Network Performance Visualization: Insight Through Animation.

Brown J.A., McGregor A.J., Braun H-W

Abstract— **In passive and active measurement projects it is easy to generate large volumes of raw data. Although this allows for detailed analysis, the large volume is an impediment to understanding the data and locating interesting events. One approach to this problem is to use visualization tools to produce a graphical rendering of the data that allows a user to explore the data "by eye."**

Cichlid is a visualization tool that provides high-quality 3-D, animated visualizations of a wide range of network analysis related data sets. Cichlid allows the viewer to explore and interact with the data sets in real time. It was designed with remote data generation and machine independence in mind; data is transmitted via TCP from any number of sources (data servers) to the visualization engine (the client), which displays them concurrently.

Cichlid features real-time data display, point-and-click user feedback, and dynamic data coloring and labeling. Sequences of frames may be captured for later encoding in a standard movie format and single frames may be rendered at arbitrary resolutions.

Using Cichlid has allowed us to gain new insights into the data we collect and will, we feel sure, aid others in the networking community, particularly network providers, to more easily gain an overview of the data that they collect.

Keywords— **Network performance, visualization, OpenGL.**

I. Introduction

If a picture is worth a thousand words, then an animated visualization is worth a thousand static graphs.

The National Laboratory for Applied Network Research (NLANR)[1] has a number of network measurement projects that produce large volumes of data. These include the OCxMON Passive Monitoring and Analysis project (PMA), the Active Monitoring Project (AMP), and the collection and analysis of SNMP and BGP data.[2][3] All together, these projects produce gigabytes of data and thousands of Web pages of graphs and summaries each day. A large volume of data is necessary because it is not known apriori which parts of the data will contain interesting features and because different users are interested in different parts of the measured systems. As a consequence of the scale of the systems, it is not humanly possible for users to scan all of the data or web pages for interesting artifacts, but in not doing so, important network events may be overlooked. As one approach to address this need, NLANR has developed the Cichlid[4] tool for visualizing and animating data sets.

Cichlid allows its users to view data sets in three dimensions, as if they were physical objects. The data can be animated (to show changes over time) and the point of view can be changed (zooming and moving around the data). Often, a physical analogy of the data can be developed; for example, network delays between a collection of sites, visualized as an interpolated surface and animated in time, gives a display similar to the view from flying over mountainous terrain (see figure 1). Different landscapes are related to different network conditions. We have found that using Cichlid for network data visualization, provides new insights into the data we collect.

Cichlid is written in C, using the OpenGL[5] and GLUT[6] graphics libraries. The code is portable and is currently being used on the FreeBSD, Linux, Microsoft Windows, and IRIX platforms. The source code for Cichlid is freely available, as are the sources for GLUT, a window system interface for OpenGL programs, and for Mesa[7], an OpenGL-like graphics library. Thus, the tool can be built for free, and used on standard PC hardware, as well as high-end SGI workstations.

J.A. Brown and H-W Braun are with the National Laboratory of Applied Network Research (NLANR), San Diego Supercomputer Center, University of California, San Diego, La Jolla, USA Email:{jabrown,hwb}@nlanr.net

A.J. McGregor is with The University of Waikato, Hamilton, New Zealand. Email: tonym@cs.waikato.ac.nz

This work is funded, in part, by NSF Cooperative Agreement No. ANI-9807479. The U.S. government has certain rights in this material.

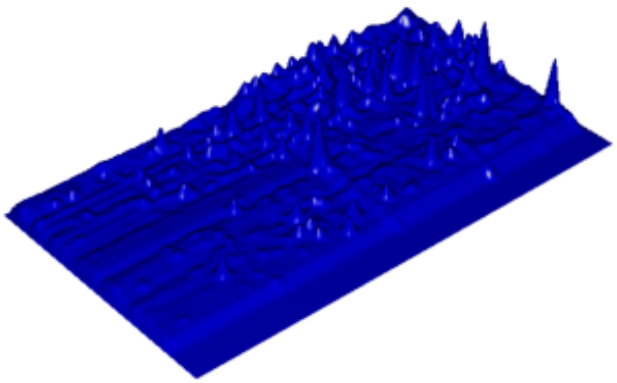


Fig. 1. Network "Terrain"

