User Guide

Welcome to jNetPcap SDK. jNetPcap is a java software development kit for network administrators, protocol developers, network software engineers and university level students. The basic function of jNetPcap is to provide a java wrapper to popular libpcap library for capturing network packets. The SDK also provides additional capabilities not found in the original native libpcap project.

jNetPcap project at this time, does not provide any pre built applications such as packet sniffer, security monitors, firewalls or any other applications that libpcap is typically used for. It is a pure library that makes the job of creating such applications much easier.

The additional capabilities are nicely incorporated, but are not essential for the proper usage of libpcap API. The project is made up of a java library, a native shared library (a .dll on win32 systems and .so file on unix based systems.) This library provides the interface (JNI bridge) between java world and native platform software and libpcap library itself. The software also requires a separate installation of the libpcap software on both unix and win32 systems. Please see Chapter 1 for more information on installation issues ([still under construction].)

To get started you may want to start with Chapter 1 which goes through all the basics. Chapter 2 goes over the core libpcap wrapper API and its usage. Chapter 3 discusses the "Protocol Decoding Framework", an extension that is not part of libpcap library, but a jNetPcap additional functionality. Specifically jNetPcap SDK provides sophisticated packet decoding capabilities and many supplied protocols. Chapter 4 goes into more implementation detail and discusses many important issues that every programmer needs to address and plan for in hers or his project. Performance and resource usage are essential in choosing the right architecture for your application.

Do not hesitate to ask questions at either project website's forum or as comments on any of these book pages of this user guide.

Here is the link to project website, if you are not reading this page from the original webserver: http://jnetpcap.com

Ch 1 - The Basics

There are two parts to jNetPcap SDK. The first part is the libcap wrapper, which provides nearly all of the functionality of libpcap native library, in a java environment. This wrapper, just like libpcap itself, is lowest level API in jNetPcap SDK.

The second part is the packet decoding framework. This is a collection of packages that allows a captured packet to be processed and decoded. The programmer then uses a set of java classes called protocol headers, to work with these decoded packets.

The libpcap wrapper is not dependent on the packet framework and can work completely without invoking any of the packet framework api. The packet decoding framework is also independent of the libpcap wrapper. It provides methods that can scan and decode any packet buffer in memory, not just the ones delivered from the wrapper.

Of course, the two API are linked seemlessly and you can ask libpcap wrapper to deliver fully decoded packets to you. This approach provides the best of both worlds and lets you work in any mode you choose.

1.1 - Getting Around

Let briefly go over all of the packages in jNetPcap SDK and what they contain:

```
org
  +-> jnetcap - this is the main libpcap wrapper package. It contains
      +         all of the API for accessing libpcap functionality.
      |         These classes and methods do very little of their own
      |         logic and simply pass your requests over to native
      |         libpcap which handles those the requested actions.
      |
      +-> winpcap - extension to libpcap wrapper that provides WinPcap
      |          functions. This is operating system dependent package
      |          and you must use WinPcap.isSupported() call before
      |          using any classes and methods in this package.
      |
      +-> nio - native IO and memory classes. This package defines
      |         memory management classes that allocate native memory,
      |         peer native structures and functions to java classes.
      |         This is also where the very important JBuffer class resides.
      |
      +-> util - various utility classes. This is where you will find
      |          logging helpers, JConfig class which manages configurations
      |          through property files and address resolvers.
      |
      +-> protocol - a library of supported CORE protocol headers.
      |              This is where you will find Ip4, Tcp, Udp, Ethernet
      |              and a host of other header definitions, ready for you
      |              to use.
      |
      +-> packet - packet decoding framework. This package defines important
          |      components of the decoder. JScanner, PcapPacket, and the very
          |      important baseclass JHeader. JScanner decodes packets
          |      and stores the packet state information in native structures.
          |      PcapPacket class reads this state information and can
          |      peer (or reference via native memory) header objects.
          |      All header classes subclass JHeader class.
          |
           +-> format - packet formatters. They dump the decoded packet
          |      content in textual form. You can use TextFormatter or XmlFormatter.
          |
          +-> structure - this package is utilized by protocol builders.
          |      It contains building blocks of decoded packet that
          |      formatters use. This is a package you will not normally
          |      have to deal with.
          |
          +-> annotate - annotation interfaces used in writting protocol headers.
                 Unless you are writting a new protocol header definiton
                 you will not need access any of these annotations.
```

1.2 - Configuration through properties

All parts of SDK are custimizable through configuration files. These are almost normal java's properties files, at least their structure is. These property files define various parameters and properties that are use by SDK to configure its behaviour.

The SDK provides all neccessary properties directly in the JAR file. They are called "builtin" properties. You will find them in the classpath in "resources" directory. There are currently two properties files:

- builtin-config.properties - provide general SDK configuration

- builtin-logger.properties - provide logging facility configuration

Both of these default builtin configuration can be easily overriden by placing a new configurarion file anywhere in the SDK's search path (described below.) The user supplied properties are combined with the builtin ones, overriding any properties that conflict with the user properties taking the precence.

As a quick example, if you place a file alled "config.properties" or "logger.properties" in you current working directory, when you start up jNetPcap SDK, that file will be found and loaded, overriding any builtin properties that were defined. You can also specify any properties you like on the command line when starting the java VM with the usual "-D" command line option. The precedence of properties is as follows:

1. System properties - defined on the command line
2. user properties - defined in a user properties file
3. builtin properites - default properties supplied with the SDK.

Properties file pre-processor

There is a pre-processor applied to properties files, that recombines property lines that have been escaped with a backslash-newline at the end of a property line. This pre processor is applied directly on the IO stream when the file is being read by a normal java Properies class. This pre processor then allows properties to be defined on multiple consecutive lines escaped with backslash-newline escape sequence. Here is an example

```
property1 = value 1 \
  value 2 \
  value 3
propert2 = value 4
```

After the pre processor, the file would like like this:

```
property1 = value 1 value 2 value 3
property2 = value 4
```

Otherwise, they are completely normal Property files.

