

INTERACTIVE NETWORK ACTIVE-TRAFFIC VISUALIZATION

Robinson, N., Scaparra, J.

Computer Science

Texas A&M University

College Station, TX 77943, USA

Abstract – The Interactive Network Active-traffic Visualization (INAV) was designed as a passive monitoring solution for use in real-time network environments. This paper will focus on the improvements of network traffic analysis and unique considerations under investigation that can exist within a network topology. INAV can effectively map a network under extreme conditions – upwards of ten-thousand nodes per second. This is of interest to network administrators, as an accurate traffic map of the network can be created very quickly, where relevant information is collected for sense-making tasks using non-intrusive methods. The underlying technologies used by INAV can support other types of network infrastructures, and significantly changes the current definition of passive scanning. The INAV system supports a variety of platforms via a protocol written in both C++ and Java.

1. INTRODUCTION

Interactive network active-traffic visualization (INAV) is a monitoring solution for use in real-time network environments, and is unique such that it only monitors the traffic that is currently active between nodes within a network and affords an intuitive visualization of that information - which consists of multiple layers of information/interaction. Several considerations exist that must be considered for this choice of research: to investigate and explore the problems and solutions associated with the visualization of large graphs and amounts of data. Also to visualize the collected information without the problem of overwhelming the user, and particularly important is how to visualize the network infrastructure in real-time. Additionally, INAV is a useful tool designed for use by network administrators for the discovery of network traffic trends and behavior. This is relevant as the network administrator is responsible for the reliability and traffic between network segments, which is collected by utilizing passive scanning techniques.

There will be two sections of INAV, the server and the client. The server is responsible for collecting traffic statistics and sending them to the client module (visualization) which will consist of a graph in which the nodes are associated with IP addresses, and the edges represent the traffic between those two nodes. As traffic is collected, new nodes will emerge, and their associated traffic (the edges) change in hue/saturation, as to represent the throughput of traffic between those nodes

(figure 1 & 2) and is explained by the legend (figure 3). This is also interactive, so that feedback is provided to the user throughout the different layers of information: computer/router, traffic data types, and the active network infrastructure.

By providing this information in real-time and on-demand the network administrator will be able to evaluate health diagnostics of the network and discover network traffic/routing trends. This is important as the efficiency and robustness of a network depends on the proper configuration of equipment in order to suit current traffic needs. The important processes in our research are the dynamic, interactive visualization coupled with real-time active statistics of network traffic. This bridges an already existing gap between "snapshot" network dataflow graphs and real-time packet sniffing applications.

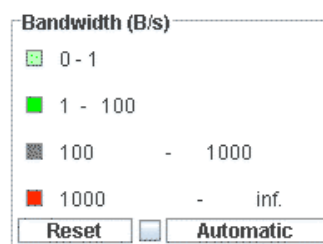
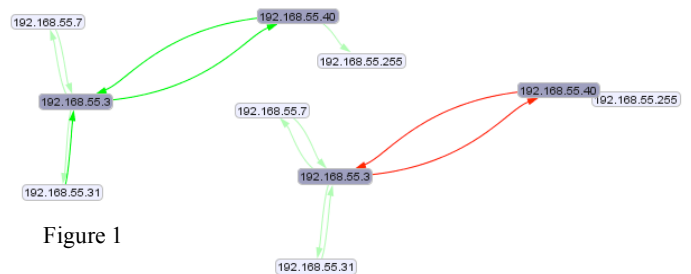


Figure 3

Fig. 1-3. Examples of an edge changing based upon traffic between the connecting nodes. The color green indicates traffic in the 1-100 Bytes per second range, and the color red indicates a traffic rate greater than 1,000 Bytes per second.

2. CONTRIBUTIONS AND BENEFITS

There is currently plenty of development in many different network mapping applications and there also exists network diagnostic software, however a key difference is that there is no concurrent link between the visualization of passive network traffic in a real-time environment and time-traffic analysis, both examples of where INAV provides a solution. This is achieved by utilizing multiple layers alongside details on demand. For network administrators, it is essential to know network health and efficiency, and INAV provides this information through the use of an interactive graph – and at the highest level, the user will see the throughput thresholds between nodes (figure 1,2) and be able to monitor their efficiency. The INAV project meets these needs by providing feedback and interaction through a web accessible interface.