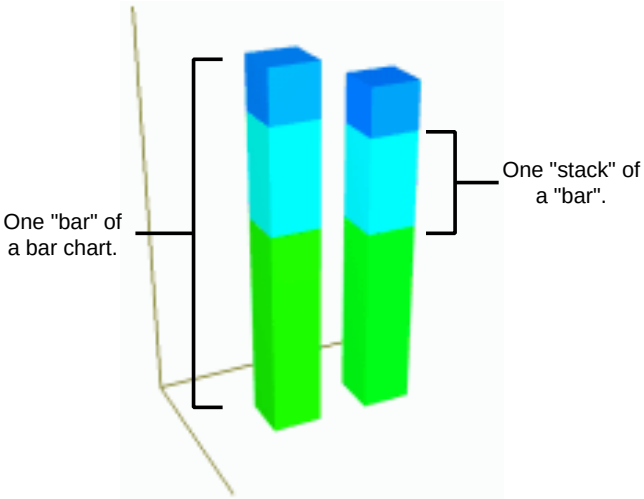


Fig. 2. Bar Chart Components

Cichlid is implemented as a client-server system; one or more data servers provide raw data through TCP connections to a client which renders the data graphically. Splitting the system this way allows the data server(s) to be on one or more remote machines, possibly the systems where the data are collected or stored. The client, which performs the rendering, is run on the user's workstation. Care has been taken in the design of the protocols and support software to minimize the data transfer required between the clients and the servers.

II. Functionality

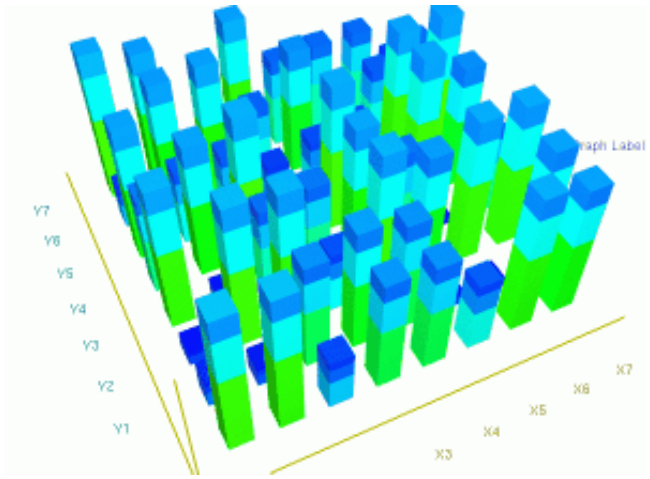
A. Types of Visualization

Cichlid currently supports two types of graphs: 3-D bar charts, which are useful for displaying numeric quantities that are functions of two independent variables, and vertex/edge graphs, which are good for representing topology. Cichlid bar charts have been used to show data sets that include matrices of network delays between pairs of sites, traffic distribution over address blocks, as well as traffic volume by protocol and source. The vertex/edge mode has been used to visualize data sets that include latencies from a single site to others, as well as those showing network evolution over time. In most cases, these data sets are animated in time by repeatedly supplying data from successive measurements at real-time intervals.

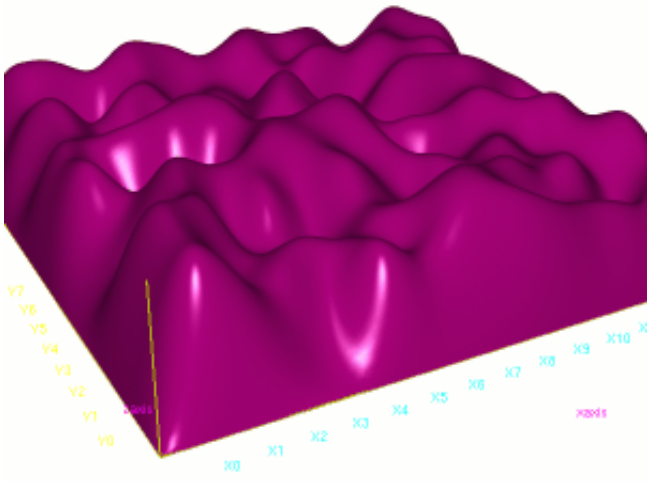
The two types of graphs that Cichlid supports are described below:

