

Building a front-end for netchar

Eric Wing
University of California, San Diego
ewing@ucsd.edu

Richard Phipps
University of California, San Diego
rhipps@ucsd.edu

Mai Nguyen
University of California, San Diego
mnguyen@cs.ucsd.edu

John Datuin
University of California, San Diego
jdatuin@cs.ucsd.edu

Abstract

Several tools have been developed to gather information to characterize links of the Internet path between any source and destination. Such tools, such as pathchar, clink, pchar, and netchar, generate text output that can be difficult to analyze for a sufficiently large data set. In this paper, we describe the design and implementation of a front end for netchar. The front end is a graphical user interface (GUI) that presents a visual representation of the output generated by netchar's data-processing back end. The user can select specific graphical objects in the display and view the data associated with each object. We also discuss issues and problems that we encountered during the creation of the front end.

1 Introduction

The current tools for estimating link characteristics are similar to traceroute and ping. The use of the time-to-live field, or ttl, field of the IP packet header is what is primarily used to determine the path between a source node and a destination node. By setting the ttl field of a packet to a value n , then at least n links should be traversed by the packet

before expiring. If a router receives a packet that has expired (i.e., the ttl field is no longer valid for the packet to be forwarded to the next link), it drops the packet and sends an ICMP error packet back to the originating source. The ICMP packet contains the address of the router sending the ICMP packet that the source can use to determine the path.

By varying the sizes of the packets to transmit and measuring the times when ICMP error packet are received, a source node can determine the bandwidth and latency [1, 5]. By sending multiple packets of each varying size and ttl value, queuing delays and packet loss information can be computed [1, 5, 6].

There were several constraints that were realized during the development of these tools. The constraints were that they were to execute only on the end hosts, they were to be transparent to the actual network traffic, and they were not dependent on any particular network protocol [5].

In the following section we give some background information on the evolution of these tools. Next we describe netchar. Next we present the front end for netchar and its major components. Finally, we discuss issues and problems encountered during the initial design stage through the integration and testing portion, describe possible future work, and conclude with a summary.

2 Background

In 1997, Van Jacobson developed the tool, pathchar, to find the link characteristics between any source and destination on the Internet as a means to study the problems of congestion [4]. pathchar is similar to traceroute in its use of IP's time-to-live (ttl) field. Nodes along a path between a source and a destination are found through the ICMP error packets that get sent back to the source from

routers that receive expired packets sent by the source. pathchar is different in that it sends out a series of probes with varying ttl values and packet sizes, that, through statistical analysis of the measured times when the ICMP packets are received, link characteristics such as bandwidth, delay, average queue, and loss rate of every hop can be inferred [1].

Though source code was (and still is) not available for pathchar, Allen Downey, starting in 1998, developed a similar tool called clink [2]. clink is an independent implementation of pathchar, written in C using Linux, and is based on Downey's own simple version of traceroute, trout [2]. In 1999, he presented a paper [1] on his experience using pathchar and proposed some ideas on improving its performance.

In 1999, Bruce Mah introduced another independent implementation of pathchar called pchar, which is based on algorithms from pathchar [6]. pchar is written in C++, has support for IPv6, and works on multiple platforms (BSD, Solaris, Linux) [5].

3 What is netchar?

Drawing upon the results from their own independent implementation of pathchar, Allen Downey and Bruce Mah developed netchar as a means to extend pathchar, clink, and pchar [3]. Netchar, along with the other tools, operates by sending out multiple packets, or probes, of varying sizes and ttl values towards its destination. By measuring the round-trip time from sending the packet to receiving an ICMP message, Netchar can determine the link characteristics from a source (running netchar) to a destination. Netchar has the added ability to send probes from multiple hosts (granted that each host has the netchar software and the execute permissions are set appropriately) to provide a graph of the network connecting the probing hosts and the

links that appears in the paths between them [3]. The advantages they expect to provide over pathchar include: more accurate estimated link characteristics, better handling of route changes and path asymmetry, repeatability of measurements, and reduced data collection overhead [3].