Another prospective use for INAV is for the small office/home office (SOHO) user. One key issues they would find valuable is to assist them troubleshooting network connections. One such scenario would be if your ISP goes down you will notice that link deactivated. Following this SOHO usefulness – INAV will allow for the discovery of mal-ware or some unknown virus on your system, as the traffic information of the mal-ware will be visible the second they communicate with servers across the internet. The feedback mechanism will be intuitive and based upon current conceptual models and ideas so that the information presented can be easily understood.

At the data-link layer – in which the INAV server analyzes packet information (figure 4) – there will be a server with two network interfaces. The first interface will be for data collection and is where the traffic will be passively sniffed. The second is for communicating with the client (visualization module) which will display the data, issue configuration commands, and request additional node data. These actions are non-trivial, as the display and processing of thousands (and potentially millions) of simultaneous nodes and edges is visually and computationally difficult. One problem that was quickly discovered between the client and server was that performing trace routes for large graph sets placed an undue, large strain on the network – causing latency timeouts to occur. To overcome this problem, INAV only performs this on inspections of a node, and places a cap on the trace routes at some restrictive bandwidth limit and caches the results into a lookup table. The only active network detection method used is an ICMP trace route.

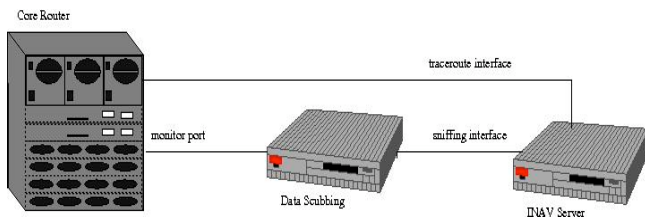


Figure 4

3. PRIOR WORK

Many other applications have done some part of collecting or visualizing of network traffic. Open source projects such as cheops-ng [1] map networks and can dynamically find devices on the network (figure 5) with little or no user input. Other papers and applications exist for visualizing network traffic and dealing with the large amounts of data that is associated with computer data networks [4]. Our research combines these efforts and incorporates real-time information about the network to give an instantaneous snapshot of the current state of the network.

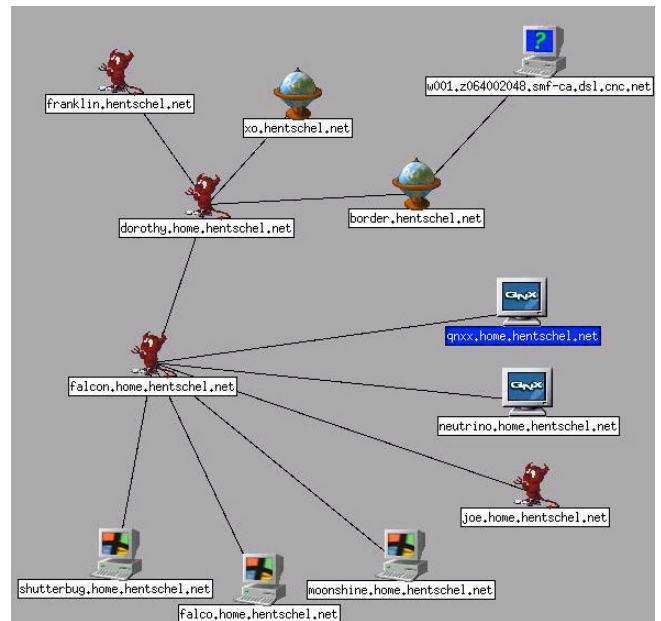


Figure 5

One such network management tool for mapping and monitoring your network is Cheops-ng [1]. Cheops-ng is split into a server and client architecture much like our design with the server running on a separate computer on the network. The client interacts with the server to actively discover entities on the network. Cheops-ng can also be used to communicate with devices after their discovery. Interaction is afforded by the use of right clicking on an entity which will drop down a menu allowing connections to services that were discovered via a port scan of the system. It is also possible to rerun the scans and reverse lookups of the hostname. Another useful feature of Cheops-ng is operating system detection. This allows system administrators to see the different types of systems that are on the network.