- Bar Charts

A bar chart object (see figures 2 and 3) consists of a rectangular array of "bars." Each bar con-



(a) Stacked Bar Chart



(b) Interpolated Surface

Fig. 3. Examples of Cichlid Bar Charts

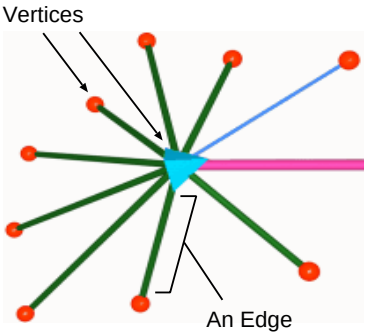


Fig. 4. Vertex-Edge Graph Components

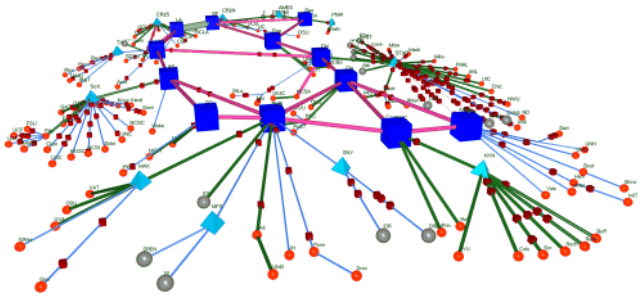


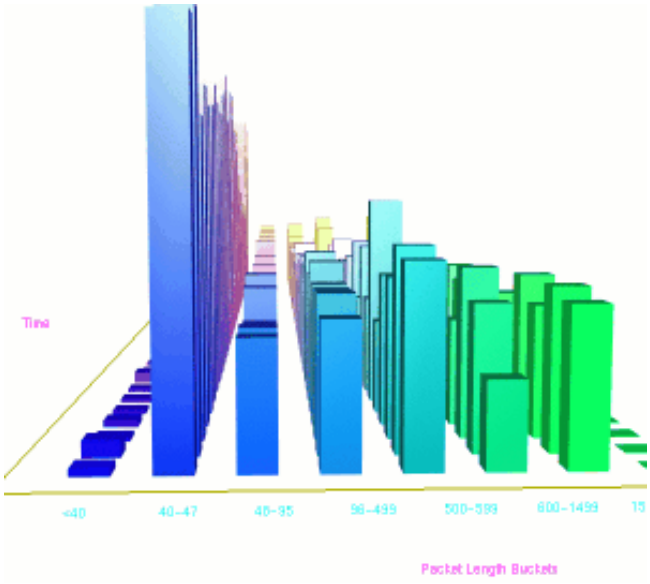
Fig. 5. Vertex-Edge Graph Example

sists of one or more "stacks," and each stack contains height and color attributes. The stacks are placed end-to-end to form the bars, and the bars are placed on the base plane at evenly spaced intervals to form the bar chart.

