Investigating Wired and Wireless Networks Using a Java-based Programmable Sniffer

Michael J Jipping
Hope College
Department of Computer Science
Holland, MI 49423 USA
+1 616 395 7509
jipping@cs.hope.edu

Andrew Kalafut
Bradley University
Department of Computer Science
Peoria, IL 61625 USA
+1 309 677 3101
akalafut@cs1.bradley.edu

Nathan Kooistra
Hope College
Department of Computer Science
Holland, MI 49423 USA
+1 616 395 7509
kooistra@cs.hope.edu

Kathleen Ludewig
Hope College
Department of Computer Science
Holland, MI 49423 USA
+1 616 395 7509
ludewig@cs.hope.edu

ABSTRACT

Teaching students about networking requires laboratory investigation into network data. Such investigation requires examination of both wired and wireless network data. Most available network traffic sniffers are either too expensive or too cryptic to use. To implement network experiments in a classroom setting, we have developed NetSpy: a Java-based network sniffer that allows plug-in Java modules to analyze network data. NetSpy works with both wired and wireless networks. Modules are written by students as part of their experimentation with traffic data. This paper describes the NetSpy system and the way it can be used in a Networking class

Categories and Subject Descriptors

K.3 **[Computers & Education]:** Computer & Information Science Education - *Computer Science Education*.

General Terms

Design, Experimentation, Security, Human Factors.

Keywords

Network Concepts, Java, Network Sniffer, Pedagogy.

1. INTRODUCTION

In the computer science curriculum, the Networking course stands out by virtue of its requirements. The course requires that a lot of difficult material be delivered to students, typically via "hands-on" active learning activities. Active learning and experimentation in a Networking course is typically done by examining a network via its traffic and interpreting the applications and protocols that run across that network. This experimentation is increasingly involving both wired and wireless networks.

Unfortunately, tools that facilitate network experimentation are either expensive or difficult for students to use. Special purpose tools to sniff networks are quite expensive and do not allow general purpose programming to analyze network data. Tools available on Linux or Windows platforms generate cryptic output that is not easily deciphered or analyzed. In addition, wireless network data is sufficiently different from wired network data that tools often do not work on both platforms.

This paper describes a system designed to teach Networking concepts and analysis by using Java to program a platform that delivers both wired and wireless network data. We have developed a system called **NetSpy** that consists of a network sniffing platform coupled with a Java plug-in interface. Using this interface, students can write Java plug-in modules that can receive network data for analysis. NetSpy has been implemented for handheld computers, extending its usefulness and experimentation ability.

This paper will present some background on NetSpy, followed by an examination of its system components. We will consider examples of NetSpy Java programming and conclude with some discussion on NetSpy's use.

2. BACKGROUND AND MOTIVATION

A key to the Networking course is network data analysis. In our course, we would like to perform activities such as traffic measurement, observing and analyzing network protocols in detail, mapping protocol usage to patterns of application usage, and developing solutions to network problems.

ITiCSE '04, June 28-30, 2004, Leeds, United Kingdom.

However, network analysis is difficult to do, especially if students need to develop or learn all aspects of the analysis tools. If students program network snooping themselves, for example, it is very easy for them to get mired in the data structures and system interfaces needed to properly fetch a network packet and extract data that can be analyzed. Students should focus on the analysis itself, not the mechanics of traffic sniffing.

Very little software exists to assist a student in this endeavor. For a general purpose platform, there are indeed tools that will watch network traffic. Applications such as snoop on Solaris platforms and tcpdump [4] on Linux are very effective in fetching data traffic and displaying the contents of each network packet. Unfortunately, these tools display the data in a cryptic manner and are not at all programmable. Tools such as etherape [1,2] and ethereal [5] on Linux devices attempt to present network data in a format that is more easily analyzed and understood, but again do not allow for general analysis through custom designed programs.

We seek a platform for network analysis that provides the following functionality:

- *Network data retrieval*: The platform should accurately read network packets and present these packets to the user in a clean, usable manner.
- *Ability to filter network data*: The platform should be able to filter the network packet data to present only the specific type of data requested by the user.
- *Network data object orientation*: For intuitive analysis, the platform should present network data to the user in an object-oriented manner, with a clean, usable method interface.
- *Programmable network data analysis*: The platform should allow the user to use a wide range of custom-built, programmable analysis tools to examine network data.
- *Media-rich:* The platform should apply to both wired and wireless networks.

