A Visualization Tool For SYN Flooding Attack Detection

Joshua Foster
Supervisors: Drs. K.R. Subramanian and Gail J. Ahn
May 2002

Department of Computer Science
The University of North Carolina at Charlotte
Charlotte, NC 28223-0001

A Visualization Tool For SYN Flooding Attack Detection

Joshua Foster
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, NC 28223-0001

Abstract - **Computer system attacks and misuses are of great concern in today's highly network-based society. Of all the types of intrusions, denial-of-service (DoS) attacks are perhaps the most widely known. The goal of completely secure systems will never be realized; therefore intrusion detection will continue to be necessary. Manual methods of intrusion detection are no longer feasible with the large-scale, high-traffic networks of today. Graphical representation of network data presents more information to the user in a shorter amount of time and enables recognition of patterns not evident in textual data. Of all the types of intrusions, denial-of-service (DoS) attacks are perhaps most widely known. The SYN flooding DoS attack is the most popular and easiest to implement of these attacks. In this paper, we discuss and demonstrate a tool for visualization of network data specifically geared toward SYN flooding attack detection.**

1 Introduction

Intrusion detection has become a major concern in this age of large-scale, high-volume networks. Most researchers agree that no system that is connected to a network can ever be completely secure. Computer systems are not capable of detecting and preventing all possible types of misuses and attacks, so the burden falls on the system administrator to review and analyze network usage data to detect possible intrusions [4]. In the past, manual analysis of network traffic and audit logs was used to identify the causes of security incidents. However, networks have grown in size and volume to the point where this type of textual analysis is no longer sufficient. Data visualization, or graphical representation of large amounts of information, seems well-suited to the task of representing network infrastructure and traffic flow, from the domain level down to individual packets of information sent between devices.

Many forms of intrusion exist. Among these, denial-of-service (DoS) attacks have recently been in the spotlight, as they have been successfully executed against a number of popular web services in the past few years [4]. SYN flooding is a highly popular type of DoS attack, owing to its relatively easily implementation and difficulty in tracing.

In this paper, we will discuss a tool for the representation of network data in graphical form. Specifically, we consider the attributes of the data from which we can construct patterns that have a high possibility of making SYN flooding attacks easily visible. We will begin with a description of the SYN flooding attack, discuss methods for detection of DoS attacks and show examples of previous work involving visualization of network data. We will then discuss in detail the implementation and use of the visualization tool, and conclude with the results of our testing and experimentation with the program.

2 SYN FLOODING

The SYN flooding attack is an easy-to-execute type of DoS attack that has become highly popular. If executed successfully, the attack will render a target system unable to accept new connections from other systems during the attack and for some time after. The attack may be made difficult to trace, which is another reason for its popularity. Hackers may “spoof”, or falsify, their IP addresses with relative ease.

SYN flooding is made possible by the connection mechanism of TCP. In order to establish a TCP connection, two devices must first complete a *three-way handshake*, a mechanism which requires three packets of information to be sent between the systems. A system (*client*) seeking a connection with another system (*server*) must first send a SYN packet to that server. The header of this packet contains a synchronization flag and a randomly generated sequence number. If the server is willing to accept the connection, it will return a SYN/ACK message. This is a synchronization/acknowledgment packet and contains several important pieces of information. The packet acknowledges the previous packet by including in its acknowledgment field the previous sequence number incremented by one. In addition, the packet contains another randomly generated sequence number. When the client receives the SYN/ACK packet, it will generate an ACK packet. This packet has no sequence number, but includes an acknowledgment for the SYN/ACK packet as before – the previous sequence number is incremented by one. The TCP connection is established, or *closed*, when the server receives this last packet.

What makes a SYN flooding attack possible is that after the second (SYN/ACK) packet, the server must keep data for this half-open connection in memory and wait for the client to respond with the final ACK packet. At this stage, the connection is referred to as *open*, because it has been initiated but not finalized. Two packets of the three-way handshake have been sent and received, and the server will wait a certain length of time for the third and final packet. A SYN flooding attack scheme does not send this final ACK packet. Instead, it sends many SYN packets in a very short period of time, trying to establish multiple connections with the server. The server replies to all of these packets and waits for each acknowledgment. If the client sends enough SYN packets in a short amount of time, the server’s connection buffer will overflow. The results depend on the operating system, but in all cases the service that was attacked is rendered useless, and legitimate users requesting connections with that service are denied [5].

3 MOTIVATION FOR VISUALIZATION

SYN flooding attacks take place at the network level, so we must use individual packet data, such as that produced by *tcpdump* or *WinDump*, to identify them. These programs produce log files of packet header data. To detect an attack, one must scan through a log file looking for a large amount of open connections being established in a short amount of time. This is possible, but would be very time-consuming and error-prone. These log files may contain hundreds of thousands of packet header entries per second. In addition, reading text is a serial process, which means only a small part of the data can be focused on at any particular time. Receiving information in this way, one may not be aware of the patterns produced by a SYN flooding attack. An image, on the other hand, is interpreted in parallel. Large volumes of information can be stored abstractly in a single image. This form of data is easier for the human mind to process, which means the user may gather more data in less time. Given the fact that intrusion detection is quickly becoming an essential part of network administration, it is our objective to make intrusion detection easier and faster by building a tool to represent network data graphically.

4 PREVIOUS WORK

Visualization of network data is not a new concept. In this section we present two previous studies, both dealing with the representation of data from network log files.

The work of Juslin (Figure 1) consists of reading tcpdump data into a Perl script, parsing it, and displaying it with GNUPlot. The data is formatted with respect to three axes; time, number of packets, and SDP, or source address, destination address, and destination port [3]. This produces interesting visuals from which the user may gather a large amount of information. For example, there are several long, narrow blocks in the visual, pointed to by arrows. These are port scans, in which the attacker will quickly and sequentially query all the ports on a system. Also, spikes in network traffic appear as 'towers' in the graph. However, the visualization is static. It presents a picture of network activity sometime in the past, and hence cannot be used with realtime network traffic.

Another interesting piece of work comes from Erbacher, Walker, and Frinckle. This visualization (Figure 2) is more in the direction of this project, as it represents one central system and all the clients attempting to connect to it. The network data is shown as an animation over time that can be paused, advanced, or rewound. Hence, it is feasible to manipulate the system to display realtime data. There is a wealth of information contained in each frame. For example,