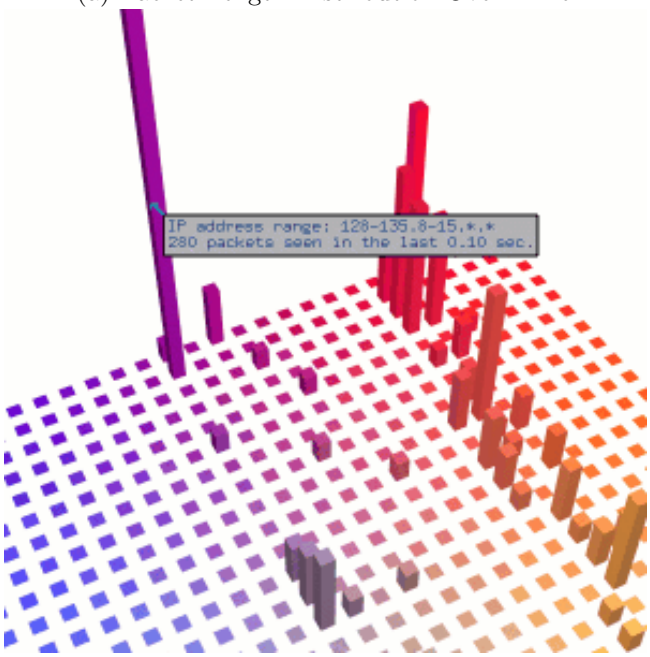
- Vertex-Edge Graphs

A vertex-edge graph (see figures 4 and 5) consists of two arrays: one of vertex structures, and one of edge structures. This corresponds closely to the abstract mathematical representation of a graph as $G(V, E)$, except that the vertices and edges in Cichlid contain not only connectivity information, but also graphical attributes. Each vertex is defined by a `VtxInfo` structure, which contains a 3-D location vector, a relative size parameter, color information, and a drawing style hint. Each edge is defined by an `EdgeInfo` structure, which indicates the vertices where the edge terminates, the directionality of the edge, size, color, and style. The vertices are defined to exist in a 3-D coordinate space, which is declared ahead of time; this space is mapped to the final graphical representation. The edges are defined to connect pairs of vertices. This information does not in itself define a graphical representation; it describes the graph, and provides hints as to how it should be drawn. This is a very flexible model in that arbitrary graphs can be displayed; the only condition is that they must be laid out in Euclidean space.

Figure 6 shows examples of Cichlid visualizations of real data. Figure 6(a) is a plot of bucketed packet lengths over time. Each bucket along the X-axis represents a range of packet sizes. Older data samples are spread along the Y-axis. The height of each bar indicates the number of packets that fall in a particular bucket during a sample interval. Figure 6(b) is an address space visualiza-



(a) Packet Length Distribution Over Time



(b) Traffic Volume By Address Block

Fig. 6.

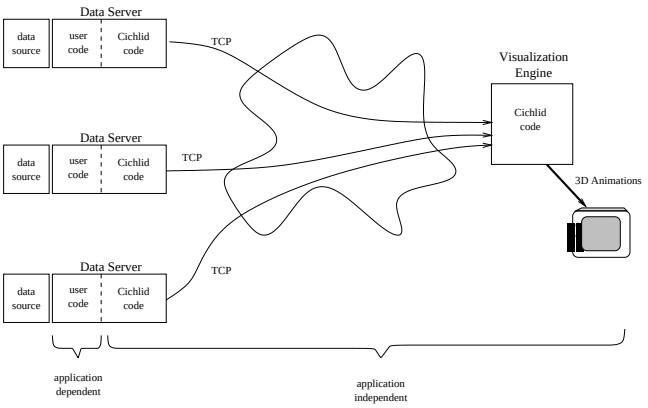


Fig. 7. Cichlid Components

tion. Each bar corresponds to a portion of the IP address space; the height represents the number of packets sent during the previous sample.

Regardless of the types of graphs being displayed, the user's point of view can be changed independently of the data objects, allowing the data sets to be explored from different angles. Rendering parameters can be adjusted by the user, allowing them to view the objects as solids, wire-frame models, and in the case of bar charts, interpolated surfaces.

B. Distribution of Function

A visualization built using Cichlid is composed of two sets of components, shown in figure 7. The system contains application-specific code (which produces data), and an application-independent visualization engine (which consumes data and renders the visualization). Data is transported between these two components using TCP connections, with the data producers acting as servers and the data consumers as clients.

While the data normally flows from the server to the client, some data moves in the opposite direction. A user is able to query the server by clicking on a component of the visualization. The query is sent to the data server which replies with a text string. The visualization engine, then displays the string as part of the visualization. This can be used, for example, to support pop-up labels indicating the source of a measurement (see figure 6[b]).

Cichlid provides an entire visualization infrastructure, including the rendering and data transfer functionalities. The Cichlid user who wishes to build a new visualization need only write code in a Cichlid server which manipulates that data to conform to one of Cichlid's data models. The Cichlid system then handles the data transport, output visualization, and user interaction. Since the infrastructure itself is data and application-independent, creating a new visualization is reduced to merely having to write code to import application-specific data and hand it to Cichlid through a function-call interface. This allows new visualizations to be developed rapidly and with a minimum of effort.