Another study performed by five national researching institutes, Lawrence Berkeley National Laboratory, High Performance Computing Research Department, National Energy Research Sciences Center, Los Alamos National Laboratory, and the University of New Mexico worked toward identifying, characterizing, and visualizing "anomalous subsets of as large of a collection of network connection data as possible" [2]. They analyzed petabytes

worth of network traffic over the course of 24 months. In order to manage the data Lawrence Berkeley "developed a unique indexing, storage and retrieval system known as *FastBit* that uses extremely efficient compressed bitmap indices." [2] In addition to the very advanced database machine-learning is used for automatic classification and novelty detection. Los Alamos's *Emaad* system can detect anomalies unsupervised. With these capabilities they were able to perform Intrusion Detection to identify failed connections and graph them. For visualization of the data two different methods were used, the SpaceShield and HyperSpace. The SpaceShield Viewer (figure 6) displays hosts on a globe representing the world and in the middle sits the internal hosts for the network. In contrast the HyperCude Viewer (figure 7) shows cubes which represent similar connections. "Expert analysts are experimenting with these clusters to try to identify an intuitive understanding of how these data are correlated"[2].

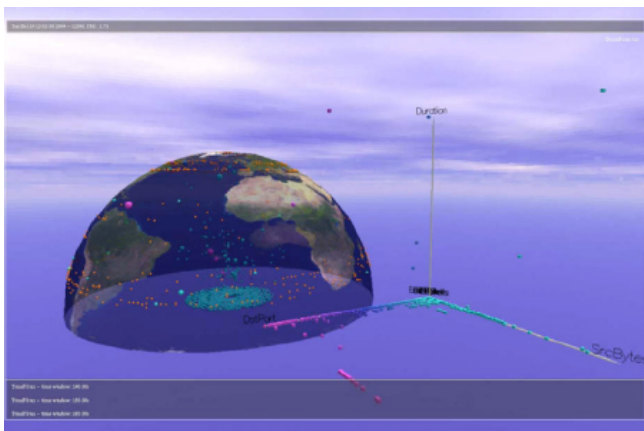


Figure 6

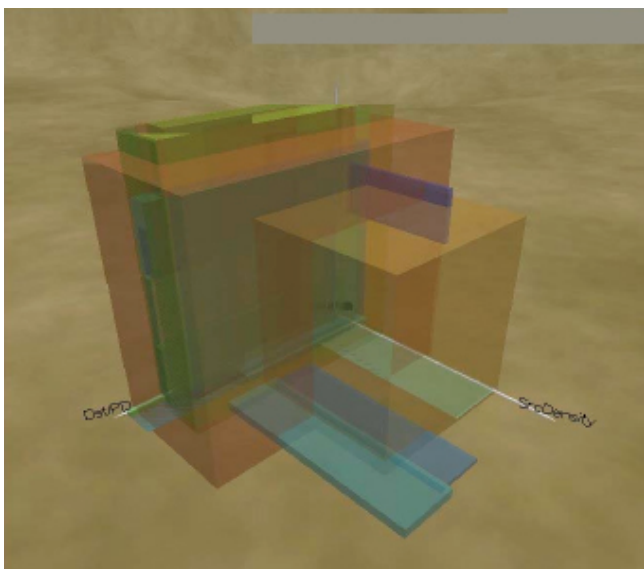


Figure 7

In a similar paper entitled *Niche Works-Interactive Visualization of Very Large Graphs* focuses on managing the large amounts of data that networks provide and displaying

them visually [3]. NicheWorks, a tool created by the Bell Labs Visualization Group is part of a suite of visualization views for performing interactive analysis of large datasets. The visualization takes place in two parts. In part one, the initial layout; the algorithm must be capable of laying out up to a million nodes on a graph in just a few minutes. Then in the second part, the improvement phase, the user can use more advance incremental algorithms to shape the graph more accordingly. Studies have shown humans perceive things as instantaneous if they occur "sufficiently fast" (within 200ms). Because there is no way to re-draw a million node graph in that time partial-drawing solutions had to be developed. By predicting the number of nodes that can be drawn in 200ms they are able to keep the graph's shape very similar to the full graph with all the nodes. "NicheWorks allows users to visualize weighted networks with hundreds of thousands of nodes and edges."[3]