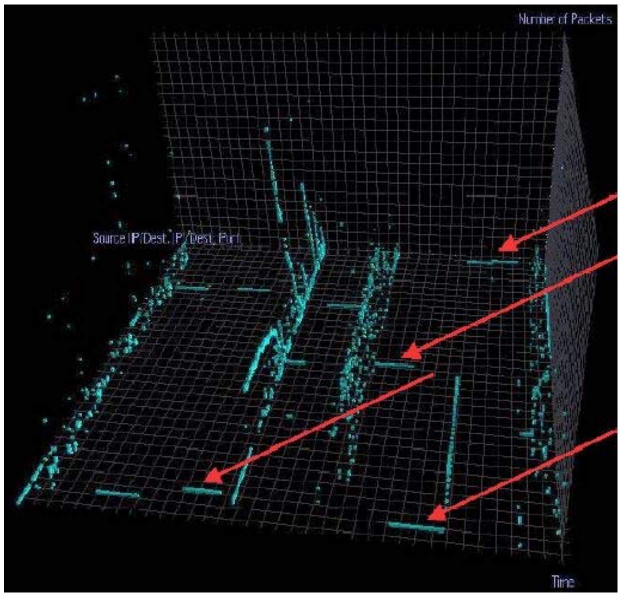


Figure 1. Previous Work – Juslin

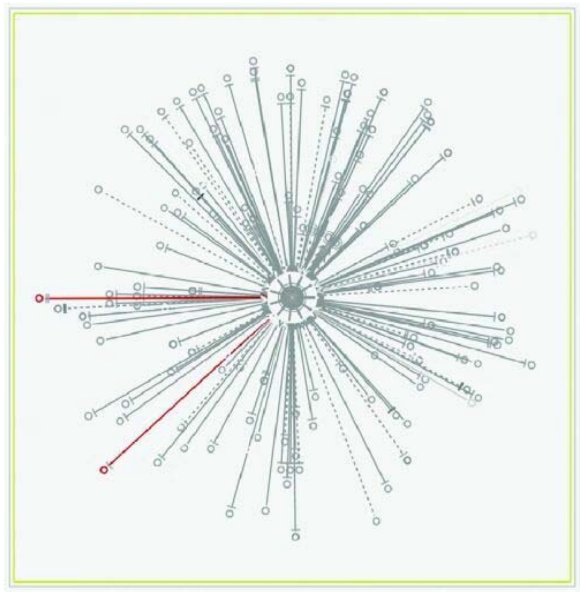


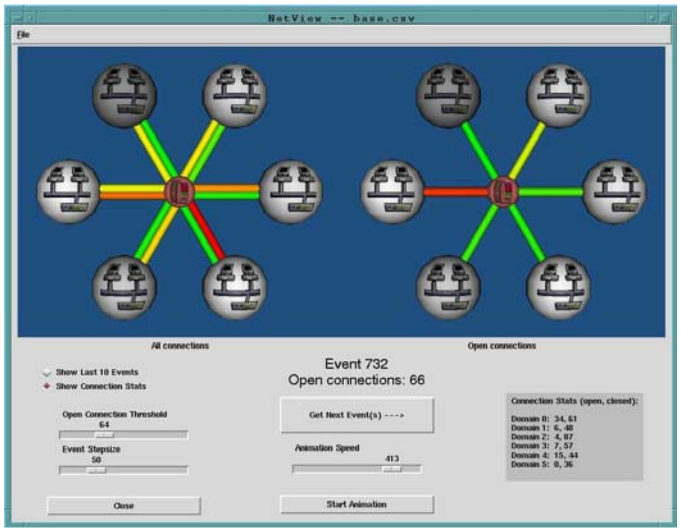
Figure 2. Previous Work – Erbacher

the node and line colors represent suspicious activity as determined by the system, such as users attempting to gain extended privileges. The lines themselves are of different styles which represent the type of service being used, such as Telnet or FTP. The radius of each node represents its network locality to the central system. The nodes near the center are likely to be on the same LAN as the target system, while the nodes near the edge of the visualization are connecting from completely different networks [1]. The problem with this type of visualization is that it is abstract and non-interactive. Exact data such as time and IP addresses is consumed by the application, and cannot be recovered through the visualization. The user may watch an attack in progress, but is powerless to take action because there is no way to obtain the exact time or IP address of the attacker from the application.

5 THE VISUALIZATION ENVIRONMENT

Our visualization tool seeks to convey a large amount of relevant information to the user while keeping the interface clean and organized. We represent network data in the form of open and closed connections, the most relevant form for detection of SYN flooding attacks. The log files we use represent a large number of clients connecting to a central server; these clients are divided into arbitrary domains which surrounding a central system (Figure 3). This system has a pair of thick lines extending to each domain; these represent links between the server and all of the clients in a particular domain. The counterclockwise side of these links represents the open connections to the server, while the clockwise side represents closed connections. Each link is color-mapped on a scale of green to red representing its number of connections in relation to the other links of the same type. The domains are mapped from black to white based on the total number of connections between their clients and the server. The user simply clicks a button to get the next group of open and closed connections from the log file and update the visualization accordingly.

The bulk of the program, consisting of the GUI, data manipulation and supporting code, was implemented in the Python scripting language. A C++ class was built to parse the log files and return the relevant information to Python. In our case, this relevant information is in the form of *events* representing opening and closing connections. The C++ class parses a log of packet headers looking for synchronization and acknowledgment packets that are characteristic of a three-way handshake. A set of VTK wrappers for Python is used to handle the graphical aspects of the visualization. The Visualization Toolkit is an open-source, platform-independent 3D graphics library for geometric rendering and image processing.

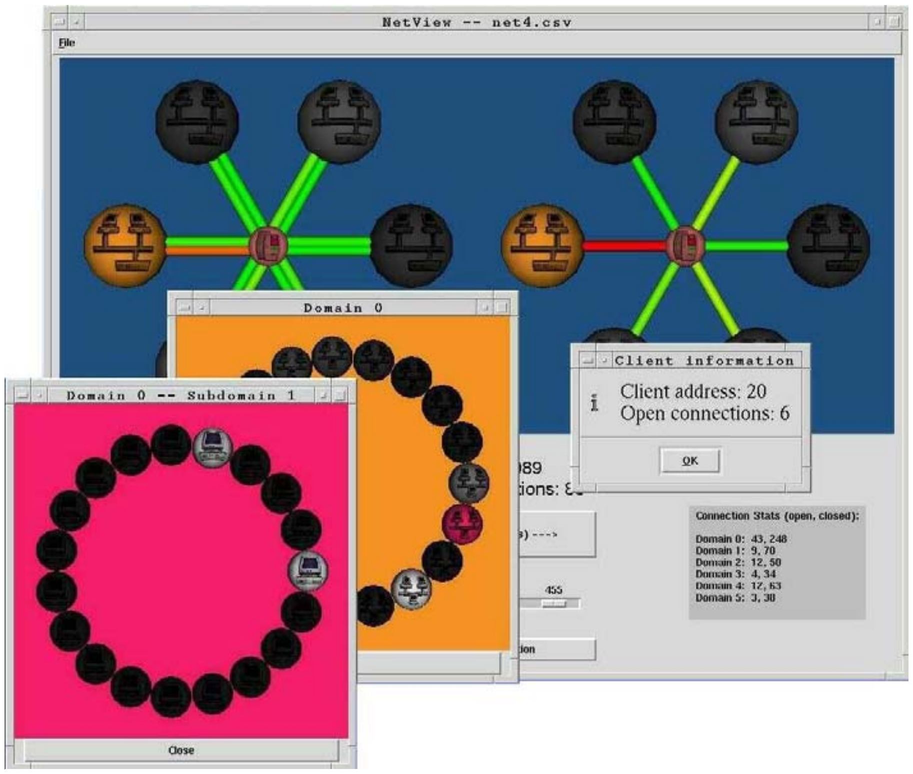


```
Event 732
Open connections: 66
```

```
Connection Stats (open, closed):

Domain 0:  34, 61
Domain 1:  6, 48
Domain 2:  4, 87
Domain 3:  7, 57
Domain 4:  15, 44
Domain 5:  0, 36
```

Figure 3. NetView interface



Client address: 20
Open connections: 6

```
Connection Stats (open, closed):

Domain 0:  43, 248
Domain 1:  9, 70
Domain 2:  12, 50
Domain 3:  4, 34
Domain 4:  12, 63
Domain 5:  3, 30
```

Figure 4. View of subdomains and clients within a domain

Perhaps the most important feature of the program is interactivity. Each log file we process has an average of two thousand clients connecting to the central server. For clarity, these are grouped into subdomains distributed within six top-level domains. When a domain is clicked, a window will appear showing all the subdomains within that domain. The domain in the main window will also change color to reflect the background color of the new window (Figure 4). Each subdomain in the new window is mapped from black to white, representing the total number of open connections for all the clients contained within it. A click on any subdomain opens up a third window showing all the clients within that subdomain. These are also color-mapped based on their numbers of open connections. The user may click on any of these individual clients to obtain information such as their IP address and their number of open connections. Other features include right-clicking on any domain or subdomain to return the address of the client with the largest number of open connections within that group.