III. Design

The Cichlid visualization system performs three primary functions:
1. *Abstraction and Modeling* - representing real-world data in an abstract and application-independent manner.
2. *Collection and Distribution* - accepting data at the sources, and delivering it to the visualization engine.
3. *Visualization* - rapidly rendering data into attractive visualizations.

A. Abstraction and Modeling

A powerful feature of Cichlid is that its internal data representations are not tied to any particular application; they are abstract models. Generally, when user data enters the system, it goes through user code which manipulates the data to fit one of the models Cichlid provides. We call these abstract models Cichlid "data sets." The `DataSet` object family is responsible for representing and operating on data sets efficiently, and for providing architecture-independent external representations on demand. It is worth noting that these data models are not inherently tied to any particular graphical representation or display API.

Cichlid provides several different data models. Each is designed to be powerful enough to represent useful data, simple enough to be convenient, and general enough so that each new application that comes along does not require specific extensions to the models. As much as possible, the graphical details of the visualization are left out of the parameter set. There are no advanced visual parameters such as reflection, collision behavior, transparency, or fogging specified in the data

models. Supporting such parameters would add to the complexity of the underlying system and burden the server writer by requiring more information to be produced.

Some of the parameters in the data models are fixed at the time that the `DataSet` objects are created — for example, the coordinate spaces over which vertex-edge graphs will be defined — but most parameters can be changed at any time by the data server. This flexibility allows for modeling (and visualizing) continuously changing data sets.

The `DataSet` object family represents all of the abstract data models supported by Cichlid. It includes methods to populate and update the data models, and codecs to convert between the internal object representations and compact, machine-independent external representations used to transport the objects to remote processes.

The encoded data are typically transmitted over wide-area networks, so the encoding schemes are designed with an emphasis on saving space. To facilitate this, the codecs can operate in a differential mode (encoding state changes in the objects), if this encoding takes up less space than the object itself.

Although minimizing the volume of data transferred is of primary importance, speed is also a concern since CPU time is often at a premium at the decoder, which is also the visualization client. In particular, a tradeoff between speed and accuracy is made for floating-point values. Since the spatial quantities involved in visualizations need not vary over many orders of magnitude, and since a small loss of accuracy does not affect the visualizations noticeably, floating point values are quantized to fixed-width integers for transmission.

The `DataSet` object family contains several sub-objects, one for each data model supported. The `DataSet` object interface provides methods that are common to all sub-objects, such as destruction and encoding, and it exposes methods that are specific to the sub-object types.

While methods to operate on the sub-objects are provided, the means to explicitly reference them are not; all exposed methods take `DataSet` pointers. This adds the need for some run-time pointer table lookups, but simplifies the programming interface somewhat. All operations on the `DataSet` objects must be performed through method calls; the user is not free to assign directly to any members of a `DataSet`. This strictly procedural interface allows the `DataSet` objects to cheaply track state changes and perform differential encoding, without having to do large set comparisons.

B. Data Collection and Distribution

One of the key features of Cichlid is that it is a distributed system. There are data servers, where user data enters the system, and data clients, which consume the data and generate visualizations. User code at the server converts the data into one of Cichlid's models as a `DataSet` object, and built-in server code maintains the `DataSet` object as well as connections to clients. The communications and protocol modules which the client and server share are responsible for propagating state changes from the server to the client in an efficient and architecture-independent manner.

There are two phases of data collection that take place in a Cichlid visualization. The first is when user code collects application-specific data, analyzes them, and populates `DataSet` objects; the second is when the Cichlid client collects the modeled data from Cichlid servers. The data distribution takes place between the two: the `DataSet` is encoded and distributed from the Cichlid servers to waiting Cichlid clients.

The data collection code that abstracts the application-specific data into `DataSet` models is the responsibility of the server writer, since the toolkit itself has no knowledge of the application domain. The analysis code populates the `DataSet` objects through calls to the accessor methods that are exposed for manipulating the models. In practice, the user server code that does this is invoked from the Cichlid server library through one of the function callbacks available — for example, it can be configured to be invoked upon every client request, or each time the server finds itself with no immediate client requests pending. As the user code manipulates the `DataSet` over time, the `DataSet` model becomes a dynamic, abstract representation of the data to be visualized.