Expandable properties

Properties are retrieved from the properties file using normal `Properties.getProperty` and `Properties.setProperty` accessor methods. The values can be post processed by SDK to provide additional functionality. This is where `ExpandableString` and `ConfigString` come in. These are fancy strings that allow replacement of certain special variables and properties that are defined within them. They lookup the values of the properties and variables that are embeded in the string using supplied SDK's configuartion environment. For example:

```
property1 = hello
property2 = @{property1}
```

These 2 properties, when expanded, will both contain the value of "hello". Here is a code sample:

```
String property1 = JConfig.getProperty("property1");
String property2 = JConfig.getExpandedProperty("property2");

System.out.println("property1=%s and property2=%s\n", property1, property2);

// Output produced is
// property1=hello and property2=hello
```

Properties can reference other properties that reference other properties. The expansion happens recursively until the string can no longer be expanded or until a reference to another property is invalid. Either way the expansion stops.

This is a powerful way for properties to inherit their values from other, possibly more global properties, while providing the flexibility of smaller configuration units.

Search Paths

The real power of expandable properties is brought to fruition in search paths. A search path, is a property that defines SearchPath expressions for various kinds of search spaces. There are currently 3 kinds of search types possible:

- File - performs a search of the local filesystem
- Classpath - perfroms a search within a CLASSPATH
- URL - uses a URL to locate a resource

A search path is a normal property that defines a combination of these 3 types of search elements. You can use each element any number of times with a search path and you can reference other search path properties. All of their values will be combined. Here is an example of the main search path the SDK uses to search for resources such as config files:

```
search.path = \
    'File(@{user.home}/@{resources.subdir}/${name})' \
    'File(@{${name}})' \
    'File(@{user.dir}/${name})' \
    'File(@{java.io.tmpdir}/${name})' \
    'Classpath(resources/${name})' \
    'Classpath(${name})' \
    'URL(@{resources.${name}.url})'
```

Lets start by describing what is ${name}. Its a variable supplied at runtime and is usually the target of the search. So if we were seearching for "config.properties" file ${name} would expand to "config.properties". @{user.home} property is a standard system property inherited from System.getProperties() call. So is @{usr.dir} which is the current working directory.

The search path has 4 File search elements, 2 Classpath elements and 1 URL based search element. They are searched in order they appear in the declaration. Lets take a look at the first entry and substitute some concrete values. Assume @{user.home} = /home/guest, @{resources.subdir} = .jnp and we are looking for file named "oui.txt". The first line would expand to:

```
'File(/home/guest/.jnp/oui.txt)'
```

Very clean result for such a complex looking line. The next ones are even cleaner:

```
search.path = \
    'File(/home/guest/.jnp/oui.txt)' \
    'File(oui.txt)' \
    'File(./oui.txt)' \
    'File(/tmp/oui.txt)' \
    'Classpath(resources/oui.txt)' \
    'Classpath(oui.txt)' \
    'URL(@{resources.oui.txt.url})'
```

Notice that last entry @{resources.oui.txt.url}, was a little special. That is we had a reference to a variable ${} within the a property reference. Variables are always expanded first, then the properties are expanded on the result of the first variable expansion. In our case, we don't know what @{resources.oui.txt.url} property is, its not defined, therefore that expansion fails to resolve completely. It will be ignored. The remainder expanded just fine and appropriate search path algorithms are used to look for that resource. There are other search paths defined in the builtin config used to locate various resources, temp files, home direcotry for jNetPcap files, etc.

Its also worth noting that these search paths and any properties for that matter can inherit other property values, no matter how complex. Best example is the address resolvers. Address resolvers resolve a binary address to human label, such as ip address to hostname. The mapped result is cached in a cache file, which is stored on the local disk. Resolver's use properties to define numerous properties including a search path that they use to search for cache files. Instead of rewritting a search path from scratch, you can provide a reference to already standard search path like this:

```
resolver.search.path = @{search.path}
```

And the entire search path we just discussed, will be inherited by this new property. The user can override the search path in a user config file by simply defining a new value to property "resolver.search.path", or leave things at default using search path. If the user overrides the default "search.path" property any other properties that reference the global default search path will also be changed all at the same time.

1.2.1 - Using JConfig class

JConfig is the main configuration class, that reads in properties and uses search paths to find resources. It also configures the logging system from user and builtin logger properties.

When JConfig is first initialized by the JRE, it uses a special sequence of steps to initialize the SDK and envionment. The initialization steps are as follows:

1. Create a basic, barely initialized logger so that error messages at this very importat startup sequence can be reported.
2. Export some global builtin variables into a variable table
3. Load builtin, user and system properties and combine them using a special CompositeProperties class. The CompositeProperities class, allows 3 different Properties objects to be queried and accessed for each getProperty request. This class provides properties inherinance.
4. Export more variables that may have been defined in configuartion properties. Any property that begins with a prefix of "var." is automatically exported as a variable.
5. Now look for builtin and user supplied logger.properties files. Combine the using CompositeProperties class and reinitialize LogManager. LogManager will reread the combined properties.
6. Setup PropertyChangeSupport for dispatching property changes to listeners.

After initialization the logger and SDK configuration properties are loaded. JConfig is not ready for property change listeners to register themselves with JConfig for any property changes. During registration, each listener is immediately notified of the config values, overriding any default internal values they may have had.

At this point JConfig is up and running.

1.2.2 - Using ConfigString class

The power to expand property values comes from ConfigString class. This is a class that binds itself with JConfig properties and variable table and can recursively expand any property or variable references within the string.