An animation feature is included in the program. This simulates real-time data coming into the program from the network, an objective which we eventually hope to accomplish. Another feature of our tool is the server open connection threshold. A main concern during a SYN flooding attack is the amount of memory used by open connections on the server. If the maximum number of open connections is reached, the system will no longer be able to accept incoming connection requests. A slider is provided for the user to set the buffer size of the server. If the number of connections coming into the central system exceeds this number, the server will flash along with the domain with the highest number of open connections at that time.

6 Experimentation and Results

The central idea of the program is that SYN flooding attacks may be represented by a large number of open connections, and the method of coloring that we use will make these attacks easily visible to the user. We had five datasets available with which to test this idea. The data is courtesy of the Information Exploration Shootout project [2], and represented normal network traffic along with four simulated attacks. They are provided in the form of CSV (comma-separated value) files, parsed from tcpdump data. For security reasons, the IP addresses have been removed. We describe the datasets and the results of our experimentation below.

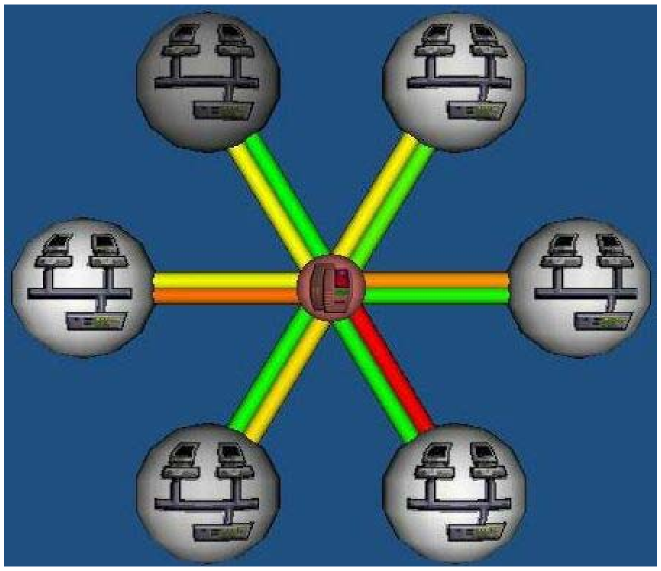


Figure 5. Baseline traffic

6.1 Baseline.csv

The first file available for analysis is an eighteen-minute sample of normal network traffic. Stepping through the file, one can see that most open connections get closed very quickly, usually in the next time step. This type of behavior is what one would expect during normal network operation. After reading through the entire file, it is evident from the domain colors that connections are relatively evenly distributed, and there are very few open connections in comparison to closed connections.

6.2 Net1.csv

This log represents an IP spoofing attack. Normally, a SYN flooding attacker will want to disguise his location, so packets are made untraceable with a "spoofed", or falsified, source address. It is evident in the visualization that a majority of open connections are coming from one single domain, and right-clicking on that domain reveals that 25% of these connections are from one single client. This, coupled with the fact that there are a larger-than-average number of open connections coming from the other domains, may suggest that the attacker is only spoofing his/her IP address for a small part of the simulation.

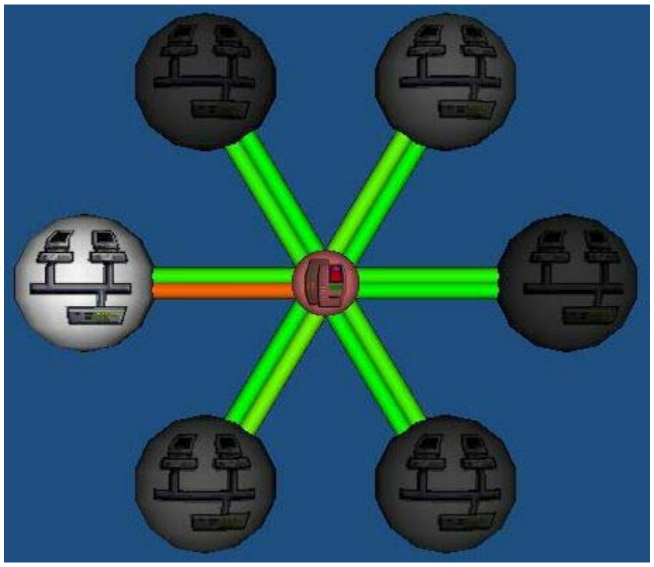


Figure 6. IP Spoofing attack

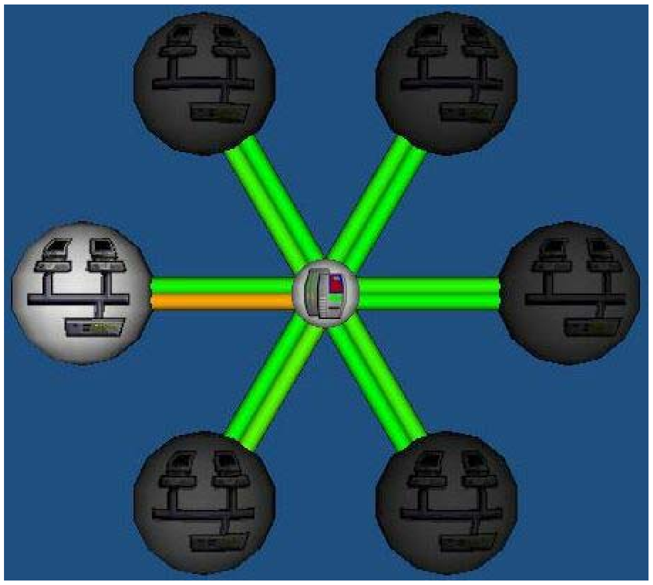


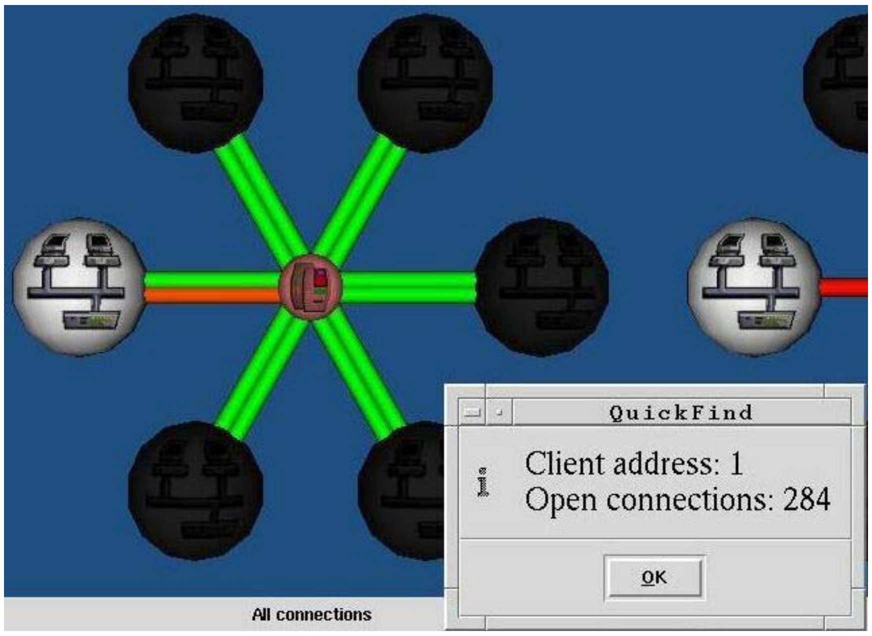
Figure 7. Password guessing attack

6.3 Net2.csv

Net2.csv is a simulated password-guessing attack. In this type of attack, a 'dictionary' of commonly used passwords is sent, one word at a time, to the server. If the server indicates an incorrect password, the attacking program will try another word. There is some doubt as to whether this dataset is relevant to our objective, since password guessing is a data attack rather than a network-level attack. However, running this log file through the program indicated that there were a larger-than-normal number of closed connections generated. This may be an indication of multiple password-guessing attempts on the server.