The Cichlid server library handles the overall control flow of the server. As clients can come

and go, it is the responsibility of the server library to ensure that each client gets an accurate representation of the `DataSet` model's state at the time it receives a request for it. Not only is it important to transmit the state information correctly, it's also important to transmit it efficiently. The server makes use of the codec methods provided with the `DataSet` objects to assemble the raw data to be transmitted to a given client; it uses the protocol and communications modules common to the clients and servers to frame the data for transmission, and to perform all socket operations needed to transport the data.

On the client side, the data distribution completes with the client receiving and decoding the data, essentially reversing the steps that took place in the servers. The client's copy of the `DataSet` object is thus set up as a mirror of the one on the server. The client performs these operations for each server to which it is currently connected, collecting the data streams and keeping all of the `DataSet` objects as up-to-date as the performance of the network and the client allows.

To aid clients and servers in communicating with each other, and to prevent them from attempting to communicate with incompatible clients and servers (be they incompatible versions of Cichlid, or the latest sendmail replacement), a rudimentary greeting-handshake protocol is implemented. This protocol uses a plain-text description of the fixed `DataSet` parameters as well as the program and protocol version numbers. Using a text format has several advantages including making it possible to identify orphaned servers by simply using telnet to connect to them.

A simple framing protocol is also implemented, in which all data sent between clients and servers after the handshake is sent in complete "frames," which can vary in size from frame-to-frame, but must have a predetermined size which cannot be changed during transmission — much like a datagram. The protocol includes distinct headers to help detect protocol errors, and provides multiplexing based upon a "frame type" parameter. The protocol was designed so that many streams could be processed in a nonblocking fashion without the need for threading or additional processes. This framing protocol runs on top of TCP, so is not necessary to provide redundancy checking.

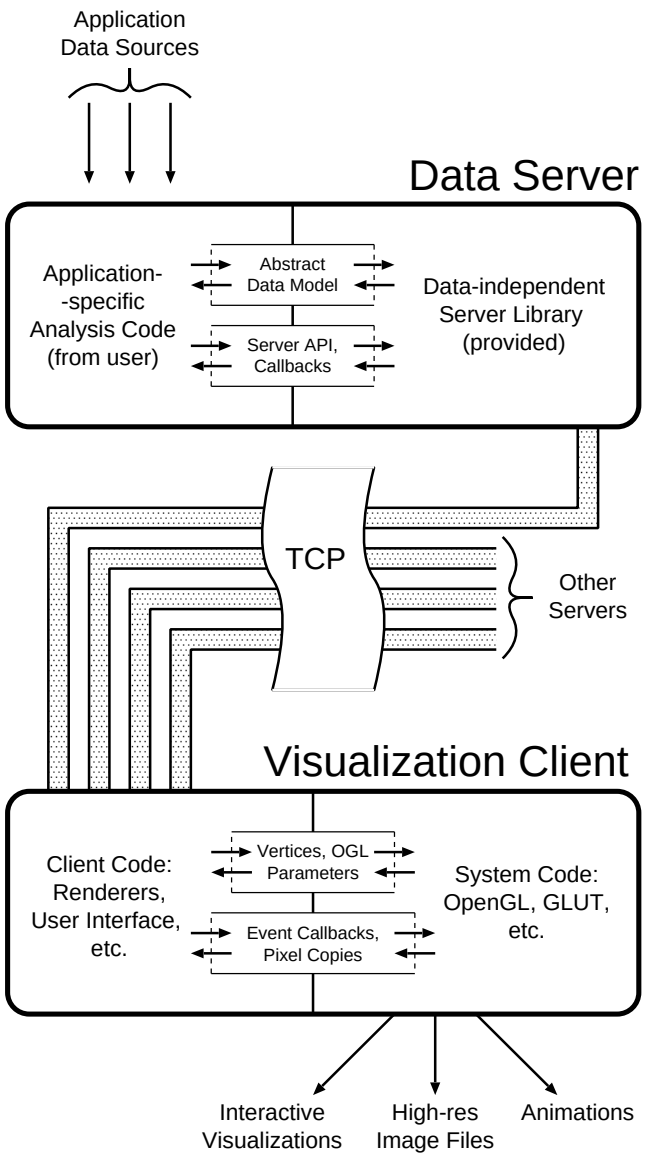


Fig. 8. Cichlid Data Flow

Figure 8 illustrates the high-level layout and data flow of a Cichlid visualization.