4. THE SYSTEM

The INAV research project is split into two distinct parts, the INAV server (INAVD) which gathers information and sends it to the INAV client (the visualization). The communication between the two occurs over a TCP/IP network which will allow them to operate in different locations or on the same physical machine. INAVD is developed using C++ and the client is developed using Java. This configuration allows for the server to run on a lightweight Linux system, with root privileges (which are needed to allow INAVD to put the interface into promiscuous mode) and allows the client to run on multiple types of architectures in a web environment.

In order for the INAVD to see the traffic it either needs to be configured as a router that the traffic is going through or have an interface that a switch is mirroring traffic to. INAVD will then collect information and discover the routes traffic is taking between nodes. It will temporarily store this state information and then relay it to the INAV client. It will also be responsible for bandwidth statistic reporting which subsequently let the INAV client show link congestions. The INAV client will use a sliding window for which the graph data is stored, which is necessary as of the amount of traffic that networks create is considerable. The client will serve as a lightweight GUI responsible for relaying user input to the server and displaying server and node data.

4.1 INAVD – the back end

The INAV server (INAVD) is the workhorse of the entire system, since the client is a lightweight GUI component the data collection and processing of tens of thousands of nodes must occur on the server. INAVD is a multithreaded application written for Linux using the pthreads [5] and pcap [6] libraries. The pcap library is used to sniff data from the network passively. There are many other network analysis tools that use the pcap library to collect their data, wireshark [8], tcpdump [9]. Another web based project, ntop [10], which collects data on bandwidth and other information about the host on a given network and presents the information on a web page. All of these projects, while significant, cannot

easily allow the user to go through large data sets and draw conclusions quickly.

A unique feature is its ability to passively analyze traffic and produce highly informational information, providing a snapshot of current traffic on the network to the client. While INAVD's goal is to provide traffic statistics to the client in close to real time, the data must be presented to the user long enough for them to create ideas and evaluate what is visualized. In order to accomplish this, and time frame is established and all traffic that that has been seen within this time frame is shown. This is known as the edge life within the graph structure. The default edge life is 180 seconds, and is configurable depending on your network conditions. The snapshot that is taken is queued and sent to the client at regular intervals. This interval is known as the refresh rate of the graph/ the default refresh rate is 2 seconds and is also configurable to adapt to the user's needs.

4.1.1.1 System Hardware

INAVD was designed to be suitable for enterprise networks as well as smaller SOHO networks. In order to be able to handle the data associated with enterprise networks the server must be very efficient. Through the uses of multiple threads and carefully constructed data structures and algorithms, we are able to achieve this goal on relatively cheap hardware. The computer running the server is a 1.4 GHz Pentium III with 512 MB of RAM. It is processing up to 1Gb/sec from a span port on a core switch of the Computer Science Department at Texas A&M.

4.1.2 INAVD Internal Operations

There are four main threads associated with the program. This first thread is the main thread that starts all of the worker threads and is responsible for the creation of most data structures. Next is the sniffer thread. This thread uses the pcap library to sniff packets from the network and then store this data in specially crafted containers and place them into a queue that is capable of filtering base on the information in the packets. Then the process thread takes the packets from the queue and gathers all the relevant information that is needed and places them into the graph structure. The client-communication thread handles all communication with clients and retrieves information from the Graph structure and forwards it to the clients.

The packet container contains pointers to different classes used to store data on the different protocols being sent over the network. Currently INAVD supports Ethernet, IP, TCP, UDP, and ICMP. By wrapping the packet data received from the sniffer the server can easily look at any portion of the packet for filtering and data analysis.

The graph structure has been designed to be able to store millions of edges and sort through them and gather information very quickly. In traditional computer science courses students are taught to represent graphs using adjacency matrices and list. Adjacency matrices are too memory intensive and adjacency list are too slow for our

application. Instead, the graph structure is based on a hash map. The data structure of the graph is a hash map whose key is the IP address of the source node of the edge and the data is another hash map. The second hash map's key is the IP address of the destination node of the edge and the data is an object containing all the data needed to be stored about the edge.