6.4 Net3.csv

This dataset is the most visually interesting of the five. The simulated attack is port scanning, in which an attacker would query every port on the target system, one by one, in a very short amount of time. One would expect a large number of short-lived connections in rapid. This was exactly what was shown in the visualization as the log was parsed. The network traffic was almost exclusively from one single client. In fact, this client established over 8600 connections in only 18 minutes. Given this, we can deduce that each connection must be active for a very small amount of time. These rapid, short connections are exactly what are expected of a port-scanning attack.



QuickFind

Client address: 1
Open connections: 284

Figure 8. Port scanning attack

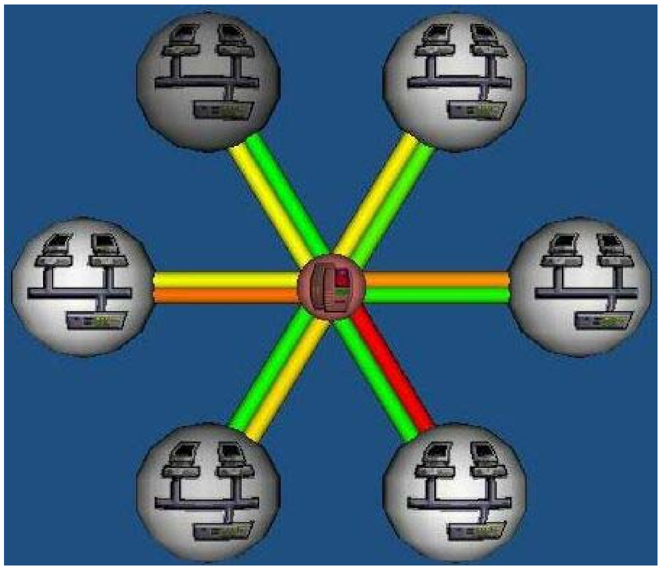


Figure 9. Network hopping attack

6.5 Net4.csv

Network hopping is simulated in net4.csv. This is simply another way of disguising the source of an attack where the SYN packets are routed through multiple intermediate nodes before they reach the central server. Visually, this log resembles the baseline traffic log. Connections are relatively evenly distributed among all domains. One may surmise that if attack packets were being routed randomly, they would arrive at the server through random domains, thus appearing as normal network traffic.

7 Conclusion

This tool is interesting and valuable in that it conveys more relevant information and a larger volume of data than textual analysis of log files, the information is conveyed in an easily-comprehensible from, and events relative to SYN flooding are highlighted and brought to the attention of the user. We envision this program as one day being a useful tool to system administrators.

However, several other objectives need to be accomplished. First, the incorporation of real-time data analysis is important to network visualization. The program currently requires preprocessed data, but it may eventually be modified to use packet header data directly from the network level. Also, the time dimension needs to be somehow incorporated into the program. Perhaps in the future the program will graph open connections versus time for a selected client, subdomain, or domain. This would be a useful tool since SYN flooding attacks are characterized by large amounts of connections being opened in a short period of time. Among others, these enhancements would make the program a much worthier tool for intrusion detection.

References

[1] R. Erbacher, K. Walker, and D. Frincke, "Intrusion and Misuse Detection in Large-Scale Systems", in IEEE Computer Graphics and Applications, Jan-Feb 2002.

[2] The Information Exploration Shootout, The Institute for Visualization and Research. [http://iris.cs.uml.edu:8080/network.html]

[3] J. Juslin, "Intrusion Detection and Visualization Using Perl", O'Reilly Open Source Convention, San Diego, 2001.

[4] R. Kemmerer and G. Vigna, "Intrusion Detection: A Brief History and Overview", in Security and Privacy, 2002.

[5] "TCP SYN Flooding and IP Spoofing Attacks", CERT Advisory CA-96.21. [http://www.ciac.org/ciac/bulletins/g-48.shtml]

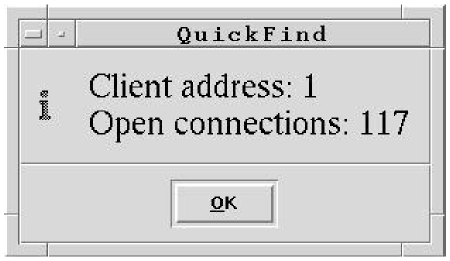
APPENDICES

Several reference materials follow. First is a user's guide for the NetView tool, followed by the slides of a presentation given on this tool on May 1, 2002. The last supplement included is the source code to the program.

NetView, the network data visualization tool discussed in this paper, consists of a python script with supporting C++ code. It is invoked with a command line such as `python netview.py 2`, with the optional number being 0 for baseline traffic, or 1 through 4 for `net[1-4].csv`. The program will begin with the geometry set appropriately for the log file (based on number of clients broadcasting in the file), and all connections will be cleared. Adjust the event stepsize slider to indicate the number of events wanted for each click of the button.

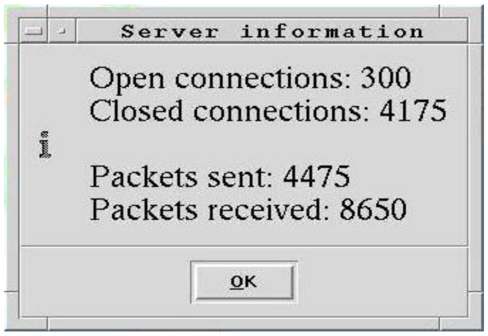
When the *Get Next Events* button is clicked, the window will be updated with the first group of open and closed connections from the file will be shown. If the option *Show Last 10 Events* is selected, the last ten of these will be displayed in the message box. Otherwise, the message box will display the number of open and closed connections for each domain. As you continue to get events, the domains and links will update their colors to reflect their numbers of open and closed connections.

To view an individual client's contribution to the visual, click on the appropriate domain. A window will open up showing all the subdomains in that domain. Notice that the domain's color will match the new windows background color. Click on the appropriate subdomain to show a window with all the clients in that subdomain. The window coloring also applies here. A click on any client brings up a message box with that client's address and number of open connections.


QuickFind

Client address: 1
Open connections: 117

Figure 10. QuickFind


Server information

Open connections: 300
Closed connections: 4175

Packets sent: 4475
Packets received: 8650

Figure 11. Server info

QuickFind, server information, and animation are three important features. In an attack situation, the user will want to know immediately which particular client is initiating the most open connections. One way is to descend through the domain and subdomain level, each time clicking on the object with the highest number of open connections. An alternative to this is to click with the right mouse button on the domain or subdomain of interest. This will bring up a message box with the address of the client that has the highest number of open connections in that group (Figure 10). The server, when clicked on, will display its number of open and closed connections as well as the number of packets it has sent and received (Figure 11). Animation is toggled on and off with the *Animation* button, and its speed is controlled both by the slider and the event stepsize.

Source Code

The source code to NetView consists of five files: `netview.py`, `domain.py`, `subdomain.py`, `link.py`, and `netread.cxx`.