ConfigString class comes with a public constructor so anyone can create an instance of one. You can use JConfig.getTopProperties() and JConfig.getGlobalVariables() calls to acquire the JConfig configuration tables, or as a convenience you can call JConfig.createConfigString(). The result will be a ConfigString object already property intialized.

Once you have a ConfigString object, you can call on its ConfigString.expand(String name) method. Where the parameter name is a runtime variable that is expanded for ${name} references within the original string. If name is known not to be used within the string, it is customary to provide an empty double quote string literal "" for that parameter.

ConfigString.expand() returns a boolean value. True means that string was expanded fully, or false if a unresolved reference was encountered. The unresolved reference can be in the immediate source string or any of the sub properties it referenced, may levels deep. In any case, an unresolved property reference or variable usually means that the value of the string is useless.

ConfigString objects can also have single quoted strings within. Any single quoted strings will be escaped from expansion algorithm.

The original string that was supplied to the ConfigString construtor is set as a template string, untouched through out the expansion process. ConfigString.reset() method will restore a ConfigString to its initial state, with the template string being used as the basis for the next expansion. This is a good way to repeach the expansion process under new set of properties or variables.

1.2.3 - The Defaults

Lets take a look at some of the properties that are defined in the build-config.properties and what they control. You can always override any of these properties by either supplying them on the command line with "-D" option, or in a config.properties file placed in the SDK's search path.

Here is a link to the entire builtin-configure.properties file.

Currently properties are only used for configuration, logging and resolvers. Eventually they will control nearly everything in jNetPcap SDK. (Resolvers is a new package that is used to resolve addresses to hostnames or human readable labels.)

User config and default search path

The builtin-config.properties file defines search path that is used to locate the user defined config.properties file. It also defines the name of the file its looking for by default to be "config.properties"

- config.name - defines the name of the user defined configuration file "config.properties" by default.
- config.bootstrap.search.path - defines the search path that JConfig class uses to locate the user configuration file.
  - in @{config.file} - this property is not set by default. It allows a user configuration file to be specified by defining this property as absolute path to a property file.
  - in directory defined by System property @{user.dir} which is the current working directory as well.
  - in @{config.dir} that is undefined by default, but can be set to user specific configuration directory containing this an possible many other config files.
  - @{home.dir} - this is jNetPcap home directory. By default this is a ".jnp" subdirectory in user's normal home directory.
  - In classpath under resources directory and under ./ classpath directory.

The config.bootstrap.search.path looks in lots of different places for user configuration file. It search each of those areas for a file named config.properties. If that file is found, it is merged with the builtin-config.properties, allowing user properties override the builtin values.

jNetPcap home directory

jNetPcap needs a place to store files such as cache files and possibly user configurations. All the search paths default to looking in various places and especially the the @{home.dir} property. This property is a compound property:

subdir = .jnp
home.dir = @{user.home}/@{subdir}

Where @{user.home} property is a java system property that points at user's home directory and @{subdir} is defined to be ".jnp". This directory won't exist by default and either the user can create it or you can allow the SDK to do it for you when its needed the first time. The property @{home.mkdir} control this permission:

home.dir = false

The default is false, which denies SDK from creating this directory. If you change this property to true, the directory will be created automatically, but only when its actually needed.

To change the name of the jNetPcap sub directory, simply define a different value for @{subdir} property.

subdir = jnetpcap

1.3 - Logging facility

jNetPcap SDK uses the normal JRE logging facility found in package "java.util.logging" for logging messages.

The logging facility is initialized through both builtin and user supplied logger properties. It can be configured according to the user's requirements.

1.4 - Installation

(Note: this is temporary document with some basic information. More comprehensive document is under development)

jNetPcap software is distributed as a installable package on most operating systems. The only prerequisite is that "libpcap" library or WinPcap on windows systems, be pre-installed prior to installing jnetpcap.

The distribution package installs the supplied "shared" library and jNetPcap development jar file into appropriate directories for each platform. On Unix systems this is typically "/usr/lib" and "/usr/share/java" directories.

On windows, the distribution package is a ZIP file. You need to unzip the contents of this file to a working folder. Both the supplied ".dll" file and the jNetPcap jar file need to be reachable by the applications which use it. The '.dll' file can be either copied into \windows\system** folder or added to the PATH variable in "autoexec.bat". The ".dll" file can also be specified to the java VM during startup using "-Djava.library.path=" option.

On all platforms, the supplied jar file, needs to be added to CLASSPATH or build environment variables.

1.5 - Compiling Source

(Note: this is temporary document with some basic information. More comprehensive document is under development)

jNetPcap library is composed of 2 components. The shared native library written in C++ language and the java code that is bound to it using java's JNI extensions.

Acquiring source code

Source code can be acquired from a downloaded distribution package or from SVN repository on SourceForge.net servers.

For exmple to checkout the latest from the development trunk:

```
svn co https://jnetpcap.svn.sourceforge.net/svnroot/jnetpcap/jnetpcap/trunk jnetpcap
```

As of development build 1.3.b0005, source files are no longer provided with the binary distribution of jNetPcap. Sources now reside in their own sperate packages such as RPMS or as -src for platforms that do not support specialized source packaging.

Common Prerequisites

In order to compile complete jNetPcap software you will need both C++ compiler and java development environments installed. When all the prerequisites and needed tools are installed in your environment, the compilation stage is simply execution of ANT script for "compile" or one of the "package" targets. There are no makefiles, all software is compiled using ant and various optional tasks.

In general the following prerequisites need to be installed before jNetPcap can be compiled.

- Java SDK 5.0 or above (jNetPcap does not compile under GNU java)
- libpcap library (libpcap-dev or WinPcap)
- g++
- ANT and optional tasks (which are: ant and ant-nodeps packages)
- Note: other required ANT packages are distributed in the lib directory, but may need to be updated if revision conflicts develop.
- jUnit library for compiling testcases
- subversion client to checkout the source code

Once this minimum set of prerequisites is installed you can run "ant compile" on the command line.