This structure allows the server to accomplish its goal of being able to get graph information and update the graph very quickly. Currently the information stored about an edge includes bandwidth usage for the last 6 seconds, the time of the last known traffic, and connection information including ports and protocols. The bandwidth usage and the last known transmission statistics allow us to assign each edge a weight. The weights in the graph are based on a weighted average of the last 6 seconds worth of bandwidth. The most recent bandwidth information is weighted more heavily than the older data. By storing the last access time, we can correlate the data stored in the last six seconds with the current time. By averaging over 6 seconds the user is allowed more time to see changes in the bandwidth and because the more recent statistics are given more weight a spike in usage is still seen as a spike in the weights. Connection information is stored to allow a user to gather information about the services that are generating the traffic. When the client performs a request for node data the server gathers all the connection information on edges from the node and relays them to the client to be displayed. This data can show if someone is going to web sites or other services, on the web, assuming that the servers being access are running on standard ports.

4.2 Communication Protocol

The communication protocol between the client and the server has several modes. Currently the different types of communication that can take place are configuration, graph refresh, node data request and response, filter configuration, and removed nodes.

Type: Config (1)

[1 Byte: Type][4 Bytes: Size of xml][xml]

Type: Graph Refresh (2)

[1 Byte: Type][4 Bytes: Number of Edges]

[Edges (12 Byte increments)]

Edge: [4 Bytes: Source IP][4 Bytes: Destination IP]

[4 Bytes: weight]

Type: Node Data (3)

Request: [1 Byte: Type][4 Bytes: IP]

Response: [1 Byte:Type][4 Bytes: Size of xml][xml]

Type: Removed Edges (5)

[1Byte: Type][4 Bytes: Number of Edges]

[Edge without weight (8 Byte Increments)]

Edge without weight:

[4 Bytes: Source IP][4 Bytes: Destination IP]

4.2.1 Configuration

When the client connects to the server the first thing the server does is send the client the configuration that is currently running on the server. The configuration is sent in an xml format that is easy to parse. If the client would like to change a parameter, the client sends the server a message of this type and the server changes the settings and then resends the configuration out to all the clients to make them aware of these changes. Current configurable options for the server are the refresh rate at which the server sends a graph refresh or dead node data, and the edge life, which is a measure of how long an edge will remain in the graph without transmitting data. An example of the configuration XML is as follows:

```
<RefreshTime>2</RefreshTime>
<EdgeLife>180</EdgeLife>
```

4.2.1.1 Graph Refresh

When designing the protocol for resending the graph every refresh second, efficiency is important. If we were to set the refresh rate to 1 on a corporate network where there was enough traffic to generate just 20,000 edges with either a bitwise protocol or a ASCII base protocol, like the one used for the configuration, we would consume the entire link. This demonstrates the need for the most efficient protocol that we can produce. It was originally decided that we must resend the entire graph every refresh second because the weights associated with the edges are dynamic and continuously changing. Future versions of INAVD may change this so that there are channels types in the communication protocol for new edges and modified edges. This is to allow for any edges that were idle during the refresh period to not be resent, resulting in a smaller graph. The current protocol for the graph refresh is a bitwise protocol because of these efficiency considerations. An ASCII xml based protocol would have taken much more processing power and produced much more data to transmit across the network. In our bitwise protocol it takes 4 bytes to send an IP address. In an ASCII based protocol IP address take anywhere from 7-15 bytes to transmit. Because we are using a bitwise protocol an edge takes only 12 bytes however an ASCII protocol would have taken 17 – 40 bytes (not including XML tags if used). Because the data is stored in the graph in its bitwise (i.e. non-string form) representation it is also much less computationally intensive to send the data in a bitwise manner.

4.2.1.2 Node Data

When the user clicks on a node in the client the client will represent them with more information on that node. These items currently include the IP address, the DNS name, bandwidth usage inbound, outbound, and total, and a listing of the connections to and from that node. The connection information shows more than what is seen by looking at the

visualization. Port numbers and protocols are presented and help users determine which services that node is offering and what services it is connected to on other computers. The node data communication protocol actually encompasses two different protocols – one for the server and one for the client. The client uses the node data request protocol in which it sends the IP for which it is requesting information in a bitwise fashion. The server then processes that request and sends the client a response in XML using the same format as the configuration data but with the node data type in the type field of the packet. Because the data is presented to the client in XML it is easy to parse and display the information on the screen in a string format to the user.