NETVIEW.PY

```
#!/usr/bin/python

from newRenderWidget import *
import tkSimpleDialog, tkMessageBox
from domain import Domain
from link import Link
from time import sleep
from string import atoi
import netread


NUM_SOURCES = [ 1685, 2283, 1942, 2141, 2278 ]

FILENAMES = [ "base.csv", "net1.csv","net2.csv",
              "net3.csv", "net4.csv" ]

USAGE = """Usage:  python netview.py [n]
        where n is:
          0 - baseline.csv
          1 - net1.csv
          2 - net2.csv
          3 - net3.csv"""

class NetView( Frame ):

  def __init__( self, parent=None ):

    if len( sys.argv ) < 2:  fnum = 0
    else:
      try:  fnum = int( sys.argv[1] )
      except ValueError:
        print USAGE
        sys.exit()
      if fnum < 0 or fnum > 4:
          print USAGE
          sys.exit()

    open_coords = [[[ -10, 0, -.3 ], [ -.5, 0, -.3 ]],
                   [[ -5.4, 0, 8.66 ], [ -.4, 0, 0 ]],
                   [[ 4.6, 0, 8.66 ], [ -.4, 0, 0 ]],
                   [[ 10, 0, .3 ], [ .5, 0, .3 ]],
                   [[ 5.4, 0, -8.66 ], [ .4, 0, 0 ]],
                   [[ -4.6, 0, -8.66 ], [ .4, 0, 0 ]]]

    clsd_coords = [[[ -10, 0, .3 ], [ -.5, 0, .3 ]],
                   [[ -4.6, 0, 8.66 ], [ .4, 0, 0 ]],
                   [[ 5.4, 0, 8.66 ], [ .4, 0, 0 ]],
                   [[ 10, 0, -.3 ], [ .5, 0, -.3 ]],
                   [[ 4.6, 0, -8.66 ], [ -.4, 0, 0 ]],
                   [[ -5.4, 0, -8.66 ], [ -.4, 0 ,0 ]]]

    attk_coords = [[[ -10, 0, 0 ], [ 0, 0, 0 ]],
                   [[ -5, 0, 8.66 ], [ 0, 0, 0 ]],
                   [[ 5, 0, 8.66 ], [ 0, 0, 0 ]],
                   [[ 10, 0, 0 ], [ 0, 0, 0 ]],
                   [[ 5, 0, -8.66 ], [ 0, 0, 0 ]],
                   [[ -5, 0, -8.66 ], [ 0, 0, 0 ]]]


    # --- Build render windows --- #
```

```
ren1 = vtkRenderer()
ren2 = vtkRenderer()
ren1.SetBackground( .1, .3, .5 )
ren2.SetBackground( .1, .3, .5 )
ren1.SetViewport( 0, 0, .50, 1 )
ren2.SetViewport( .50, 0, 1, 1 )

srvSrc = vtkSphereSource()
srvSrc.SetRadius( 1.5 )
srvSrc.SetThetaResolution( 16 )

texMap = vtkTextureMapToSphere()
texMap.SetInput( srvSrc.GetOutput())

srvMap = vtkDataSetMapper()
srvMap.SetInput( texMap.GetOutput())

reader = vtkBMPReader()
reader.SetFileName( "textures/server.bmp" )
self.txr = vtkTexture()
self.txr.SetInput( reader.GetOutput())
self.txr.InterpolateOn()

self.servAct = vtkActor()
self.servAct.SetMapper( srvMap )
self.servAct.SetTexture( self.txr )

ren1.AddActor( self.servAct )
ren2.AddActor( self.servAct )

self.lut_all = vtkLookupTable()
self.lut_all.SetHueRange( .33, 0 )
self.lut_all.SetSaturationRange( 1, 1 )
self.lut_all.SetValueRange( 1, 1 )

self.lut_open = vtkLookupTable()
self.lut_open.SetHueRange( .33, 0 )
self.lut_all.SetSaturationRange( 1, 1 )
self.lut_all.SetValueRange( 1, 1 )

self.lut_domain = vtkLookupTable()
self.lut_domain.SetHueRange( 0, 0 )
self.lut_domain.SetSaturationRange( 0, 0 )
self.lut_domain.SetValueRange( .2, 1 )

self.dom = range( 6 )
self.open_link = range( 6 )
self.clsd_link = range( 6 )
self.attk_link = range( 6 )

self.filename = FILENAMES[fnum]
self.num_srcs = NUM_SOURCES[fnum]
self.num_subs = int( math.ceil( math.sqrt( self.num_srcs/6.0 )))
factor = math.ceil( self.num_srcs/6 )

for i in range( 5 ):
  self.dom[i] = Domain( self, i, self.num_subs, factor, i*factor+1 )

self.dom[5] = Domain( self, 5, self.num_subs, self.num_srcs-(5*factor), 5*factor+1 )

for i in range( 6 ):

  self.dom[i].setPosition( attk_coords[i][0] )

  ren1.AddActor( self.dom[i].getSphere())
  ren2.AddActor( self.dom[i].getSphere())

  self.open_link[i] = Link( i )
  self.clsd_link[i] = Link( i )
  self.attk_link[i] = Link( i )
  ren1.AddActor( self.open_link[i].getActor())
  ren1.AddActor( self.clsd_link[i].getActor())
  ren2.AddActor( self.attk_link[i].getActor())

  self.open_link[i].setPoints( open_coords[i][0], open_coords[i][1] )
  self.clsd_link[i].setPoints( clsd_coords[i][0], clsd_coords[i][1] )
  self.attk_link[i].setPoints( attk_coords[i][0], attk_coords[i][1] )


# --- Build GUI --- #


Frame.__init__( self, parent )
self.master.title( "NetView -- " + self.filename )
self.menubar=Frame( self, relief=RAISED, bd=2 )
self.menubar.pack( side=TOP, fill=X )

self.msg = StringVar()
self.stepsize = IntVar()
self.speed = IntVar()
self.threshold = IntVar()
self.show_what = IntVar()
self.msg.set( "\n\n\n\n\n\n\n\n\n\n" )
self.threshold.set( 40 )
self.stepsize.set( 50 )
self.show_what.set( 0 )

self.animOn = 0
self.cur_event = 0
self.maxall = self.maxopen = self.maxclsd = 0
self.log = None

filembutton = Menubutton( self.menubar, text="File", underline=0 )
filembutton.pack( side=LEFT )
filemenu = Menu( filembutton, tearoff=0 )
filemenu.add_command( label="Reset", command=(lambda s=self: s.open( 1 )))
filemenu.add_separator()
filemenu.add_command( label="Exit", command=self.quit )
filembutton['menu'] = filemenu

self.win = newRenderWidget( self, self, width=1000, height=450 )
self.win.GetRenderWindow().AddRenderer( ren1 )
self.win.GetRenderWindow().AddRenderer( ren2 )
cam1 = ren1.GetActiveCamera()
cam1.SetPosition( 0, 90, 0 )
cam1.SetFocalPoint( 0, 0, 0 )
cam1.SetViewUp( 0, 0, -1 )
cam1.Zoom( 1.8 )
cam2 = ren2.GetActiveCamera()
cam2.SetPosition( 0, 90, 0 )
cam2.SetFocalPoint( 0, 0, 0 )
cam2.SetViewUp( 0, 0, -1 )
cam2.Zoom( 1.8 )
self.win.pack( side=TOP, padx=10, pady=3 )

mid_frm = Frame( self )
Label( mid_frm, text="All connections", width=70 ).pack( side=LEFT )
Label( mid_frm, text="Open connections", width=70 ).pack( side=RIGHT )
mid_frm.pack( side=TOP )

low_frm = Frame( self )
low_lf_frm = Frame( low_frm, width=300 )
low_mid_frm = Frame( low_frm, width=300 )
low_rt_frm = Frame( low_frm, width=300 )

Label( low_lf_frm, text="" ).pack( side=TOP )
Radiobutton( low_lf_frm, text="Show Last 10 Events",
             variable=self.show_what, value=1 ).pack( side=TOP, anchor=W )
Radiobutton( low_lf_frm, text="Show Connection Stats",
             variable=self.show_what, value=0 ).pack( side=TOP, anchor=W )
Label( low_lf_frm, text="" ).pack( side=TOP )
```

```
    Scale( low_lf_frm, label="Open Connection Threshold", variable=self.threshold,
           orient=HORIZONTAL, width=10, length=200,
           to=200 ).pack( side=TOP )
    Scale( low_lf_frm, label="Event Stepsize", variable=self.stepsize,
           orient=HORIZONTAL, width=10, length=200, from_=1,
           to=200 ).pack( side=TOP )
    Label( low_lf_frm, text="" ).pack( side=TOP )
    close_btn = Button( low_lf_frm, text="Close", width=30, command=self.quit )
    close_btn.pack( padx=10, pady=10 )

    self.status_lbl = Label( low_mid_frm, text="", font=("Helvetica", 18 ))
    self.status_lbl.pack( side=TOP )
    next_btn = Button( low_mid_frm, text="Get Next Event(s) --->",
               width=30, height=3, command=self.nextEvents )
    next_btn.pack( padx=10, pady=10 )
    Scale( low_mid_frm, label="Animation Speed", variable=self.speed,
           orient=HORIZONTAL, width=10, length=200, from_=1,
           to=500 ).pack( side=TOP )
    Label( low_mid_frm, text="" ).pack( side=TOP )
    self.anim_btn = Button( low_mid_frm, text="Start Animation",
               width=30, command=self.toggleAnim )
    self.anim_btn.pack( padx=10, pady=10 )

    Message( low_rt_frm, textvariable=self.msg, width=200,
             bg="gray", bd=3 ).pack( side=TOP, anchor=N )

    low_lf_frm.pack( side=LEFT )
    low_mid_frm.pack( side=LEFT, padx=100 )
    low_rt_frm.pack( side=LEFT )
    low_frm.pack()

    self.open( 0 )
    self.pack()


  def open( self, reset ):

    if reset == 1:  self.log.reset()
    else:
      self.log = netread.NetReader( "data/parsed/" + self.filename )

    self.cur_event = 0
    self.maxall = 10
    self.maxopen = 5

    self.lut_all.SetRange( 0, 10 )
    self.lut_all.Build()
    self.lut_open.SetRange( 0, 5 )
    self.lut_open.Build()
    self.lut_domain.SetRange( 0, 10 )
    self.lut_domain.Build()

    for i in range( 6 ):
      self.dom[i].reset()
      c1 = self.lut_all.GetColor( 0 )
      c2 = self.lut_open.GetColor( 0 )
      self.open_link[i].setColor( c1 )
      self.clsd_link[i].setColor( c1 )
      self.attk_link[i].setColor( c2 )

    self.updateStatusLbl()
    self.updateServerStatus()

    if self.show_what == 1:
      self.msg.set( "Last Events:\n\n\n\n\n\n\n\n" )
    else:
      self.msg.set( "Connection stats (open, closed):\n\n\n\n\n\n\n\n" )

    self.win.Render()
```

```
def updateStatusLbl( self ):

  sum = 0
  for i in range( 6 ):
    sum = sum + self.dom[i].open_conns

  msg = "Event " + `self.cur_event` + "\n" + "Open connections: " + `sum`
  self.status_lbl.configure( text=msg )


def toggleAnim( self ):

  if self.animOn == 0:
    self.animOn = 1
    self.anim_btn.config( text="Stop Animation" )
    self.nextEvents()
  else:
    self.animOn = 0
    self.anim_btn.config( text="Start Animation" )


def nextEvents( self ):

  newmsg = "Last Events:\n\n"
  for i in range( self.stepsize.get()):
    conn = self.log.getNextEvent()

    if conn == 0:
      if i == 0:
        if self.animOn == 1:
          self.animOn = 0
          self.anim_btn.config( text="Start Animation" )
        return
      else:
        break

    self.cur_event = self.cur_event + 1

    if conn < 0:                   # connection opened

      x = int( abs(conn)*6.0/self.num_srcs )
      self.dom[x].addOpenConn( abs( conn ))
      self.open_link[x].setColorFrom( self.lut_all, self.dom[x].open_conns )
      self.attk_link[x].setColorFrom( self.lut_open, self.dom[x].open_conns )
      self.dom[x].setColorFrom( self.lut_domain )
      if self.stepsize.get()-i < 11:
        newmsg = newmsg + "Connection opened: client " + `-conn` + "\n"

    else:                          # connection closed

      x = int( conn*6.0/self.num_srcs )
      self.dom[x].addClsdConn( conn )
      self.open_link[x].setColorFrom( self.lut_all, self.dom[x].open_conns )
      self.clsd_link[x].setColorFrom( self.lut_all, self.dom[x].clsd_conns )
      self.attk_link[x].setColorFrom( self.lut_open, self.dom[x].open_conns )
      self.dom[x].setColorFrom( self.lut_domain )
      if self.stepsize.get()-i < 11:
        newmsg = newmsg + "Connection closed: client " + `conn` + "\n"

  # correct the range of the color maps

  while self.dom[x].open_conns > self.maxall or self.dom[x].clsd_conns > self.maxall:
    self.maxall = 2*self.maxall
    self.lut_all.SetRange( 0, self.maxall )
    self.lut_all.Build()
    self.lut_domain.SetRange( 0, self.maxall )
    self.lut_domain.Build()
    for i in range( 6 ):
      self.open_link[i].setColorFrom( self.lut_all, self.dom[i].open_conns )
      self.clsd_link[i].setColorFrom( self.lut_all, self.dom[i].clsd_conns )
      self.dom[i].setColorFrom( self.lut_domain )
  while self.dom[x].open_conns > self.maxopen:
    self.maxopen = 2*self.maxopen
    self.lut_open.SetRange( 0, self.maxopen )
    self.lut_open.Build()
    for i in range( 6 ):
      self.attk_link[i].setColorFrom( self.lut_open, self.dom[i].open_conns )

  if self.show_what.get() == 0:
    newmsg = "Connection Stats (open, closed):\n\n"
    for i in range( 6 ):
      newmsg = newmsg+"Domain "+`i`+":  "+`self.dom[i].open_conns`
      newmsg = newmsg+", "+`self.dom[i].clsd_conns`+"\n"

  self.msg.set( newmsg )

  num = max = 0
  for i in range( 6 ):
    if self.dom[i].open_conns > max:
      max = self.dom[i].open_conns
      num = i

  self.updateServerStatus()
  self.updateStatusLbl()
  self.win.Render()

  if self.animOn == 0:
    c = self.dom[num].getColor()
    self.dom[num].setColor( [ 1, .5, .5 ] )
    self.win.Render()
    sleep( .25 )
    self.dom[num].setColor( c )
    self.win.Render()

  else:
    self.after( 501-self.speed.get(), (lambda s=self: s.nextEvents()))


def updateServerStatus( self ):

  sum = most = 0
  for i in range( 6 ):
    sum = sum + self.dom[i].open_conns
    if self.dom[i].open_conns > self.dom[most].open_conns:
      most = i

  if sum < self.threshold.get():  self.servAct.GetProperty().SetColor( 1, 1, 1 )
  else:  self.servAct.GetProperty().SetColor( 1, .5, .5 )


def showServerInfo( self ):

  open_total = clsd_total = p_sent = p_rcvd = 0

  for i in range( 6 ):
    open_total = open_total + self.dom[i].open_conns
    clsd_total = clsd_total + self.dom[i].clsd_conns

  p_sent = open_total + clsd_total
  p_rcvd = open_total + ( clsd_total * 2 )

  msg = "Open connections: " + `open_total` + "\n"
  msg = msg + "Closed connections: " + `clsd_total` + "\n\n"
  msg = msg + "Packets sent: " + `p_sent` + "\n"
  msg = msg + "Packets received: " + `p_rcvd`

  tkMessageBox.showinfo( "Server information", msg )


def actorPicked( self, actor ):
  if actor == self.servAct:
    self.showServerInfo()
    return

  for i in range( 6 ):
    if actor == self.dom[i].getSphere():
      self.dom[i].openWin()
      break


def showMostConns( self, actor ):

  for i in range( 6 ):
    if actor == self.dom[i].getSphere():
      self.dom[i].showMostConns( None )
      break


if __name__ == '__main__':  NetView().mainloop()
```