RPM packaging requirements

In addition to the common requirements, in order to be able to build RPM packages on various linux flavors the following is also required:

- rpmrebuild or rpm-build packages (called differently on different platforms)

The java **rpm ant task** is already located in the jnetpcap distributed lib directory and is referenced directly from the build file. It does not need to be installed by default.

Debian packagin requirements

The supported debian based environments already come with required packaging development support. This is comprable to

The java **deb ant task** required to perform the debian build, is already located in the jnetpcap distributed lib directory and is referenced directly from the build file. It does not need to be isntalled by default.

Compatiblity with GNU java environment (gcj)

jNetPcap bulids are not compatible with GNU java compiler `gcj` which is commonly and easily installed on various OS flavors. The incompatiblity is with JNI generated header files using `javah` command. GNU `javah` compiler *mangles* names of constants exported from java class files to JNI header files.

Name mangling, or making a name unique, is very common when using JNI mappings for C/C++ function names that are mapped to java *native* methods. However, GNU javah compiler also mangles the names of the constants that are exported to header files. This is different from the behavior of Sun provided JDK. This behavior is unportable between platforms. Therefore jNetPcap native library code can not be compiled with GNU javah compiler.

1.5.1 - RedHat/Fedora/CentOS

Commands to install development environment on a blank CentOS install:

Executed as root (user "mark" is only used as an example, replace with your own username):

1. `yum install java-1.6.0-openjdk-devel`
2. `yum install libpcap-devel`
3. `yum install rpmrebuild`
4. `yum install gcc-c++`
5. `yum install ant-nodeps` (needed for "javah" task)
6. `yum install ant-junit` (optional) install junit for running the ant "test" target
7. `yum install subversion`
8. (optional) Create user if you don't have an account (use your own user name):
   `useradd mark`
   and assign a password:
   `passwd mark`
9. `su - mark`
10. Checkout source code (complete details):

    ```
    svn checkout https://svn.code.sf.net/p/jnetpcap/code/jnetpcap/trunk jnp_1.4
    ```

    Note: please note the use of '_' (underscore) in the directory name. This is required by 'rpmbuild' tool if you plan on building rpm packages. The rpmbuild tool converts dashes into underscores causing problems with the build process. Therefore it is recommended to use underscores when naming the top level build directory.

11. As root run jUnit testcases to make sure everything compiles correctly. Tests need to be run as root, as some of the tests check interface parameters that require root privilege:

    ```
    su
    ant clean test
    #if everything tested OK
    exit
    ```

12. from the jnetpcap SVN working directory invoke the "package-rpm" target which will compile the shared library and all the java classes. Output is under "dist/" directory.

    ```
    ant -Dos=centos5 clean package-rpm
    ```

13.
14. `rpm -iv dist/jnetpcap-1.3.b0003-centos5.i386.rpm`
15. `ls -l /usr/lib/libjnetcap*`

    ```
    lrwxrwxrwx 1 root root      33 Jul 4 17:33 /usr/lib/libjnetpcap.so -> /usr/lib/libjnetpcap.
    -rw-r--r-- 1 root root 1193147 Jul 4 17:29 /usr/lib/libjnetpcap.so.1.3.b0003
    ```

16. `ls -l /usr/share/java/jnetpcap*`

    ```
    -rw-r--r-- 1 root root 463175 Jul 4 17:29 /usr/share/java/jnetpcap-1.3.b0003.jar
    lrwxrwxrwx 1 root root     38 Jul 4 17:33 /usr/share/java/jnetpcap.jar -> /usr/share/java/j
    ```

17. `ls /usr/share/doc/jnetpcap*`

    ```
    javadoc
    ```

Other names that should be used with the "-Dos=" argument to ant during build. You can substitute the major OS version number at the end:

1. fc9 - for FedoraCore OS
2. CentOS5 - for CentOS

This example assumes that we are taking a snapshot of the latest state of the development trunk (SVN trunk) which is assigned a build number (defined in file build.number in jnetpcap working directory). Build numbers are incremented after each official or internal build by jnetpcap builders.

1.5.1.1 - RHEL

Compiling on Red Hat Enterprise version, requires few extra step in order to install the neccessary build environment.

In addtion Red Hat Enterprise Linux version 4 uses fairely old version of libpcap (version 0.8.3) which is below standard requirements of jNetPcap installations. However in order to provide support on these still common platforms in production environments, RHEL installations have a lower requirement for libpcap. Libpcap versions below 0.9.7 do not provide support for 2 functions: `pcap_injet` and `pcap_sendpacket`. These functions are ommitted from jNetPcap API. When using jNetPcap API, you can check if these 2 functions are supported with methods: `Pcap.isPacketInjectSupported()` and `Pcap.isPacketSendSupported()`.

The following packages and their dependencies need to be installed for a build environment neccessary to compile jNetPcap software:

redhat> uname -a

```
Linux localhost.localdomain 2.6.9-89.ELsmp #1 SMP Mon Apr 20 10:34:33 EDT 2009 i686 i686 i386
GNU/Linux
```

redhat> rpm -q libpcap jdk gcc-c++ ant ant-nodeps subversion junit rpm-devel

```
libpcap-0.8.3-12.el4_6.1
jdk-1.6.0_02-fcs
gcc-c++-3.4.6-11
ant-1.6.5-4jpp
ant-nodeps-1.6.5-4jpp
subversion-1.1.4-2.ent
junit-3.8.2-4jpp
rpm-devel-4.3.3-32_nonptl
```

Some of these packages are not easy to aquire for red-hat platforms. The rpms can be either downloaded manually or using an installer such as **yum**.

Installing Sun's java JDK

You have to download the linux distribution manually from http://java.sun.com. The self extracting and installing package is very easy to install.