In addition, the ability to use a handheld computer platform would be very beneficial. Such a feature would enable students to sample data from sources all over campus - not just from a single computer laboratory or classroom.

3. THE NETSPY SYSTEM

NetSpy is a network data gathering system written primarily in Java. The packet capturing code is written in C++, and makes use of the PCAP library [3] for some functions. The Java Native Interface (JNI) serves as the connector between the C++ packet capturing code and the Java data presentation code of NetSpy. NetSpy runs on both desktop and handheld Linux systems, including the Sharp Zaurus SL-5500, and can be used on both Ethernet and wireless 802.11 networks.

NetSpy allows users to define simple *filters*, which allow only certain packets through based on packet length, protocol, type, source or destination hardware address, source or destination IP address, or port number. These filters can be combined using logical operators to form more complex filters. NetSpy also allows users to write plug-in modules in Java, called *operations*, to process captured packets. The user can then create an *agent*, consisting of the combination of an operation and a filter, which NetSpy uses to process data it obtains from the network.

3.1 The NetSpy Foundation

NetSpy uses both Java and C++ for portability and for the ability to capture network data. The portability of Java allows for the creation of user interface code that works on both the Zaurus and on desktop Linux platforms. The C++ code enables NetSpy to use PCAP for much of its packet capturing.

PCAP is an open source packet capture library that allows a system to easily fetch packets from a network. Packets are obtained from PCAP as byte arrays, which contain all the packet information received from the network card. PCAP does not do any processing of the packet data itself, but passes the arrays of packet data on to other parts of the program for processing. NetSpy allows the user to choose between three network types which determine its behavior in capturing packets. Two of these types, "Ethernet" and "Standard Wireless," rely on the PCAP library for their packet capturing.

The third network type does not use PCAP. We used the Sharp Zaurus SL-5500 with ROM versions 2.8 and 3.1 as our implementation platform. The versions of the PrismII network card utilities and drivers that are included on the Zaurus are out of date and cannot easily be updated. PCAP cannot work with these old versions, and thus does not work properly on the Zaurus. In response to this restriction, it was necessary to write packet capturing code that did not rely on PCAP in order to implement wireless packet capturing on the Zaurus SL-5500. The third network type in NetSpy, "PrismII Wireless", uses this code. The code for PrismII cards captures packets and stores the information from them as byte arrays, so the data looks the same as if it were captured using PCAP. This data is then passed onto the rest of the program and processed in the same manner as data obtained from PCAP.

NetSpy makes heavy use of JNI. JNI is a programming interface that allows Java to interact with another language, in this case C++. Once each packet has been captured using the C++ code, it is put into a format usable by the Java code. The packet data is then transferred to the Java code through JNI and instantiated as an object of the NetSpy `Packet` class in Java. In addition, JNI is used each time a packet is converted to its appropriate packet type. For example, if a regular Packet is passed to the Ethernet class, the Ethernet header information is extracted in C++, and the Java class is then populated with that information using JNI. JNI was required because low-level packet capturing is not possible in Java. It is also much easier to extract packet header information, in which some fields can be individual bits, in C++ than in Java.

After a packet is retrieved, it is passed to all agents that have specified filters that accept it. The first thing most agents should do is cast the packet as either an Ethernet or a wireless packet using NetSpy's `Ethernet` or `Wireless` classes. The `Ethernet` class extracts the Ethernet header data from the packet, allowing this data to be used in other parts of the program, such as for filtering, or in an operation. All of the header information is made available to use in operations through functions in the `Ethernet` class. The `Ethernet` class also provides a function to get the remaining packet without the Ethernet header as a byte array. Once the Ethernet packet is created, upper level packets, such as an IP packet, can be constructed from it. Below is the interface for the Ethernet class:

```
public class Ethernet {
   public Ethernet(Packet p);
   public boolean isEthernet();
   public byte[] getDestinationAddress();
   public byte[] getSourceAddress();
   public short getType();
   public byte[] getDataBytes();
}
```