There are three components that form netchar: 1) the data collection infrastructure (based on pchar), 2) the data processing back end (based on clink), and 3) the graphical front end. The data collection portion is responsible for sending packets (i.e. probes), recording and storing the timing measurements in database files, and sending these files to a central location for processing [3]. The back end portion is responsible for processing the collected data to generate the link characteristics and graph of the path between the source and destination [3]. The front end is responsible for providing a graphical visualization of the data generated by the back end portion.

4 The front end

The front-end component is a graphical user interface that presents a visual representation of the data collected and generated by the data collection and data processing portions of netchar. In the GUI, the user can click on the node and link objects of the displayed graph and view the characteristics associated with each object.

The front end for netchar consists of three main components: the parser, the graph-layout engine, and the graphics driver.

4.1 The parser

The parser component is a Perl script that parses the text output generated by the back end into uniquely formatted data files that is used by the graph-layout engine. The parser currently produces a node

data file and a link data file for each input data set.

The node data file contains a list of all the unique nodes in the graph (i.e., no duplicates nodes are present), as well as the information associated with each node (i.e., unique ID number, IP address, nickname, and full name). The link data file contains a list of all the unique links in the graph as well as the information associated with each link (i.e., unique ID number, which path the link belongs to, the start node of the link, the end node of the link, the three sets of bandwidth information computed and their range, and the number of probes sent along the link).

4.2 The graph-layout engine

The graph layout engine is composed of two main parts: 1) a graph maintenance object, and 2) a set of graph layout algorithms. The graph maintenance object reads in the node and link data files generated by the Perl parser, and constructs an internal representation of the described graph. All graph information and actions are carried out through this graph maintenance object, including traversing the graph nodes, links, and querying the graph for specific information, such as the minimum bandwidth in the graph.

The graph maintenance object maintains a list of all the nodes and links within the graph through a list of node and link classes, respectively. A node object (i.e., an instantiation of a node class) contains all the information given about a particular node, such as the IP address, nickname, and full name. Likewise, a link object contains all the information for a particular link, such as path number, the nodes it connects, and statistical information such as bandwidth characteristics. Both objects are easily extendable to contain future parameters.

The graph maintenance object provides a set of query routines from which to get graph information, such as information on global bandwidth or bandwidth of a particular link, as well as addition routines to parse through different paths within a topology.

The graph layout algorithms allows for customization of the layout of the graph on the screen. All layout algorithms exist as independent objects and may be handed to the graph object at any time to re-layout the graph with a new algorithm. This ensures and enforces an orthogonal relationship between the layout algorithms and the maintenance portion of the graph engine, allowing easy expansion of the types and choices of layout algorithms in the future.

4.3 The graphics driver

The graphics driver is written using OpenGL, GLUT, and GLUI. The OpenGL Utility Toolkit (GLUT) is an API that extends OpenGL to write window system independent OpenGL programs. GLUI extends GLUT and is a C++ GUI API that provides more controls, such as buttons, check boxes, radio buttons, and list boxes.

Since the graph layout engine stored the node and link data in lists that are accessed via query routines, our solution to drawing the graph was through the concept of a connectivity matrix. In essence, it is a 2D matrix of size Nodes x Nodes, that contains connection information. In its simplest form, the each cell contains a 1 if there is a link going from i to j, or a zero otherwise. This matrix representation gives us a global perspective of connections so we do not get lost following paths. Furthermore, the matrix allows us to easily represent asymmetric paths as Matrix[i][j] is different than Matrix[j][i]. We go into more detail about the graphics driver in the next section.

5 Design Issues and Problems

From the beginning, we knew that work on the front end would be a challenge, if not a monumental effort for all of us. We did know that we needed to do some text file manipulation (which Perl was the obvious answer) and some graphics. We tossed around some ideas of using Java, as well as some Java graphics packages, but since C/C++ was much more to everyone's experience, we decided to stick with C/C++. Everyone, with the exception of Eric, did not have any Perl or OpenGL experience, but we decided to go with Perl and OpenGL.

For our development environment, some of us wanted to work in MS Windows, others in Unix/Linux, and another in BeOS. Since everyone had a PC at home, and coincidentally had a version of Linux running on their machine, we decided to stick with Linux.