If there is already a java environment installed you need to setup **alternatives** to Sun's JDK.

```
alternatives --install /usr/bin/java java /usr/java/latest/bin/java 1
alternatives --install /usr/bin/javah javah /usr/java/latest/bin/javah 1
alternatives --config java (make sure JDK is selected is primary)
alternatives --config javah (make sure JDK is selected is primary)
setenv JAVA_HOME /usr/java/latest
```

Note: you can specify a specific version of JDK if more then one is installed to alternatives in the `/usr/java` directory.

Installing using up2date

Some of the above packages can be installed from local RHEL CD or from one of the RHN channels.

- up2date subversion
- up2date gcc-c++
- up2date libpcap
- up2date rpm-devel

Installing YUM on red-hat EL version 4

The remaing packages are easiest to install from `jpackage.org` yum repository.

You can skip this step if you already have **yum** installed and configured. Otherwise we have to download **yum** package and its prerequisites manually.

```
cd /tmp
mkdir yum
cd yum

wget http://dag.wieers.com/rpm/packages/yum/yum-2.4.2-0.4.el4.rf.noarch.rpm
wget http://dag.wieers.com/rpm/packages/python-elementtree/python-elementtree...
wget http://dag.wieers.com/rpm/packages/sqlite/sqlite-2.8.17-1.el4.rf.i386.rpm
wget http://dag.wieers.com/rpm/packages/python-urlgrabber/python-urlgrabber-2...

rpm -iv python-elementtree-1.2.6-7.el4.rf.i386.rpm
rpm -iv sqlite-2.8.17-1.el4.rf.i386.rpm
rpm -iv python-sqlite-1.0.1-1.2.el4.rf.i386.rpm
rpm -iv python-urlgrabber-2.9.7-1.2.el4.rf.noarch.rpm
rpm -iv yum-2.4.2-0.4.el4.rf.noarch.rpm
```

Next we need to configure yum for `jpackage.org` repository which contains all the required java-noarch type packages. You would replace or append to (if you already have a working yum configuration) the `/etc/yum.conf` configuration file with the following:

/etc/yum.conf

```
[main]
cachedir=/var/cache/yum
debuglevel=2
logfile=/var/log/yum.log
pkgpolicy=newest
distroverpkg=redhat-release
tolerant=1
```

```
exactarch=1
```

```
# Be sure to enable the distro specific repository for your distro below:
# - jpackage-fc for Fedora Core
# - jpackage-rhel for Red Hat Enterprise Linux and derivatives

[jpackage-generic]
name=JPackage (free), generic
mirrorlist=http://www.jpackage.org/mirrorlist.php?dist=generic&type=free&release=1.7
failovermethod=priority
gpgcheck=1
gpgkey=http://www.jpackage.org/jpackage.asc
enabled=1

[jpackage-fc]
name=JPackage (free) for Fedora Core $releasever
mirrorlist=http://www.jpackage.org/mirrorlist.php?dist=fedora-$releasever&type=free&release=1.7
failovermethod=priority
gpgcheck=1
gpgkey=http://www.jpackage.org/jpackage.asc
enabled=0

[jpackage-rhel]
name=JPackage (free) for Red Hat Enterprise Linux $releasever
mirrorlist=http://www.jpackage.org/mirrorlist.php?dist=redhat-el-$releasever&type=free&release=1.7
failovermethod=priority
gpgcheck=1
gpgkey=http://www.jpackage.org/jpackage.asc
enabled=0

[jpackage-generic-nonfree]
name=JPackage (non-free), generic
mirrorlist=http://www.jpackage.org/jpackage_generic_nonfree_1.7.txt
failovermethod=priority
gpgcheck=1
gpgkey=http://www.jpackage.org/jpackage.asc
enabled=0
```

Now we are ready to install various no-arch type packages needed by jNetPcap.

Installing remaining packages using yum

This would be quiet easy step now if it weren't for a gatcha. The ant package distributed with **jpackage.org** (usually have jpp somewhere in the name) has a bug (yes this is a reported bug by many) that has a prerequisite requirement for `sun-devel`. This requirement comes from installation on a SULinux and is not applicable for red hat linuxes, since no such package exists anywhere. Therefore what you need to do is install **ant** package prerequisites first using yum. The manually download the ant package and install it using rpm and force it to not do dependency check.

Here are ant dependencies:

```
package: ant.noarch 1.6.5-4jpp
dependency: jaxp_parser_impl
provider: classpathx-jaxp.noarch 1.0-0.1.beta1.10jpp
provider: crimson.noarch 1.1.3-17jpp
```

```
provider: xerces-j2.noarch 2.9.0-2jpp
dependency: config(ant) = 1.6.5-4jpp
provider: ant.noarch 1.6.5-4jpp
dependency: xml-commons-apis
provider: xml-commons-jaxp-1.2-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.3-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.1-apis.noarch 1.3.03-11jpp
dependency: jpackage-utils >= 1.6
provider: jpackage-utils.noarch 1.7.5-1jpp
dependency: java-devel
Unsatisfied dependency
```

Notice that java-devl is an unsatisfied dependency. This is a report bug with the packaging. It apparently had been removed as a requirement and then put back. So we have to install the dependencies first in a separate step:

```
yum install classpathx-jaxp
yum install crimson
yum install xerces-j2
yum install xml-commons-jaxp
yum install jpackage-utils
```

Next we manually download the ant rpm from `jpackage.org` repository and install it using rpm:

```
wget http://mirrors.dotsrc.org/jpackage/1.7/generic/free/RPMS/ant-1.6.5-4jpp....
rpm -iv --nodepends ant-1.6.5-4jpp.noarch.rpm
```

After ANT has been installed, you need to check and make sure that Sun's JDK hadn't been removed during this entire process. If it has, you need to re-install it.

Now we install the remaining packages:

```
yum install ant-nodeps
yum install ant-junit
```

Checkout source code and compile

After the environment is setup, you should now be able to checkout the source code and perform builds.