C. Data Visualization

The Cichlid client, the most complex part of the system, is responsible for all of the visualization functions. It has a set of renderer modules which produce graphical representations of the abstract data sets, as well as code to maintain the on-screen visualization and to provide an interface to enable users to interact with the displayed data sets.

The data provided by the user analysis code is ultimately shipped to one or more clients for visualization. The operations performed by each client are more complex than those performed by the

servers; but since the client code operates solely on abstract `DataSet` models, it requires no modification to be used with new visualizations. The visualization developer is thus insulated from the details of the graphics system, particularly from the numerous calculations that must be performed, and from the OpenGL API, which has a seemingly endless number of state variables to worry about.

The visualization client is responsible for maintaining the display of all data from each of the connected servers, for requesting new data from the servers as it has the processing capacity to handle more, and for interfacing with the visualization user to allow them to interact with the displayed graphs — changing viewpoints and rendering settings, requesting the server to perform application-dependent operations on the data, and even allowing the user to request application-dependent information about specific elements of the displayed `DataSet` directly from the "user" server code.

The primary source of complexity in the client is in the code which performs the OpenGL renderings of the `DataSet` objects. Each `DataSet` sub-object in the shared "data set" library has a corresponding "renderer" in the client that is responsible for generating all the OpenGL calls to visualize it. The renderers bypass the `DataSet` object's procedural interface for speed. The renderers themselves store only ancillary graphical information and drawing option settings. When they perform the rendering, they read the data directly from the `DataSet` objects. This violates the object-abstraction model, but this is acceptable due to the speed improvement achieved because the renderer does not have to waste time copying large amounts of information out of `DataSet`s, and it does not have to make method calls to get at the data. While the function call overhead would not ordinarily be a problem — it's fine at the server end — the renderer code may be called very frequently, and each call can involve iterating over the `DataSet` several times. Given this need for speed, and that the renderers do not modify the `DataSet` objects, the function call interface only gets in the way. We trade cleanly decoupled renderers and `DataSet` objects for speed.

The renderers corresponding to the various `DataSet` sub-objects contain full knowledge of the internals of that type of `DataSet`; they also have several different strategies for rendering that `DataSet`. The various strategies offer trade-offs between rendering in a manner that more closely matches the style hints provided in the `DataSet`, and rendering in a manner that is as fast as possible.

IV. Strengths and Weaknesses

The strengths of the Cichlid system derive from the fact that it is not designed for a specific visualization. Requiring that the data be abstracted before it enters the Cichlid system ensures data independence. It does preclude detailed analysis in the visualization client, and it complicates user interaction with the data, since any requests for information from, or detailed manipulation of, the application-specific data must be handled remotely in the data server.

Portability is one of the principles underscoring the entire design. The vast majority of the system is written in ANSI C, with no dependencies on a particular platform or on nonstandard extensions to the C library. While this results in code that is portable to different operating systems, Cichlid needs to perform graphical output, interact with the user, do high-resolution timing, and perform socket I/O, the specifics of which are system-dependent. The use of the OpenGL and GLUT APIs allow for platform-independent graphics operations; but several extensions are supported on specific platforms, such as the use of Crystal Eyes stereo glasses on SGI workstations. In order to confine the scope of the system-specific code, it is compartmentalized in special-purpose modules, the most prominent of which are the socket I/O module and the general-purpose system-dependent module which handles simpler things like timing, random number generation, size-specific data type definitions, and header file management. The primary task in porting Cichlid to new platforms is the adaptation of those two modules to the target OS. It currently is adapted to the BSD-style UNIX, IRIX, Linux, and win32 programming environments. While the servers make use of these modules, they have no inherent need for a windowing system or graphics libraries, and may be built on systems without those accou-

terments.