DOMAIN.PY

```
#!/usr/bin/python

from vtkpython import *
from Tkinter import *
from subdomain import Subdomain
from newRenderWidget import *
from whrandom import *
import tkMessageBox


class Domain:


  def __init__( self, cb, n, sd, srcs, start ):

    self.callback = cb
    self.num = n
    self.open_conns = 0
    self.clsd_conns = 0
    self.first = start
    self.num_subs = int( math.ceil( srcs / sd ))
    self.subdom = range( self.num_subs )
    self.old_color = [ 0, 0, 0 ]

    for i in range( self.num_subs-1 ):
      self.subdom[i] = Subdomain( self, i, n, self.num_subs, start+( i*self.num_subs ))

    self.subdom[self.num_subs-1] = Subdomain( self, sd-1, n, srcs-((sd-1)*self.num_subs ),
start+(sd-1)*self.num_subs )

    #-----------------------
    # Build subdomain window

    self.visible = 0
    self.ren = vtkRenderer()
    self.window = Toplevel()
    self.window.withdraw()
    self.window.title( "Domain "+`self.num` )
    self.renwin = newRenderWidget( self.window, self, width=400, height=400 )
    self.renwin.GetRenderWindow().AddRenderer( self.ren )
    Button( self.window, width=50, text="Close", command=self.closeWin ).pack( side=BOTTOM )

    for i in range( self.num_subs ):

      a = i*2.*math.pi/self.num_subs
      self.subdom[i].setPosition( 3*math.cos( a ), 0, 3*math.sin( a ))
      self.ren.AddActor( self.subdom[i].getSphere())
    cam = self.ren.GetActiveCamera()
    cam.SetPosition( 0, 24, 0 )
    cam.SetViewUp( 0, 0, -1 )
    cam.Zoom( 1.6 )

    self.renwin.pack()

    #--------------------------------------------
    # Domain actor ( to be used in main window )

    self.src = vtkSphereSource()
    self.src.SetRadius( 3 )
    self.src.SetThetaResolution( 16 )
    texMap = vtkTextureMapToSphere()
    texMap.SetInput( self.src.GetOutput())
    s_map = vtkDataSetMapper()
    s_map.SetInput( texMap.GetOutput())
    reader = vtkBMPReader()
    reader.SetFileName( "textures/network.bmp" )
    txr = vtkTexture()
    txr.SetInput( reader.GetOutput())
    self.actor = vtkActor()
    self.actor.SetMapper( s_map )
    self.actor.SetTexture( txr )


  def reset( self ):

    self.open_conns = self.clsd_conns = 0
    self.actor.GetProperty().SetColor( .5, .5, .5 )
    for i in range( self.num_subs ):
      self.subdom[i].reset()


  def setPosition( self, p ):

    self.src.SetCenter( p[0], p[1], p[2] )


  def setColor( self, c ):

    self.actor.GetProperty().SetColor( c[0], c[1], c[2] )


  def setColorFrom( self, lut ):

    c = lut.GetColor( self.open_conns + self.clsd_conns )
    self.actor.GetProperty().SetColor( c[0], c[1], c[2] )


  def getColor( self ):   return self.actor.GetProperty().GetColor()
  def getSphere( self ):  return self.actor


  def addOpenConn( self, client ):

    self.open_conns = self.open_conns + 1
    sub = int(( client-self.first ) / self.num_subs )
    if sub >= len( self.subdom ):  return
    self.subdom[sub].addOpenConn( client )


  def addClsdConn( self, client ):

    self.open_conns = self.open_conns - 1
    self.clsd_conns = self.clsd_conns + 1
    sub = int(( client-self.first ) / self.num_subs )
    if sub >= len( self.subdom ):  return
    self.subdom[sub].addClsdConn( client )

  def openWin( self ):
    max = 0
    for i in range( len( self.subdom )):
      if self.subdom[i].sumOpen() > max:
        max = self.subdom[i].sumOpen()

    lut = vtkLookupTable()
    lut.SetHueRange( 0, 0 )
    lut.SetSaturationRange( 0, 0 )
    lut.SetValueRange( .2, 1 )
    lut.SetRange( 0, max+1 )
    lut.Build()

    for i in range( len( self.subdom )):
      self.subdom[i].setColorFrom( lut )

    self.visible = 1
    self.old_color = self.getColor()
    c = [ random(), random(), random() ]
    self.setColor( c )
    self.ren.SetBackground( c )
    self.callback.win.Render()
    self.window.deiconify()


  def closeWin( self ):

    self.visible = 0
    self.setColor( self.old_color )
    self.window.withdraw()
    self.callback.win.Render()


  def actorPicked( self, a ):

    for i in range( len( self.subdom )):
      if a == self.subdom[i].getSphere():
        self.subdom[i].openWin()
        break


  def showMostConns( self, a ):

    if a == None:
      max = 0
      sd = 0
      tmp = range( 2 )
      for i in range( len( self.subdom )):
        tmp = self.subdom[i].maxOpen()
        if tmp[0] > max:
          max = tmp[0]
          sd = i
      if max != 0:
        self.subdom[sd].showMostConns( None )
      else:
        tkMessageBox.showinfo( "QuickFind", "No open connections" )

    for i in range( len( self.subdom )):
      if a == self.subdom[i].getSphere():
        self.subdom[i].showMostConns( None )
        break
```