- Checkout source code

  ```
  svn co https://jnetpcap.svn.sourceforge.net/svnroot/jnetpcap/jnetpcap/trunk jnetpcap
  ```
- As root run jUnit testcases:

  ```
  su
  ant clean test
  exit
  ```
- `ant -Dos=rhel4 clean package-rpm`

Installing packages manually (not recommended)

It is possible to install the remaining packages manually. It is however much easier using yum (see installing using yum section).

To install manually you have to download manually all the require packages to your local system and then install using **rpm** command.

You can install all the packages from **jpackage** repository which contains most of the java based packages for various OSes. The web URL to the respository is

```
http://mirrors.dotsrc.org/jpackage/1.7/generic/free/repodata/repoview/
```

The packages are organized alphabetically and you can click on the first letter of the package from a menu at the top of the page. All of the above specified packages need to be download and installed using **rpm** command. It is however not easy manually to figure out what all the inter-dependencies are, but its not impossible.

Below is an attempt to list some of the dependencies by these various packages. Note that some of the dependencies can go to deeper level, but atleast here are the top-level dependencies:

ant package

```
package: ant.noarch 1.6.5-4jpp
dependency: jaxp_parser_impl
provider: classpathx-jaxp.noarch 1.0-0.1.beta1.10jpp
provider: crimson.noarch 1.1.3-17jpp
provider: xerces-j2.noarch 2.9.0-2jpp
dependency: config(ant) = 1.6.5-4jpp
provider: ant.noarch 1.6.5-4jpp
dependency: xml-commons-apis
provider: xml-commons-jaxp-1.2-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.3-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.1-apis.noarch 1.3.03-11jpp
dependency: jpackage-utils >= 1.6
provider: jpackage-utils.noarch 1.7.5-1jpp
dependency: java-devel
Unsatisfied dependency
```

ant-nodeps package

```
package: ant-nodeps.noarch 1.6.5-4jpp
dependency: ant = 1.6.5-4jpp
provider: ant.noarch 1.6.5-4jpp
```

junit package

```
No dependencies for this package
```

ant-junit package

```
package: ant-junit.noarch 1.6.5-4jpp
dependency: ant = 1.6.5-4jpp
provider: ant.noarch 1.6.5-4jpp
dependency: junit
provider: junit.noarch 3.8.2-4jpp
```

And here are the second level dependencies for all the dependencies of the **ant package:**

```
package: xerces-j2.noarch 2.9.0-2jpp
dependency: xml-commons-jaxp-1.3-apis >= 1.3
provider: xml-commons-jaxp-1.3-apis.noarch 1.3.03-11jpp
dependency: jaxp_parser_impl
provider: classpathx-jaxp.noarch 1.0-0.1.beta1.10jpp
provider: crimson.noarch 1.1.3-17jpp
provider: xerces-j2.noarch 2.9.0-2jpp
dependency: jpackage-utils >= 1.7.2
provider: jpackage-utils.noarch 1.7.5-1jpp
dependency: /bin/sh
Unsatisfied dependency
dependency: /usr/sbin/update-alternatives
Unsatisfied dependency
dependency: xml-commons-resolver11
provider: xml-commons-resolver11.noarch 1.3.03-11jpp
package: crimson.noarch 1.1.3-17jpp
dependency: /usr/sbin/update-alternatives
Unsatisfied dependency
dependency: xml-commons-apis
provider: xml-commons-jaxp-1.2-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.3-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.1-apis.noarch 1.3.03-11jpp
dependency: jpackage-utils >= 1.6
provider: jpackage-utils.noarch 1.7.5-1jpp
dependency: /bin/sh
Unsatisfied dependency
package: classpathx-jaxp.noarch 1.0-0.1.beta1.10jpp
dependency: /usr/sbin/update-alternatives
Unsatisfied dependency
dependency: xml-commons-apis
provider: xml-commons-jaxp-1.2-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.3-apis.noarch 1.3.03-11jpp
provider: xml-commons-jaxp-1.1-apis.noarch 1.3.03-11jpp
dependency: /bin/sh
Unsatisfied dependency
package: jpackage-utils.noarch 1.7.5-1jpp
dependency: /bin/sed
Unsatisfied dependency
dependency: /bin/egrep
Unsatisfied dependency
dependency: /usr/bin/perl
Unsatisfied dependency
dependency: /bin/sh
Unsatisfied dependency
```

Like I said, trying to resolve all of these dependencies can get quiet difficult rather quickly. None the less I have provided as much data as I can to help your resolve these dependencies manually.

1.5.2 - Debian/ubuntu

Commands to install development environment on a blank uBuntu install:

Executed as root (user "mark" is only used as an example, replace with your own username):

1. `apt-get install openjdk-6-jdk`
2. `apt-get install libpcap-dev`
3. `apt-get install g++`
4. `apt-get install ant` (pulls in about 25Mb of other packages)
5. `apt-get install ant-optional` (needed for "javah" task)
6. (optional) install junit for running the ant "test" target
   `apt-get install junit`
7. `apt-get install subversion`
8. (optional) Create user if you don't have an account (use your own user name):
   `useradd -m mark`
   and assign a password:
   `passwd mark`
9. `su - mark`
10. Checkout source code (complete details):

    ```
    svn co https://jnetpcap.svn.sourceforge.net/svnroot/jnetpcap/jnetpcap/trunk jnetpcap
    ```

    or

    ```
    svn checkout https://svn.code.sf.net/p/jnetpcap/code/jnetpcap/trunk jnp_1.4
    ```

11. `setenv JAVA_HOME /usr/lib/jvm/java-1.6.0-openjdk`
12. As root run jUnit testcases to make sure everything compiles correctly. Tests need to be run as root, as some of the tests check interface parameters that require root privilege:

    ```
    su
    ant clean test
    #if everything tested OK
    exit
    ```

13. from the jnetpcap SVN working directory invoke the "package-deb" target which will compile the shared library and all the java classes. Output is under "dist/" directory.