The design of Cichlid is centered heavily around the use of opaque objects. The design does not expose data structures of an object to other modules; while this is sometimes awkward in C, since it necessitates the use of many accessor functions, the object-oriented design simplifies the maintenance of the code and the addition of new features. Conversion to C++ would simplify the design a bit, and as the C++ implementation in the free "gcc" compiler suite has come of age, it is a definite possibility for the future.

The Cichlid client currently lacks an intuitive user interface. The current user interface is primarily textual, with a myriad of commands being accessed through single-key inputs in the graphic window, while the command output comes out on the system's "standard output" device. This is awkward at first, since there are too many commands to intuitively map to single keystrokes, and it requires switching between the text and standard output windows to see status messages. After a bit of experience, however, the direct, terse style of command input is convenient and fast to use, and the graphical display is not cluttered up with the chit-chat of status messages. The mouse is used for camera movement and point-and-click selection. Work is underway to add pop-up menus and a bona fide status window so that the system is less daunting to new users.

As previously mentioned, all of the data transport in Cichlid is implemented on top of TCP. While having a reliable connection simplifies the system design, the system is not able to adapt the data stream effectively to account for latency or packet loss, as is possible with applications that use UDP. Not much can be done about this — the data-independence restriction prohibits the Cichlid system itself from guessing or interpolating around losses, as one could acceptably do with something like a video stream. Additionally, the client/server protocol operates in a stop-and-wait manner, without the ability to process concurrent requests. This simplified the initial implementation, but makes the visualization very sensitive to network latency, particularly at high frame rates. What the protocol needs is some sort of pipelining ability; this can be added in the future with minimal, if any, modification to the server writers' interface, because they interact strictly with data objects, irrespective of the "wire" protocol.

While the Cichlid system allows one client to be connected to many servers at once, currently, each server can only handle a single client connection at a time. This will be improved in the future, with minimal changes to the server API.

V. What's Next

There are many opportunities for making improvements to Cichlid: particularly the addition of a more intuitive user interface, and convenience features such as the ability to save and restore the state of a visualization across invocations of the client. We also would like to add new data models to the toolkit, which would allow for new types of visualizations, and to continue to improve the current models in response to requests from our users. In addition, we'd like to add some graphical niceties to the design, such as support for simple bit-mapped textures, and the ability to save output in a vector-graphic format (such as PostScript) that is more amenable to printing than bitmaps.

Some portions of the internal architecture, such as the client-server protocol, are implemented in naive ways that reduce the scalability and throughput of the tool. While these internal mechanics could use some improvements, changes of this type are not necessary for the basic functionality of the tool, and they can be implemented in an incremental fashion without intervention by the visualization designers.

All processes in Cichlid are single-threaded. This eases portability, particularly to systems without POSIX-threads, but creates some complicated nonblocking service loops. Portions of the system, particularly the client, may be converted to use threads in the future.

The modular design of Cichlid does not preclude the addition of these or other improvements. It does, however, make it difficult to introduce application-specific behavior on the client side; this is actually a feature of the design, since we have strived for generality and reusability.

In conclusion, Cichlid has been used in a number of network analysis projects, including those at NLANR and other organizations. The Cichlid visualization system has proved helpful in un-

derstanding network behavior and in highlighting anomalies. It has been particularly useful in demonstrations, as shown at recent Supercomputing conferences, where it enabled attendees to quickly understand the data being presented. The Cichlid architecture, which minimizes the work required to create new visualizations, allowed these demonstrations to be developed in just a few weeks. Building on this success, the improvements we have planned will further enhance the usefulness of Cichlid as an analysis tool.

References

[1] http://moat.nlanr.net/.
[2] A.J. McGregor, H-W. Braun, and J.A. Brown, "The NLANR network analysis infrastructure," *IEEE Communicarions Magazine*, May 2000, to be published.
[3] A.J. McGregor and H-W. Braun, "Balancing cost and utility in active monitoring: The AMP example.," *INET 2000*, June 2000, *submitted, also at http://byerley.cs.waikato.ac.nz/ tonym/papers/inet2000.*
[4] http://moat.nlanr.net/Software/Cichlid/.
[5] http://www.opengl.org/.
[6] http://reality.sgi.com/opengl/glut3/glut3.html.
[7] http://www.mesa3d.org/.