Similar to the Ethernet implementation, the `Wireless` class parses information such as protocol type, destination port, source port, destination address and source address, and provides functions to access this information. However, this task is more complicated for wireless data than for Ethernet data. Wireless packets are divided into three types: data, management, and control. Each type contains different fields in its header information. The Wireless class parses the packets and determines which information will be present in the packet, constructing the header information accordingly. All fields not present in the header of a specific wireless packet are set to null. If a packet is a data packet, a function is present to get the packet's data excluding the 802.11 header as a byte array. If a wireless packet is a management frame, the packet's sub-type is used to determine what type of management frame it is, and then its body is processed to get the information specific to that type of management frame. For example, only wireless management packets that are beacon frames and probe responses contain the current channel on which the network is listening. Functions are then provided through the classes for each management frame type to access the information specific to that type of management frame. The following function from the `Wireless` class demonstrates differences between packet types that are management frames and how they are handled.

```
public Wireless(Packet p){
  //parse the Packet header here
  if (_type == 0) {
     switch (_subtype) {
     case 0: //Association Request
         mf = new WMT_AssocRequest(this);
         break;
      case 1: //Association Response
      case 3: //Reassociation  Response
          mf= new WMT_AssocResponse(this);
         break;
      case 2: //Reassociation Request
         mf = new WMT_ReassocRequest(this);
         break;
      case 4: //Probe Request
         mf = new WMT_ProbeRequest(this);
         break;
       case 5: //Probe Response
        mf = new WMT_ProbeResponse(this);
         break;
       case 8: //Beacon
         mf = new WMT_BeaconFrame(this);
         break;
       case 12: //Deauthentication
       case 10: //Disassociation
         mf = new WMT_Disassociation(this);
         break;
       case 11: //Authentication
         mf = new WMT_Authentication(this);
         break;
    }
  }
}
```

3.2 Network Data Retrieval Components

In order to retrieve information from the packets obtained via NetSpy, the operation plug-in is used. Each operation is a Java class that extends NetSpy's Operation class. The Operation class is used in this way to make it simple to write plug-ins for NetSpy, and to hide the details of thread management from the user. The only function in the Operation class that each operation needs to override is the *task()* function. When NetSpy is run, it creates an instance of the AgentMap class, which keeps track of every filter that is defined and every active operation. It also monitors which filters and operations are associated with each agent. As the AgentMap receives packets, it passes them to each filter that is associated with an active agent. If the packet passes the filter, it is put on the PacketQueue for the appropriate operation. When an agent is started, a thread is created which executes the operation by repeatedly calling its *task()* function. The task function of each operation should take a packet from its PacketQueue, and then process the packet in Java, using functions provided in the Java or NetSpy API. In this way, users can extend NetSpy and use it to analyze network data by overriding only one function and without having to understand how the underlying packet retrieval code works.

NetSpy begins its data collection process by extracting a packet from the network. This raw packet is passed to the Network class in Java using JNI. The Network class then uses this data to construct an object of our Packet class. Next, it places this Packet object onto the PacketQueue to wait until the AgentMap is ready to process it further. When the AgentMap's thread is ready, the Dispatcher takes if off of the PacketQueue and passes it to the AgentMap, which then processes it through each individual agent. This process is depicted in Figure 1.

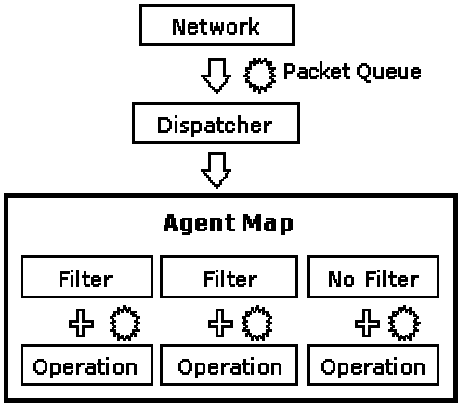

Figure 1: Network Packet Processing

3.3 Using NetSpy

NetSpy interacts with users through its graphical user interface. Its simplicity and small size are ideal for working on handheld computers as well as desktop Linux machines. The GUI allows the user to manipulate many of NetSpy's features.

The user interface of NetSpy, as shown in Figure 2, contains three menus: Filter, Agent, and Options, as well as five buttons: Start, Save, Load, Help, and Quit. The Filter and Agent menus contain all the functions of Filter and Agent, respectively. The Options menu manipulates the network settings. The Start button (which alternates as the Stop button) controls the packet flow on the network. The Save button enables the user to save the current

filter and agent files. Similarly, the Load button enables the user to load previously saved filter and agent files. The Help button brings up information about how to use each component within NetSpy. To conserve space, the area between the menus and the buttons is used for the main content area.

NetSpy provides several manipulations of filters. The user can define a filter, delete a filter, create composite filters of two or more filters, and view the properties of each filter.