SUBDOMAIN.PY

```
#!/usr/bin/python

import tkMessageBox
from newRenderWidget import *
from Tkinter import *
from vtkpython import *
from whrandom import *
```

```
import math

class Subdomain:

  def __init__( self, cb, n, dn, srcs, start ):

    self.callback = cb
    self.domain_num = dn
    self.num = int( n )
    self.open_conns = range( srcs )
    self.clsd_conns = range( srcs )
    self.num_srcs = int( srcs )
    self.first = int( start )
    self.source = range( self.num_srcs )
    self.old_color =  [ 0, 0, 0 ]

    sphr = vtkSphereSource()
    texMap2 = vtkTextureMapToSphere()
    texMap2.SetInput( sphr.GetOutput())
    map2 = vtkDataSetMapper()
    map2.SetInput( texMap2.GetOutput())
    reader2 = vtkBMPReader()
    reader2.SetFileName( "textures/source.bmp" )
    txr2 = vtkTexture()
    txr2.SetInput( reader2.GetOutput())

    for i in range( self.num_srcs ):

      self.open_conns[i] = self.clsd_conns[i] = 0

      self.source[i] = vtkActor()
      self.source[i].SetMapper( map2 )
      self.source[i].SetTexture( txr2 )

    self.visible = 0
    self.ren = None
    self.window = None
    self.renwin = None

    # -----------------------------------------------
    # Subdomain actor ( to be used in domain window )

    texMap1 = vtkTextureMapToSphere()
    texMap1.SetInput( sphr.GetOutput())
    map1 = vtkDataSetMapper()
    map1.SetInput( texMap1.GetOutput())
    reader1 = vtkBMPReader()
    reader1.SetFileName( "textures/lan.bmp" )
    txr1 = vtkTexture()
    txr1.SetInput( reader1.GetOutput())
    self.sphere = vtkActor()
    self.sphere.SetMapper( map1 )
    self.sphere.SetTexture( txr1 )


  def setPosition( self, x, y, z ):  self.sphere.SetPosition( x, y, z )
  def getNumSrcs( self ):  return len( self.source )
  def getSphere( self ):   return self.sphere


  def reset( self ):

    for i in range( self.num_srcs ):
      self.open_conns[i] = self.clsd_conns[i] = 0


  def setColor( self, c ):

    self.sphere.GetProperty().SetColor( c[0], c[1], c[2] )
```

```
def setColorFrom( self, lut ):

  c = lut.GetColor( self.sumOpen())
  self.sphere.GetProperty().SetColor( c[0], c[1], c[2] )


def sumOpen( self ):

  sum = 0
  for i in range( self.num_srcs ):
    sum = sum + self.open_conns[i]
  return sum


def maxOpen( self ):

  max = range( 2 )
  max[0] = 0
  max[1] = 0
  for i in range( self.num_srcs ):
    if self.open_conns[i] > max[0]:
      max[0] = self.open_conns[i]
      max[1] = i

  return max


def addOpenConn( self, client ):

  c = client-self.first
  if c > self.num_srcs or c < 0:  return
  self.open_conns[c] = self.open_conns[c] + 1


def addClsdConn( self, client ):

  c = client-self.first
  if c > self.num_srcs or c < 0:  return
  self.open_conns[c] = self.open_conns[c] - 1
  self.clsd_conns[c] = self.clsd_conns[c] + 1


def openWin( self ):

  self.ren = vtkRenderer()
  self.window = Toplevel()
  self.window.title( "Domain " + `self.domain_num` + " -- Subdomain " + `self.num` )
  self.renwin = newRenderWidget( self.window, self, width=400, height=400 )
  self.renwin.GetRenderWindow().AddRenderer( self.ren )
  Button( self.window, width=50, text="Close", command=self.closeWin ).pack( side=BOTTOM )

  for i in range( self.num_srcs ):

    a = i*2.*math.pi/self.num_srcs
    self.source[i].SetPosition( 3*math.cos( a ), 0, 3*math.sin( a ))
    self.ren.AddActor( self.source[i] )

  cam = self.ren.GetActiveCamera()
  cam.SetPosition( 0, 24, 0 )
  cam.SetFocalPoint( 0, 0, 0 )
  cam.SetViewUp( 0, 0, -1 )
  cam.SetClippingRange( 1, 100 )
  cam.Zoom( 1.6 )

  self.renwin.pack()

  lut = vtkLookupTable()
  lut.SetHueRange( 0, 0 )
  lut.SetSaturationRange( 0, 0 )
  lut.SetValueRange( .2, 1 )
  lut.SetRange( 0, self.maxOpen()[0]+1 )
```

```
    lut.Build()

    for i in range( self.num_srcs ):
      c = lut.GetColor( self.open_conns[i] )
      self.source[i].GetProperty().SetColor( c[0], c[1], c[2] )

    self.visible = 1
    self.old_color = self.sphere.GetProperty().GetColor()
    c = [ random(), random(), random() ]
    self.setColor( c )
    self.ren.SetBackground( c )
    self.callback.renwin.Render()
    self.window.deiconify()


  def closeWin( self ):

    self.visible = 0
    self.setColor( self.old_color )
    self.window.withdraw()
    self.callback.renwin.Render()


  def actorPicked( self, a ):

    for i in range( self.num_srcs ):
      if a == self.source[i]:
        msg = "Client address: " + `i+self.first` + "\n"
        msg = msg + "Open connections: " + `self.open_conns[i]`
        tkMessageBox.showinfo( "Client information", msg )
        break

  def showMostConns( self, a ):

    max = 0
    client = 0

    if a == None:
      for i in range( self.num_srcs ):
        if self.open_conns[i] > max:
          max = self.open_conns[i]
          client = i

    for i in range( self.num_srcs ):
      if a == self.source[i]:
        max = self.open_conns[i]
        client = i

    if max != 0:
      msg = "Client address: " + `client+self.first` + "\n"
      msg = msg + "Open connections: " + `max`
      tkMessageBox.showinfo( "QuickFind", msg )
    else:
      tkMessageBox.showinfo( "QuickFind", "No open connections" )
```

