How to Create a Feed for the Deep Node Console

Examples in this document are all Java code. Apologies to the world of the untyped.

To feed data into the console, open a socket to the console and write using Deflater. By default the console listens on port 4021.

```
s = new Socket("localhost", 4021);
pw = new PrintWriter(new OutputStreamWriter(new
                                    DeflaterOutputStream(s.getOutputStream(), true)));
```

The first line you write to the socket is the name of your feed.

```
pw.println("example_feed");
pw.flush();
```

You'll need to periodically flush the output stream so that data will flow end to end without buffering for too long. We use 200 milliseconds. We also check for a response from the console at that time so that we can close up the socket and retry if the console is no longer up. Here's our loop for doing that. Note that the console can send a command back after the "pong"... you don't need to worry about that if there are commands implemented in your feed. Note also that if we don't have anything to send to the console, we sleep the thread - so it doesn't crank the cpu endlessly polling the queue we are using.

```
private void monitor () throws IOException {
  boolean isGood = true;
  long lastPing = 0;
  while ( pw != null && isGood ) {

    if ( lastPing == 0 || System.currentTimeMillis() - lastPing > 200 ) {
      pw.println("ping");
      pw.flush();
      String line = br.readLine();
      if ( line == null || !line.startsWith("pong") )
        isGood = false;
      else if ( line.startsWith("pong packdet ") )
        sendPacketDetail(line.substring(13, line.length()), pw);
      lastPing = System.currentTimeMillis();
    }

    String str = outQueue.poll();
    if ( str != null )
      pw.println(str);
```

```
    else {
      try {
        Thread.sleep(50);
      }
         catch ( InterruptedException ie ) {}
    }
  }
}
```

Now, you simply write messages to the socket; each message is a line in the following format:

<uid>TAB<time>TAB<from node>TAB<to node>TAB<count>PIPE<severity>

<uid> is just a string which should be unique to the message being sent; just an incremented integer will do, sending guids could be a huge waste of bits

<time> is the string representation of the long value which is the number of milliseconds since the Unix epoch.

Both the <from node> and <to node> have four levels of hierarchy, and are in the following format:

<level 1>PIPE<level 2>PIPE<level 3>PIPE<level 4>

The <count> is an integer measure, and controls the diameter of the cones that are displayed in the console. The <severity> is a float between 0.0 and 1.0, and controls the redness of the cones, with 1.0 being completely red and 0.0 being completely silver.

Suppose we are creating a basic network traffic feed. We first have to pick our four levels of hierarchy, and could do something like this:

NETWORK | DOMAIN | HOST | PROTOCOL+PORT

NETWORK could be “internal”, “external”, and “dmz”. For internal traffic, DOMAIN could be the subnet. We’d end up sending messages that look like this:

12345678\tinternal|192.168.1.0|192.168.1.35|tcp54627\texternal|1e100.net|208.35.2.134|tcp80\t20|0.0

12345679\texternal|1e100.net|208.35.2.134|tcp80\tinternal|192.168.1.0|192.168.1.35|tcp54627\t500|1.0

These example messages would indicate that a machine on the 192.168.1.0 internal subnet made a HTTP request to a server in the 1e100.net domain, and received a 500-byte response that triggered some kind of alert (hence the severity of 1.0).

As your feed writes lines to the console, the console dynamically creates a 3d tree representing the hierarchy in your <from node> and <to node> message components, and sends “blips” down this tree which represent the <count> and <severity> values received over time.

Here’s some example code from the pcap feed; once values have been gathered from a sniffed packet, this code puts them together into a message and sends it to the console:

```
StringBuffer sb = new StringBuffer(String.valueOf(nextId));
nextId++;
sb.append('\t');
sb.append(String.valueOf(System.currentTimeMillis()));
sb.append('\t');
sb.append(sourceLocation);
sb.append('|');
sb.append(sourceAddr);
sb.append('|');
sb.append(sourceProt);
sb.append('\t');
sb.append(destLocation);
sb.append('|');
sb.append(destAddr);
sb.append('|');
sb.append(destProt);
sb.append('\t');
sb.append(String.valueOf(len));
sb.append('|');
sb.append("0");

String str = sb.toString();
if ( pw != null ) {
  pw.println(str);
  pw.flush();
}
```

Much of the code in the example feed and the pcap feed have to do with managing the socket. The console will respond with a single line of “pong” whenever it receives a single line of “ping”. This allows a feed to detect bad sockets and reconnect to the console.

Tagging Messages

Your feed can define tags which affect the appearance of blips in the console by adding 3d icons to their edges. After connecting to the console and sending the line with its name, the feed can send lines which define tags. Here's the format:

__td_<tag>|<shape>|<r>|<g>|<b>|<effects>

Each line sent with the "__td_" prefix defines a tag. The attributes of the tag are delimited by pipe and are:

<shape>: "text", "sphere", "torus", "cube", or "cone" (more shapes to come as we develop them…)

<r>, <g>, <b>: float values between 0.0 and 1.0, defining the color of the tag

<effects>: "none" or a comma delimited list of effects; currently "spin" and "pulse" are available

The example feed code defines its tags the following way:

```
pw.println("__td_tag1|sphere|1.0|.3|0.0|pulse");
pw.println("__td_tag2|torus|0.0|.4|.9|spin");
pw.println("__td_tag3|cube|1.0|.2|.2|pulse,spin");
```

Now that these tags are defined, any message can be tagged with one of them. Taking our example messages from above, we could modify them to look like this:

12345678\tinternal|192.168.1.0|192.168.1.35|tcp54627\texternal|1e100.net|208.35.2.134|tcp80\t20|0.0\ttag1
12345679\texternal|1e100.net|208.35.2.134|tcp80\tinternal|192.168.1.0|192.168.1.35|tcp54627\t500|1.0\ttag2

All you have to do to tag a message is add a tab, and then the tag. The corresponding icon will be displayed on the blip. As blips aggregate, they can accumulate multiple tags; the size of the tag icons will be proportional to the amount of data within the blip that was tagged that way.
If you don't define the tag before you use it, it is just text displayed on the blip.