4.2.1.3 Removed Edges

Because the client needs as many CPU cycles as possible for rendering nodes and edges it is important to ensure that all work that can offloaded to the server is done so. This is the logic behind the removed edges protocol. The client would have to do a lot of work to determine what was not in the current graph refresh as opposed to the last one. Because of this, the server stores all the removed edges since the last graph refresh, and sends them to the client at the same time as the new graph refresh. The protocol for the removal of edges is similar to the protocol of the graph refresh, however, when removing nodes the weight of the edge is not important and can actually be inferred to be zero.

4.2.1.4 Future Mapping Methods

Active mapping methods could be used to add more depth to the network maps. One such active mapping method would be to perform trace routes on all or some of the nodes in the network. This presents two problems. How do we perform these trace routes quickly and how can we keep track of which nodes along the way need to be deleted upon the original edge being deleted. Performing trace routes would also have the side effect of decreasing the visual information available about the connections. If we tried to map out the nodes there would be routers in between all of the traffic that wasn't on the same subnet. Because of this it would be impossible to visually infer where the traffic to or from a given node was coming from. Still trace routes could be useful to visually see the load on different links between the routers both on the local network and the Internet. On links that the server is unable to capture all traffic from, such as the Internet, the load shown would be the load created from the nodes we can capture data from. So in our test case we were gathering information from a span port in the computer science department that gathers its information from a core switch. This means that we would be able to see the load place on all routes from all the computers in the computer science department that were connected to that switch.

Another active mapping method might include gathering information about routing from the routers. We could also choose to implement passive mapping through static routing information used to infer routes and display the graph appropriately. This solution, however, would not be scalable

and would be limited to the SOHO users, or the very determined system administrator.

4.3 *The INAV client*

The INAV client controls the visualization of the data received from INAVD and handles configuration and node connection information tasks.. The largest issue with the design and structure of the client is the visualization of large sets of data. This task is non-trivial, introducing computationally intensive worker processes and memory, but also problems associated with the display of large amounts of data. To overcome the first two problems, we design the client around the *prefuse* [7] toolkit, as *prefuse* provides rendering capabilities and a framework for physics based modeling utilizing a graph based structure. One shortcoming of the *prefuse* toolkit is that dynamic graphs are not supported by default. Initially this was very difficult to overcome and could not be ignored as the real-time state of a network is important, and more specifically – the addition and removal of nodes/edges.

The goal of the INAV client is to display network graph information and output node connection information. Given the size of the network graph and amount of data sent over the communications socket, accomplishing our goal was difficult and required limiting the rate at which the network graph is sent over the network socket and the rate at which nodes disappear (edge life). The setting which limits the sending of the network graph is called the *RefreshTime*, and its default value is 2. The rate at which nodes/edges are removed from the graph is based upon the last point at which traffic as seen, and is known as the *EdgeLife*. The *EdgeLife* default value is 180 seconds (3 minutes).

4.3.1 *System Hardware*

The INAV client was written to support a variety of platforms, and so it is written in Java, and because it uses generic data types, it requires at least Java version 5. In order to support the large amount of data, multiple threads are utilized, and through the careful construction of data types and algorithms we are able to achieve decent performance on mainstream workstation hardware. We had success running our client application on an AMD 3200x2 CPU with 2Gb RAM, running Windows XP, as well as success running on an Intel Core Duo 2.2Ghz with 1Gb ram, running Linux. The only OS we encountered problems on was with an Apple PowerBook Pro, running OS 10.4.9.

4.3.2 *INAV client Internal Operations*

There are four threads used by the client application. The first thread is the main thread that is used to spawn all of the worker threads, and then is used to perform the node and edge construction. The next thread is the rendering thread. This is the main *prefuse* thread and is the rendering to the display as well as the algorithms behind the physics based modeling and storing the underlying graph data structures (a series of tables) which describes the current state of the on-screen elements.. There are two network threads, the network socket

reader and the corresponding writer. The network socket writer sends requests to the server – which can be configuration XML or node connection-information requests. The network socket reader thread controls the input of the configuration data, network graph data and node connection data.

