IsisInstrumentsLib

This API will allow you to create an ***IsisInstrument*** object, which can log a variety of information regarding a run of Isis^2. When you are ready to starting logging, simply call ***IsisInstrument(run_number)*** with some GUID. This will start a timer using ***IsisSystem.NOWus()*** and launch a thread that will gather all ISIS RTS data every 10 milliseconds. This interval can be adjusted by modifying the ***Period*** attribute. Once an instrument object has been created, you can register the instrument with your ***IsisGroup*** using ***myInstrument.RegisterInstrumentation(myGroup)***. This will enable the logging of internal events (please speak with Ken Birman to learn more about which events are being logged and the format of the logs produced).

Assuming that Isis Instruments is being run on multiple nodes at once, you'll want to synchronize the timers using some reasonable synchronization algorithm. I have included a ***Synchronize(offset)*** function that will enable all subsequent logged entries to use the adjusted time. You'll want to perform a round trip send with a leader of the group in order to discover the offset between each member's individual timer and the leaders timer before proceeding with your experiments. An example of this is available in Qi's InteractiveDemo, including the SVN repository for ISIS^2.

Once the timers have been synchronized, you can now proceed with your experiment. You can log events in your system using:

AppInstrumentationFunction(IsisInstrumentLogType type, Address sender, long LId, params object[]args)

The type of an ***IsisInstrumentLog*** instance is simply an identifier for the kind of event that occurred. I've included an enumeration ***IsisInstrumentLogType*** that includes a variety of common events that you may want to capture during a run of your system, though you may add more as needed. Only **append** to this list unless you're willing to break compatibility with older logs. The ***sender*** of the event is the casual origin of the event being logged - in the case that the event is a receive, the origin is simply the sender. This information is essential in order to reconstruct the causal history of the event. The ***LId*** along with the address of the node at which the event occurred will uniquely identify the event - typically I use an Application level message identifier as the ***LId***. Finally, the ***args*** parameter will allow you to log any additional information you might want to include.

As a run of the system occurs, ***IsisInstrumentLog*** instances will fill an internal ***List*** held in memory. Periodically, you'll want to call ***Flush()*** in order to write the logs to disk and clear the internal list. When the experiment is finished, be sure to call ***Shutdown()*** to perform the requisite cleanup. When you are finished, your node will have written the logs to the file 'logs/***runnumber_isisaddress***.ilog' seralized as JSON objects. You'll want to copy all these ilog files into a common directory for the analyzer.

IsisAnalyzer

This CLI program will allow you to put together graph data from the raw log files. To do this, you need to specify a path to the directory in which the log files are located, as well as a run_number to identify the specific files to be analyzed. Each file will be parsed individually, and the information deemed important will be added a master list of events (***raw_events***). For debugging purposes, I also keep track of user readable strings for each of these events in ***events***. The rest of the events are discarded. In first pass, I am also keeping track of one instance of RTS data for every raw_event added. The keys corresponding to each raw_event is stored in the dictionary ***to_process_rts_data*** and the RTS data retained is stored in ***associated_rts_data***, both of which are keyed by (***ServerName***, ***Timestamp***).

After this first pass is complete, a second pass is required to reconstruct the causal history of each event that occurred. In particular, for every send event that occurs, we'd like to identify the corresponding receive events. Right now, the code for doing this is overly complex and inefficient, but the concept behind what it needs to do is pretty simple. Each send event can be uniquely identified by the ***LId*** and the ***sender*** address. All that we need to do is group the related receive events by the tuple (***LId***, ***sender***) and output all the data points (along with these relationships) to a format that IsisChart can read and graph. Currently, the output for IsisAnalyzer is serialized as JSON into the file output.txt.

IsisChart

This part of the project is relatively straightforward. All the code is contained within the javascript embedded in ***test.html***. ***render_graph()*** performs most of the work. The data for the graph will be located in a file with the extension '.txt'. This file will just be a JSON file with a few important properties. ***data.nodes*** is just a list of all the nodes, identified by an integer id. data.messages is a list of all the messages, which themselves include a message name (called "***message***"), a ***sendEvent*** (which itself is a list of data including ***nodeId***, ***timestamp***, and an RTS data structure called ***rts***), and a list of **receiveEvents** that correspond to each **sendEvent**. All the events will be plotted on the graph (with time along the x-axis, ***nodeId*** along the y-axis) and connected to their corresponding ***receiveEvents*** using lines. The RTS data for each event is revealed on hover over using the ***tooltip*** parameter. Finally, filters are provided to filter the number of events being plotted. The ***is_included(k,len,filters)*** function will determine which filters have been checked, and is used in the ***render_graph()*** when creating the series for each node in the graph. Note that the number of line-graphs being produced is at most ***nodes.length*** * ***message_events.length***, as this is the total number of send-receive pairs.