The next step was in trying to divide the work. Since trying to learn Perl and OpenGL would be too much, it was decided that some of us would work on the Perl portion, another on the graph layout algorithms, and the rest on OpenGL. The breakdown came down to Mai and John working on the Perl code, Richard on the algorithms, and Eric on the graphics. Richard and John provided additional coding support for Eric and his OpenGL needs.

After everyone knew what to work on, the next problem was getting requirements on what the front end was supposed to do, as well as what it was to look like. Documentation on netchar was minimal at best, even though parts of it were written with other tools. Coordinating a meeting with Allen proved another problem. The time difference, as well as the varying schedules of each group member, made it difficult to schedule a meeting time.

For the parser, we knew it had to provide output data that the graph layout engine could easily

work with. We easily spent more than a day discussing what the output should be, how many files, what kind of files, and the format of the data in the files. We decided to go with two files, a node data file and a link data file. We tossed around the idea for a path data file, but that was thrown out (more on this later). We aren't sure if the two-file approach would work, but it was sufficient for the graph layout engine.

For the graph layout engine, the major obstacle was in trying to find information on graphing algorithms. After doing extensive searches through the Internet, we came to the conclusion that there was a lot of information on graphing algorithms, but none that went into sufficient detail on how to go from description to implementation, since we were pressed for time and needed to come up with a layout quickly. Nonetheless, the algorithms that Richard came up with were more than acceptable for what we needed to provide.

The graphics driver proved to be a major challenge, especially since we really didn't know how we were actually going to display the data. It was also in the graphics driver that there was a lot of programming effort. Allen provided three data sets to us for testing the front end (MIT, UMASS, and ROCH). We mainly used the MIT data set for our testing since its graph was a relatively simple tree. When we reached a point in testing the remaining data sets, UMASS provided similar results to the MIT data set. It wasn't until we tried the ROCH data set that we had to do a lot of code changes to our design.

The ROCH data set caused us the most grief since it proved a lot of our initial assumptions wrong. We didn't anticipate such things as a node to appear in multiple paths, a node with a link to itself, two links between two nodes (i.e., a link from node a to node b, and a link from node b to node a, and with

different link characteristics), as well as others.

To solve the problem of more than one link between two nodes (as in the case of a bi-directional connection), we decided to represent the link with an arrow. We could have drawn a double-headed arrow and handle these cases in a special manner, but due to time constraints, we felt the situation could be handled just as effectively drawing two arrows and using alpha blending to allow both arrows to be seen.

The next problem with the graphics driver was in OpenGL's picking mechanism. When a user clicks on an object in the main display, if other objects happen to appear under (or above) the intended object, OpenGL will return all objects as being selected. This proved to be a problem because we didn't know which object the user really meant to be selected. The overlapping could be caused by overlapping arrows (as directed by the graph layout algorithm), poor placement of nodes, or due to the user rotating the perspective of the display. Because we have left the design open to allow 3D representations, we have made the selection process more difficult. Fortunately, OpenGL does support some mechanisms for object selection. Using these, when the user initiates multiple clicks in one spot where multiple objects exist, the user may toggle through all objects in that location. An additional feature allows users to drag an object to a new position.

Sometimes it is inconvenient to have to scroll the screen to see parts of the data. Sometimes it is better to see everything on the screen. The idea behind hyperbolic geometry is to "smoosh" or "squish" the edges of the screen and expand the detail in the center. The user then can "roll" the data around to examine what is desired. It was unclear if this project would need such a mechanism, but there was interest in having the possibility

of using it. Thus we implemented a simple hyperbolic routine based upon transforming (x, y) Cartesian coordinates to polar, transforming r to $r/(r+k)$, and then converting back to Cartesian to get an x' and y' .

Because the graph layout algorithms place nodes with concern to visual organization, distances between nodes lose meaning. Nodes placed far away from each other do not necessarily mean they have slow or distant connections. To compensate for this, we have implanted color scales on the links. Currently, we have intensity, rainbow, and heated metal objects. For intensity, the brighter the color, the higher the bandwidth. Rainbow goes from red to violet, with red as the higher bandwidth. Heated metal object simulates what metal does when it gets hot. It starts off black or gray, turns red, then yellow, and then bright white. White represents high bandwidth.

