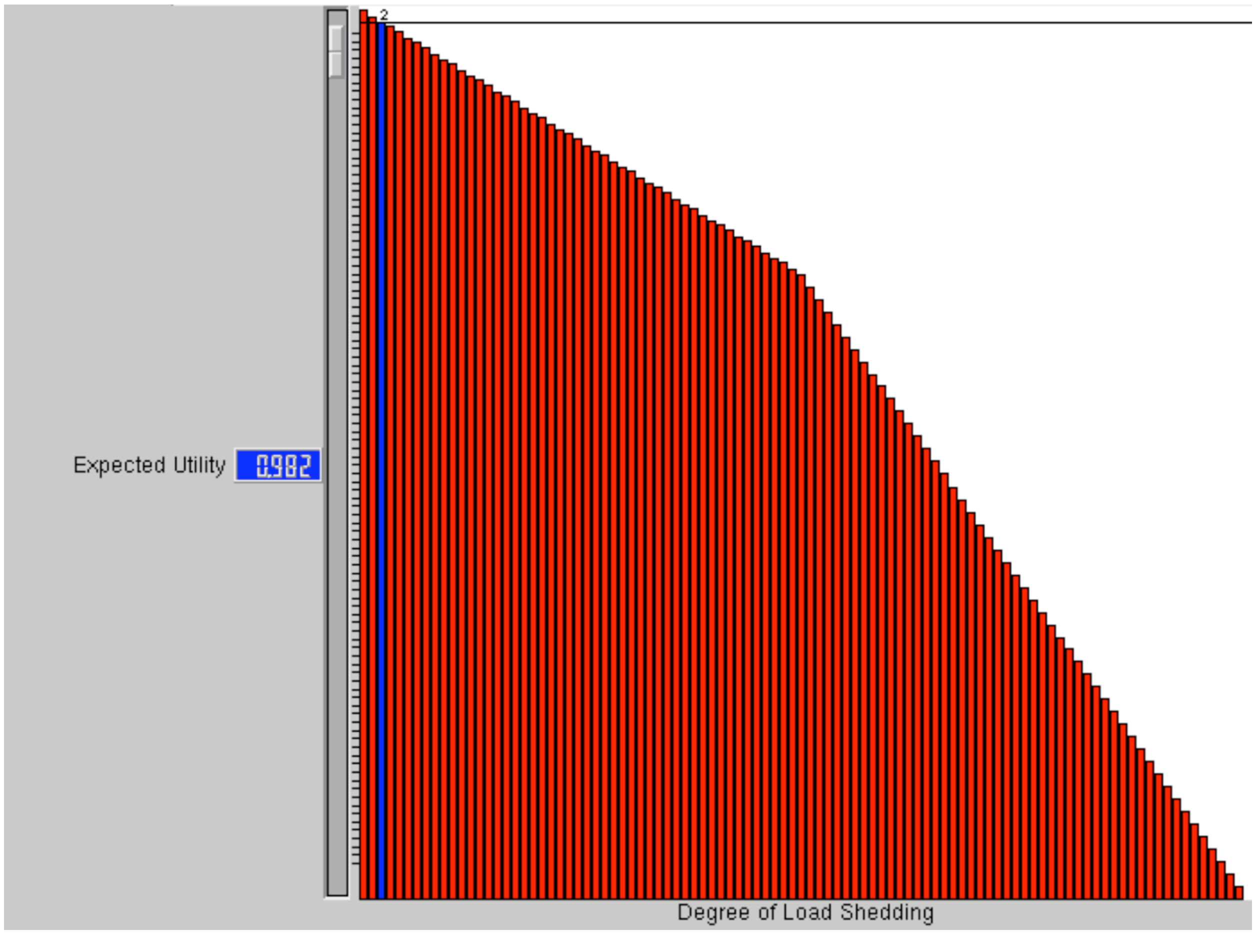
LoadShed Slider:
The loadshed slider in the “LoadShedder” tab is used to specify a desired level of expected aggregate utility for the network. The expected utility for the network is initially 1 (100%) as there is no load shedding being specified. As you move the slider down and lower the expected utility, the level of load shedding will increase. A blue box to the left of the slider indicates the current level of expected utility.



(LoadShed Slider)

Graph of Utility vs. Degree of Load Shedding:

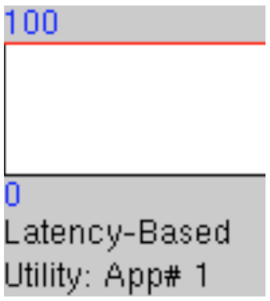
The graph of utility vs. load shedding in the “LoadShedder” tab shows a series of red bars, with one blue bar representing the currently selected level of load shedding. The bars on the graph will change over time as load shedding plans vary according to the query network you are dealing with. The bars indicate degrees of load shedding corresponding to expected utility levels. O is the lowest degree of load shedding (no load shedding) while the highest degree varies according the network.



(LoadShed Graph and Slider)

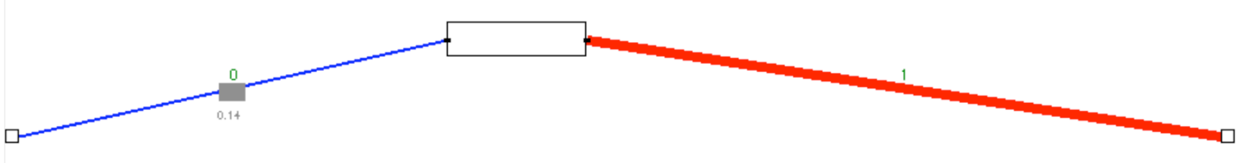
QOS Monitors:

The QOS Monitors are in a column in the “LoadShedder” tab to the right of the LoadShed Slider and Graph. Each monitor represents an application output of the currently running network. Currently QOS monitors show latency-based utility. Latency-based utility QOS Monitors use red lines to show the computed latency QOS (to what degree the applications’ time-constraints for tuples are satisfied) for each application output. As load shedding increases the utility reflected in utility monitors will likely, though not necessarily, go down.



(Latency-based Utility monitor)

Query Network:
The currently running query network is shown in the “Query Network” tab. The network is displayed as a series of boxes connected by arcs (very similar to those shown in the aurora gui). The tiny square boxes on the left and right ends of the network are inputs and outputs respectively. Also shown are tiny black rectangles which are individual ports on the boxes. Above each arc in the query network is a green number which tells how many tuples are currently on the queue waiting to be consumed by the destination box. The numbers will continually update as per the state of the network. When drops are inserted into the network by the loadshedder, gray rectangles will appears on the arcs where the drops were inserted, and the predicate specifying the drop will be shown in gray text below the arc. The arcs connecting the boxes vary in thickness and color depending upon the relative number of tuples waiting in the queue for each box. When an arc gets thicker and redder this means that a high percentage of tł tuples currently in the network are waiting on the queue corresponding to that arc. As the number of tuples in that queue decreases relative to the number of tuples in the whole network, the arc will get thinner and bluer. The feature of thickening/thinning and color-changing arcs provides the user with immediate visual feedback of where the bottlenecks in the system lie, and may help the user to define a more efficient query network. It also helps the user to get a clear overview of the current state of the query network and of how well the aurora system is performing as a whole.



(Sample Query Network, 2nd arc contains one tuple while 1st arc is empty and has a drop inserted)