    ```
    ant -Dos=ubuntu9 clean package-deb
    ```

14. `dpkg -i dist/jnetpcap-1.3.b0003-ubuntu9.i386.deb`
15. `ls -l /usr/lib/libjnetcap*`

    ```
    lrwxrwxrwx 1 root root      33 Jul 4 17:33 /usr/lib/libjnetpcap.so -> /usr/lib/libjnetpcap.
    -rw-r--r-- 1 root root 1193147 Jul 4 17:29 /usr/lib/libjnetpcap.so.1.3.b0003
    ```

16. `ls -l /usr/share/java/jnetpcap*`

    ```
    -rw-r--r-- 1 root root 463175 Jul 4 17:29 /usr/share/java/jnetpcap-1.3.b0003.jar
    lrwxrwxrwx 1 root root     38 Jul 4 17:33 /usr/share/java/jnetpcap.jar -> /usr/share/java/j
    ```

17. `ls /usr/share/doc/jnetpcap*`

    ```
    javadoc
    ```

Other names that should be used with the "-Dos=" argument to ant during build. You can substitute the major OS version number at the end:

1. debian3 - for Debian OS
2. ubuntu9 - for ubuntu OS
3. gentoo2008 - for Gentoo 2008.X OSs

This example assumes that we are taking a snapshot of the latest state of the development trunk (SVN trunk) which is assigned a build number (defined in file build.number in jnetpcap working directory). Build numbers are incremented after each official or internal build by jnetpcap builders.

1.5.2.1 - gentoo

Commands to install development environment on a blank Gentoo install. Note there are no installable jnetpcap packages at this time, but the libraries can be generated using the following procedure:

Executed as root (user "mark" is only used as an example, replace with your own username):

1. `emerge dev-java/sun-jdk`
2. `emerge net-libs/libpcap`
3. `emerge gcc`
4. `emerge ant`
5. `emerge dev-java/ant-nodeps` (needed for "javah" task)
6. (optional) install junit for running the ant "test" target
   `emerge dev-java/junit`
7. `emerge subversion`
8. (optional) Create user if you don't have an account (use your own user name):
   `useradd -m mark`
   and assign a password:
   `passwd mark`
9. `su - mark`
10. Checkout source code (complete details):
    ```
    svn co https://jnetpcap.svn.sourceforge.net/svnroot/jnetpcap/jnetpcap/trunk jnetpcap
    ```
11. As root run jUnit testcases to make sure everything compiles correctly. Tests need to be run as root, as some of the tests check interface parameters that require root privilege:
    ```
    su
    ant clean test
    #if everything tested OK
    exit
    ```
12. `ant clean build-jar comp-jni`

Currently jNetPcap does not provide a emerge package that can be fully installed on Gentoo platforms. However the above instructions will install necessary build environment to compile both native and java libraries. The compiled libraries are left in the following location `build/lib`. The file `libjnetpcap.so.*` should be installed under /usr/lib directory. The java library `jnetpcap-X.Y.*.jar` should be installed under /usr/java directory.

This example assumes that we are taking a snapshot of the latest state of the development trunk (SVN trunk) which is assigned a build number (defined in file build.number in jnetpcap working directory). Build numbers are incremented after each official or internal build by jnetpcap builders.

1.5.3 - Android OS

(Notice: experimental, unsupported)

Several users now managed to compile and run jnetpcap (offline capture only) on Android operating system. This is extremely experimental at this point. So I'm posting information about the build process and the relevant android specific makefiles to build jnetpcap version 1.3 on Android without any support, but we will do our best to answer questions in the support forum.

Here are short instructions from Sergio Jimenez[1] and attached is the ZIP file containing the entire dev setup to succesfully build jnetpcap-1.3 for android:

> ***"Sergio Jimenez"* wrote:**
>
> Extract the JNI folder inside your project like this: /workspace/MyApp/JNI
>
> Download and install the Android NDK.
>
> Add the NDK directory to the PATH.
>
> Open a shell and go inside JNI directory.
>
> Type: ndk-build
>
> Have fun.
>
> The build parameters are inside the Android.mk in the JNI folder. Android.mk is An android specific Makefile.

Credits:
[1]Univesitat Politècnica de Catalunya: http://www.upc.edu/eng/
Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels: http://eetac.upc.edu/en/

Related Projects

A working Android capture tool: http://sourceforge.net/projects/prueba-android/

Attachment	Size
jni.rar	571.39 KB

1.6 - Using in Eclipse projects

jNetPcap is a java project that comes with a **required** *native shared library*. The requirement of a *native library* typically adds confusion and presents difficulty for many as to how properly setup a project in eclipse to reference jNetPcap library correctly.

There are several ways that jNetPcap can be added to your existing java project in Eclipse IDE. Let me briefly outline them here and then lets go through the detailed steps of actually creating a proper build path so your project will compile with jNetPcap.

1. Create a jNetPcap "user library" which has all the neccessary path components configured and add that to the project's build path
2. Add jnetpcap's jar file and native library directory path to project's build path
3. Add jnetpcap's jar file to project's build path, but put the native library in global environment variable (LD_LIBRARY_PATH under unix and PATH variable under windows)
4. Add jnetpcap's jar file to project's build path, but copy the neccessary native library to a system library directory (/usr/lib under unix or \windows\SystemXX under windows).

We recommend approaches #1 and #2 for development. If you are creating a single jnetpcap dependent project, approach #2 may be all you need. On the other hand if you want to set it up once for many projects, approach #1 is what we recommend.

Setup #3 and #4 are not recommended for a build environment, and will not be discussed here.