Because there are potentially multiple paths, we have implemented some features that allow the user to isolate specific paths. There is a path selector which in the case of multiple links with potentially different data sets, the user may select a path number, and the display will attempt to return information for that path. If there is no match, it defaults back to the first path.

There also may be a need to only want to visually see a specific path or ranges of paths. We have implemented a range slider that allows the user, for example, to display paths 3 through 6.

Animation was added to fill a deficiency in both Color Scales and Path Isolation. Color Scales show the global relationship between bandwidths for all links. However, if there are more subtle differences in bandwidths between links on a specific path, the user may want these emphasized. The global color scale may not have the precision to demonstrate these

differences clearly. As such, the animation function shows the local relationship between bandwidths on a requested path.

Overall, it was a mainly an issue of time, with everyone's busy schedule, that we weren't able to do a lot of what we wanted to do for the front end. We'll see in the next section some of the features we hope to see in future versions.

6 Future Work

As stated in the last section, time was the main factor that resulted in how the front end came to be.

For the parser, we would like to add the ability to name the output files. Currently, the parser generates the files "node.dat" and "link.dat". We think it would be better to have the ability to have flexible file names that the graph-layout engine could process.

We also wish more graph layout algorithms could be incorporated. This would provide different visual perspectives on representing the data.

Of course, a more seamless integration of the three major components of the front end, along with the rest of netchar needs to be addressed later. Currently, the parser first has to be applied to the output of the netchar back end, then the parser output files needs to be renamed to allow the graph layout engine to read the files, then the front end executable needs to be executed to finish off the visualization (i.e., reading the parser output files into our data structures, building the graph, and then calling OpenGL to display the graph).

For the graphics driver, there are several items we wish to see addressed in later versions.

The selection of spheres may not have been the best choice for the nodes because transforming a sphere in the manner mentioned in the previous section is difficult.

Our result was a fake approximation of transforming the spheres hoping that the transformation of the arrows and "rolling" sensation will be enough to keep the representation feeling natural.

Arrows were implemented using a ribbon-tracing algorithm. We propose changing the nodes to a different geometric representation.

Finally, we did not implement animation for hyperbolic mode. Future work may include this feature, as well as changing the animation scale to reflect global bandwidth relationships. Local bandwidth relationships could also be added to the color scales scheme.

7 Conclusion

In this paper we have provided some historical background information on current tools that try to estimate link characteristics, explained the overall structure of netchar, described our work on the front end, and offered possible future work. We believe our implementation for the front end represents a first step in creating a user-friendly, visually-appealing, link-characteristic estimation tool.

Acknowledgements

We would like to thank Allen Downey and Bruce Mah for giving us the opportunity to work with them on Netchar. We would also like to thank Geoff Voelker for introducing us to a fun (yet intensely challenging) and educational project. Finally, we would also like to thank K Claffy and the rest of the CAIDA group at SDSC for their inspiration to create useful (and much needed) networking tools.

References

[1] Downey, Allen B. Using pathchar to estimate Internet link characteristics. In Proceedings of

SIGCOMM '99 (Colby College,
Waterville, ME, 1999).

[2] Downey, Allen B. `clink`: a tool
for estimating Internet link
characteristics.
Presented at
<http://rocky.wellesley.edu/downey/clink/>.

[3] Downey, Allen B. `netchar`: a
tool for estimating Internet link
characteristics. Presented at
<http://rocky.wellesley.edu/downey/netchar/>.

[4] Jacobson, Van. `pathchar` - a
tool to infer characteristics of
Internet paths. Presented at the
Mathematical Sciences Research
Institute (MSRI), April 1997.

[5] Mah, Bruce A. Estimating
Bandwidth and Other Network
Properties. Presented at the
Internet Statistics and Metrics
Analysis Workshop on Routing and
Topology Data Sets: Correlation and
Visualization, San Diego, CA,
December 7-8, 2000.

[6] Mah, Bruce A. `Pchar`: Child of
`Pathchar`. Presented at the DOE NGI
Testbed Workshop, Berkeley, CA, July
21, 1999.