The data flow between the network and the main worker thread – controlling the rate of input from the network to the display. The network is constantly writing data to a queue data structure, and the display is constantly removing data from the queue. By using this model of control, we are able to prevent blocking in both the network writer and the display. A challenging problem was that the traffic from the network arrives in “chunks” and being able to render this “chunk” all at once caused many problems (most notably was large amounts of lag in the system whenever this operation occurred) which were solved by implementing a timer. This timer “pushes” data to the display at the rate of 250 nodes per second. This helps offset the initial load of rendering tens of thousands of nodes at the same time.

In addition to the graph structure another key data structure used is an object registry storing arbitrary data objects into a Hash Map containing not only UI object elements, but also other data structures such as array’s and other Hash Maps. The concept of an object registry is powerful, as it allows objects to be accessible from any other data structure – allowing for objects to access and modify each other that regardless of their visibility status.

Other techniques used are asynchronous data input and interaction methods, which combined provide a direct interactive experience with the data. (figure 11) Rutkowski[7] calls it the principle of *transparency* When transparency is achieved, “the user is able to apply intellect directly to the task, and the tool itself seems to disappear.” There is nothing physically direct about using a mouse to interact with a node or edge on the screen, but if the temporal feedback is rapid and comparable, the user can obtain the illusion of direct control. [8] A key psychological variable in achieving this sense of control is obtained by the responsiveness of the INAV system, where the visual feedback occurs 250 times per second. This falls under the theoretical 1/10th of a second limit developed by Shneiderman [9]

4.3.3 *The INAV client interface*

The interface for controlling the behavior of the server and client is written using its own framework. This custom framework – a derivative of java swing components – allows for configuration options to be rapidly added to a window as a module. Utilizing visual memory patters and attention techniques are also important, as the usefulness of the INAV system largely depends on state of the network. By setting the origin of all newly created nodes to be the center of the screen and directing their movement towards their connected nodes, INAV can utilize a *pull cue*, where a new object appearing in the scene pulls attention towards it. [10] Visual attention is

not strictly limited to the *pull cue*, as INAV also uses a *push cue* which is defined as symbol which tells someone where to look. [8] By using arbitrary colors to define the amount of traffic between the nodes in the graph, INAV is able to draw attention to potential bottleneck issues within the network (figures 1, 2, 8.) using *push cues* to provide visual feedback about their state. Regarding the overall behavior of the INAV display, these different cues play an important role in directing the users' attention: 100 msec to shift attention with a *pull cue* and between 200 and 400 msec to shift the attention based on a *push cue*. [8]



Figure 8 : Red lines show heavy traffic and could be a potential cause of a network slowdown; whereas light green illustrate a negligible amount of traffic between nodes.

Spatial recognition patterns are also used to discover interesting traffic patterns within the network. Since the INAV system is physics based, each node has a mass which affects its rate and ability to be influenced by other nodes via their interconnected edge (which is considered to act as a strut – a spring with dampening properties). In addition to the physics based properties between nodes (via their edges) there are also gravitational forces acting upon all the nodes. INAV uses a negative gravitational constant, so that the nodes will want to separate from each other. This is important, as a valuable asset to the INAV project is the visual recognition and spatial sense-making that occurs. Playing upon this concept of gravity and the forces induced by the struts, the more connections made to a node, the heavier its mass becomes, in turn, influencing its behavior with other nodes in the system (figures 9, 10).