LINK.PY

```
#!usr/bin/python

from vtkpython import *

class Link:

  def __init__( self, num ):

    self.number = num

    self.src = vtkLineSource()
    tube = vtkTubeFilter()
    tube.SetInput( self.src.GetOutput())
    tube.SetRadius( .4 )
    tube.SetNumberOfSides( 12 )
    map = vtkPolyDataMapper()
    map.SetInput( tube.GetOutput())

    self.act = vtkActor()
    self.act.SetMapper( map )


  def setPoints( self, p1, p2 ):
    self.src.SetPoint1( p1[0], p1[1], p1[2] )
    self.src.SetPoint2( p2[0], p2[1], p2[2] )


  def setColor( self, c ):
    self.act.GetProperty().SetColor( c[0], c[1], c[2] )


  def setColorFrom( self, lut, num ):
    self.setColor(  lut.GetColor( num ))


  def getActor( self ):  return self.act
```

NETREAD.CXX

```
#include <stdlib.h>
#include <fstream.h>
#include <string.h>

// 0  - time
// 1  - source addr
// 2  - source port
// 3  - dest addr
// 4  - dest port
// 5  - flag
// 6  - seq num 1
// 7  - seq num 2
// 8  - ack
// 9  - win
// 10 - buf
// 11 - ulen
// 12 - op

enum Fields { TIME, SRC_ADDR, SRC_PORT, DEST_ADDR, DEST_PORT,
              FLAG, SEQ1, SEQ2, ACK, WIN, BUF, ULEN, OP };

class NetReader
{
  private:

    ifstream in;

    int open_conns[20000], num_open, line;

    char str[400], *ptr[13], *tmp;
    char filename[50];

  public:

    NetReader( const char* fn )
    {
      strcpy( filename, fn );
      reset();
    }

    int getNextEvent()
    {
      int conn;

      in.getline( str, 400 );
```

```
        while( !in.eof())
        {
          line++;
          tokenize();

          if( !strcmp( ptr[FLAG], "S" ))
            if( strcmp( ptr[ACK], "" ))
            // response packet -- ack field is occupied
            {
              conn = atoi( ptr[DEST_ADDR] );
              if( conn != 2 )
              // we don't want server-initiated connections
              {
                open_conns[num_open++] = conn;
                return -conn;   // client has opened connection
              }
            }

          if( !strcmp( ptr[FLAG], "." ))
            if( !strcmp( ptr[SEQ1], "" ) && !strcmp( ptr[ACK], "1" ))
            // final connection packet -- no seq. num's and ack = 1
            {
              conn = atoi( ptr[SRC_ADDR] );
              if( conn != 2 )
              // we don't want server-initiated connections
              {
                for( int i=0; i<num_open; i++ )
                  if( open_conns[i] == conn )
                  // remove client from open connection list
                  {
                    open_conns[i] = open_conns[--num_open];
                    return conn;   // client has closed connection
                  }
              }
            }

          in.getline( str, 400 );
        }

        return 0;   // end of file
      }

      void tokenize()    // split up the header; fields are pointed to by ptr[]
      {
        ptr[TIME] = str;

        for( int i=1; i<13; i++ )
        {
          tmp = strchr( ptr[i-1], ',' );
          *tmp = '\0';
          ptr[i] = tmp+1;
        }
      }

      void reset()
      {
        num_open = line = 1;
        in.close();
        in.open( filename );

        if( !in )
        {
          cout << "Error:  File \"" << filename << "\" not found.";
          cout << endl << "  Check path." << endl;
          return;
        }

        in.getline( str, 400 );
      }
  };
```