Agents also have many functions. It can be created, deleted, changed, and started and stopped (i.e. activated and deactivated). Just as with filters, the properties of each agent can be viewed.

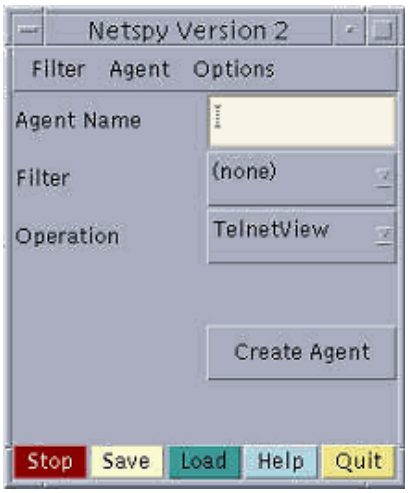

Figure 2: The NetSpy Main Interface

The Options menu configures the network settings. "Network Type" sets the network to Ethernet, Standard Wireless, or Prism II Wireless. "Channel Preferences" enables the user to listen to a specific channel or to rotate through channels when listening on a wireless network.

The following is a typical example of how a user would run NetSpy on the Zaurus using a PrismII-based wireless card. First, the user would run NetSpy to bring up its user interface. The first time NetSpy is run, the user should go to the Options menu and choose "Network Type", and change the type to PrismII Wireless. The user would then choose "Channel Prefs" from the Options menu, and select the channel on which to listen. If this was not the first time running NetSpy on the Zaurus, NetSpy would remember the last settings used, making the previous step unnecessary. The user would then go to "Define" under the Filter menu, and use the interface provided to create any simple filters needed. Then the user would go to "Define Composite Filter" and use this to combine the already defined simple filters into more complex filters. After this, the user should go to "Create Agent" and choose the desired filter and operation for the agent. Once the agent is created, the user should click the Save button (so that filters and agents may be reused the next time NetSpy is started) , and then press the Start button to start the agent running.

4. USING NETSPY TO COMPARE WIRED AND WIRELESS NETWORKS

The main advantage of the NetSpy system is found in its ability to create custom plug-ins by using both NetSpy's and Java's API. By creating a simple interface for plug-ins, NetSpy's expandability is vast. This interface is easy enough to understand for students beginning a study of networking, yet versatile enough to be used by more advanced programmers.

In a wired setting, a variety of plug-ins may be created. For example, it is possible to analyze and categorize data types, detect and monitor network intruders, and reconstruct or analyze transferred information, to name only a few uses. In particular, the following example demonstrates how a user could extract the source of each TCP packet traveling across the network.

```
short port;
PacketQueue pq;
Packet p = pq.dequeue();
Ethernet e = new Ethernet(p);
if (e.getType() ==
       Ethernet.ETHERTYPE_IP) {
    IP ip = new IP(e);
    if (ip.getProtocol() ==
       IP.IPPROTO_TCP) {
        TCP tcp = new TCP(ip);
        port = tcp.getDestinationPort();
    }
 }
```

Similar plug-ins can also be created for wireless networks. The added number of packet types creates opportunities for further packet analysis, such as classifying specific management packets. The example below illustrates how a user could extract the channel over which data is being sent (a nonexistent feature over Ethernet).

```
PacketQueue pq;
Packet p = pq.dequeue();
Wireless wp = new Wireless(p);
if (wp.getType() ==
     Wireless.MANAGEMENT) {
    WirelessMgmtTag mgmt =
                wp.getManagementBody();
    if (mgmt.getType() ==
           Wireless.MGMT_PROBERESP) {
       WMT_ProbeResponse probeRes =
                new WMT_ProbeResponse(wp);
       byte channel =
               probeRes.currentChannel();
    }
 }
```

Regardless of whether a system is listening on a wired or a wireless network, NetSpy constructs IP and TCP packets in a similar fashion. In fact, once a packet has been constructed into either the Wireless or Ethernet classes, the resulting object can be treated in the same manner. The only difference between a wired and wireless data packet is the information found in the packet's header. The following example shows that while the transfer medium is different, TCP/IP data exists and is the same in both Ethernet and wireless packets.

```
PacketQueue pq;
Packet p = pq.dequeue();
Ethernet ep = null;
Wireless wp = null;
IP ip = null;

//listening on an Ethernet network
if(p.getPacketType()==1) {
  ep = new Ethernet(p);
  if(ep.getType()==Ethernet.ETHERTYPE_IP)
```

```
    ip = new IP(ep);
} else

    //listening on a wireless network
    if(p.getPacketType() == 119 ||
      p.getPacketType() == 105) {
      wp = new Wireless(p);
      if(wp.getType()==Wireless.DATA) {
         LLCHeader_802dot11 llc =
          new LLCHeader_802dot11(wp);
         if (llc.getType() ==
           LLCHeader_802dot11.LLCTYPE_IP)
         ip = new IP(wp);
      }
   }
if (ip.getProtocol() ==
                    IP.IPPROTO_TCP) {
   TCP tcp = new TCP(ip);
}
```