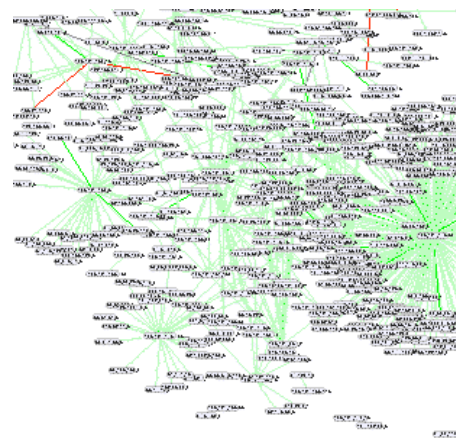


Figure 9

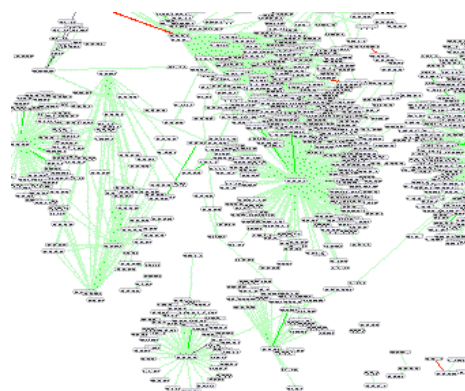


Figure 10

Figures 9, 10 illustrate the gravitational and strut forces acting upon the nodes in the graph. Figure 9 and 10 are the same section of the graph and are approximately one second apart, showing the emergence of nodes in the network.

Information	
IP Address	128.194.149.4
DNS	r2d2.cs.tamu.edu
Performance (B/s)	
Total:	657412
Inbound:	28478
Outbound:	628934
Connections:	105

Figure 11 illustrates the concept of transparency, where the user clicks on a node; the request for additional data is sent to the server and subsequently displayed in a mouse over tooltip and a dialog box, as shown here.

5. CONCLUSION

By developing the INAV application, we are able to provide a visually stimulating tool that is useful for information discovery and sense-making. INAV is able to bridge the current gap in software – combining the usefulness of packages such as Cheops-ng with the robustness of popular diagnostic utilities such as Nmap and WireShark.. The INAV system is considerably different than other systems in that it integrates a visualization layer on top of a real-time, ultra high performance collection agent, whereas current applications only have a subset of these features. INAV is different as its goal is to collect active network data in a passive and robust manner, and provide interaction and feedback for that data, affording sense-making and information discovery.

Through the use of clever techniques used to collate data, an optimized network protocol between the clients and sever, and a carefully designed visualization, INAV has succeeded.

6. REFERENCES

- [1] Brent Priddy, Cheops-ng <http://cheops-ng.sourceforge.net/>
- [2] Stockinger, K. Kesheng Wu Campbell, S. Lau, S. Fisk, M. Gavrilov, E. Kent, A. Davis, C.E. Olinger, R. Young, R. Prewett, J. Weber, P. Caudell, T.P. Bethel, E.W. Smith, S. High Performance Computing Research Department (HPCRD/LBNL); *Network Traffic Analysis With Query Driven Visualization* SC 2005 HPC Analytics Results
- [3] Wills, Graham J., Bell Laboratories (Data Visualization Group): NicheWorks—*Interactive Visualization of Very Large Graphs*, <http://willsfamily.org/gwills/>
- [4] Zavgren , John Richard Zavgren, Jr. Systems and methods for visualizing a communications network, *United States Patent and Trademark Office # 6909696*, June 2005.
- [5] pthreads: Native Posix Thread Library: <http://people.redhat.com/drepper/nptl-design.pdf>.
- [6] **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.** Libpcap: network statistics collection, security monitoring, network debugging. <http://sourceforge.net/projects/libpcap/>.
- [7] The prefuse visualization toolkit. <http://prefuse.org/>.
- [8] Wireshark <http://www.wireshark.org/>.
- [9] TCPDump <http://www.tcpdump.org/>.
- [10] Ntop <http://www.ntop.org/>
- [11] Rutkowski, C. An introduction to the Human Applications Standard Computer Interface, Part 1: Theory and principles. *BYTE* 7(11):291-310.
- [12] Ware, C. *Information Visualization: Perception for design*. Morgan Kaufmann, 2004, San Francisco, CA.
- [13] Shneiderman, B. *Designing the User Interface*. Addison-Wesley, Reading, MA.
- [14] Jonides, J. Voluntary versus automatic control over the mind's eye. In *Attention and Performance*, vol. 9, ed. J. Long and A.AD. Baddeley, 187-203. Erlbaum, Hillsdale, NJ.