Using NetSpy's capability to scan through the eleven possible wireless channels (in 802.11b), a user has the ability to sample data from a variety of networks in a single area. Such an ability opens up more opportunities for the plug-in expansion of NetSpy. It is possible to create a network sniffing tool that will list all the wireless networks in a given area. Not only is it able to list the networks, but using NetSpy's information retrieval methods, specific information can be obtained from packets captured. A network's SSID, BSSID, channel, packet count according to type, speed, WEP, and IBSS capabilities can be discovered by looking at a packet's header information that NetSpy receives and records.

An example of such a plug-in the Wireless Network Detector (WiND), whose interface is shown in Figure 3.

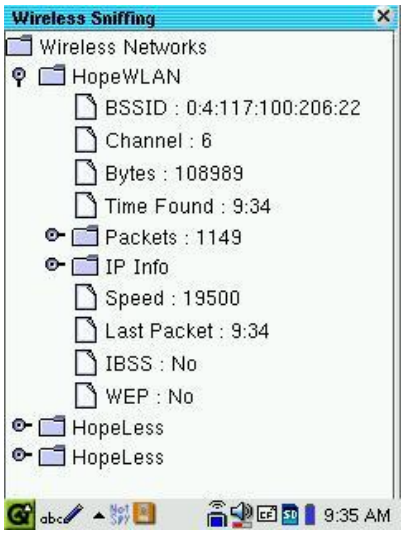

Figure 3: The WiND User Interface

5. NETSPY PROJECTS

There are several projects that NetSpy would work well for. The straightforward GUI and the ability to create custom operations provides NetSpy with much more flexibility than other network sniffing programs. It has many unique possibilities that others do not. There are several ways that we have and will be using NetSpy in a classroom setting.

Network analysis and experimentation: Through NetSpy, a user has access to a complete set of network data - all layers in the OSI network model stack. This means that protocol analysis and statistics calculation are possible. This makes projects from packet counting to protocol reconstruction (e.g., rebuilding Telnet sessions) relatively simple to build.

Network data sampling: For this exercise, students use a protocol counting module - one that simply writes numbers to a log file - and uses a handheld computer to do *protocol walking*. They sample data from all over campus, noting the different types of network applications in different areas. In their writeups and subsequent discussion, we examine differences and make predictions of network requirements based on type of usage and time of day.

Network administration: There are many network administration tools that claim to detect certain network conditions - like traffic congestion or uninvited intruders. It is interesting to see how these tools work and NetSpy can be used to detect the conditions that these administration tools do. Many of these tools are open-source, which allows students to duplicate some of their functionality in a NetSpy module.

Application debugging: Many network-based applications are difficult to write because the protocol exchanges are hard to watch and debug. Since filtering is built into NetSpy, modules can be written to track and interpret specific network protocols connected with certain applications.

6. CONCLUSION

NetSpy is an effective tool for allowing students to do protocol analysis and network experimentation. It allows users to write their own analysis tools and protocol collectors. In doing so, students are able to understand and analyze both wired and wireless network data.

For the future, we wish to build classroom components that depend on NetSpy. Laboratory manuals that use NetSpy would have users write small Java operations to experiment with the network. NetSpy's sniffing abilities could also be extended to more media, for example, GSM telephony.

References

[1] Cota, J.T., *Implementicion de un Monitor Analizador Grafico de Reden el Entorno Gnome*, Final Report on the Etherape Project, University of Seville, Spain, July 2001.

[2] *Etherape Software Repository*, available online: http://etherape.sourceforge.net.

[3] *PCAP Public Repository,* available online: http://www-nrg.ee.lbl.gov.

[4] *TCPDUMP Public Repository*, available online: http://www.tcpdump.org.

[5] *The Ethereal Network Analyzer*, available online: http://www.zing.org.