First thing first

First thing you have to do is download and install (or unzip) the jNetPcap installation package. You do not have to install (unzip or untar) the installation package under an Eclipse workspace, unless you want to for a specific. The installation can be external to the workspace. Since each jNetPcap installation package installs under a unique directory path, you can easily have multiple versions of the library and switch between them when needed. Both installable and extractable unix and windows packages are provided. Under unix the packager installed packages are intended for production environments, that have a jNetPcap requirement. At same time the JAR and unzip packages are provided incase you need multiple versions of the library where you can extract on your own and easily switch between them.

In the below examples we are going to assume that we extracted 2 versions of jnetpcap library under "c:\libs" directory (on a windows platform). For unix you can assume a home directory based path "$HOME/libs" or something similar. In the "libs" directory we installed version jnetpcap-1.2 and jnetpcap-1.3.b0010.

We are also going to assume that the user has extracted/installed all 3 packages of jnetpcap (the executable package), jnetpcap-javadoc, jnetpcap-src for both versions above.

Setup #1 - Setting up a "user library" under Eclipse (*recommended setup*)

Setting up a "user library" under eclipse IDE platform, is a way for you to define in one place a single or multiple java libraries along with all of their requirements such as "native libraries", javadoc documentation and where to do lookups for source code incase you want to drill down into a function. This is also a neat way to change versions of the library globally without having to modify build paths for each project you have setup.

To create a new "user library" under Elicpse is easy.

1. Open Preferences dialog

   ```
   Window->preferences
   ```

2. In the dialog select "User Libraries"

   ```
   either search "user" in dialog's search box
   ```

   or

   ```
   Java->Build Path->User Libraries
   ```

3. Then click "New..." button or "Import..." if you already have a user library definition saved.
4. Enter "jNetPcap" for the name of the user library. Leave the checkbox "System Library" as OFF.
5. Next we want to add actual java jar files that make up this user library

   ```
   Click the "Add JARs..." button
   ```

6. Select the "jnetpcap.jar" file under where jnetpcap was installed or extracted to (under c:\libs for our example.)
7. That should add tree content to your "user library". Expand the "jnetpcap.jar" file that appeared under "jnetpcap" user library.
8. Select "Native library location" and click "Edit..." button. There again select the directory where jNetPcap has been installed to (our C:\libs\jnetpcap-1.2 as an example).
9. Also select and "Edit..." the locations to "Javadoc location" and "Source attachment". You can point directly at the ZIP and tar files without having to extract those first.
10. Optionally once the "user library" is setup, you can export it to a file so that you can import it into

other Eclipse "workspaces" or if you want to keep a safe copy.

11. Last thing to do is to hit the "OK" button and we are done

Now that a "user library" has been setup, you can add this user library to any of your Eclipse projects that need jnetpcap. You add the "user library" to each project's "Build Path". To do that, here are the instructions:

1. Select the project you want to add jnetpcap user library to and open its properties:

   ```
   Project->Properties->Java Build Path
   ```

   or right click on the project in "Package Explorer" and select

   ```
   Build Path->Configure Build Path
   ```

2. Select the "Libraries" TAB and click "Add Library..." button
3. In the "Add Library" dialog box that shows up, select "User Library" and click "Next >" button at the bottom.
4. Your newly created "user library" should show up in a list of 1 or more "user libraries". Each library has a check box and you want to select the check box next to "jnetpcap"
5. Optionally, if you haven't setup a jnetpcap "user library" yet, there is a "User Libraries..." button within this dialog that lets you create one right there.
6. We are done, so we click the "Finish" button.

You can do this for all other java projects in your workspace that need jnetpcap library.

The benefit of using a "user library" is that you can easily change the version of jnetpcap the "user library" is referencing by modifying the paths for jar, native, javadoc and source locations in this one place. You may also create multiple "user libraries" that reference different versions of 'jnetpcap' user library. For example, you can setup a "jnetpcap-production" library that references a production version of jnetpcap and additional user libraries such as "jnetpcap-latest", "jnetpcap-1.3.b0010", etc. Then decide which of your own projects should be using which version of the library.

The source and javadocs will automatically be changed as well, making this a very robust development environment for your project.

Setup #2 - Adding jnetpcap JAR file directly to your project

Another approach is to add the jnetpcap.jar file (from directory c:\lib\jnetpcap-1.2 for example) to your project's build path. Once added you can still modify for that particular library where the required native library resides. This is a good approach if you only have a single java project that needs jnetpcap.

To add jnetpcap to your project's build path:

1. Select the project you want to add jnetpcap library to and open its properties:

   ```
   Project->Properties->Java Build Path
   ```

   or right click on the project in "Package Explorer" and select

   ```
   Build Path->Configure Build Path
   ```

2. Click the "Add External JARs..." button (or "Add JARs..." if you installed jnetpcap within the Eclipse workspace)
3. Go to your installation directory (c:\libs\jnetpcap-1.2 for example) and select the jnetpcap.jar file and click "Open" button to add the jar file to the project's build path.
4. A "jnetpcap.jar" file should show up in the "build path" tree. Expand it.
5. Select "Native library location" and click "Edit..." button.
6. Select the directory where the native file resides (c:\libs\netpcap-1.2 for example).

7. Optionally, you can do the same for "Javadoc location" and "Source attachment". You do not need to expand them from their zip or tar forms. Eclipse will take them in archive form.
8. Lastly, click the "OK" button at the bottom

Now jnetpcap library and its native library are added to your project. If you added javadoc and source locations, you can also see javadocs in Eclipse (hover or javadoc display) and drill down into jnetpcap methods and view their source.

1.7 - Using in Netbeans projects

jNetPcap is a java project that comes with a **required** *native shared library*. The requirement of a *native library* typically adds confusion and presents difficulty for many as to how properly setup a project in netbeans to reference jNetPcap library correctly.

There are several ways that jNetPcap can be added to your existing java project in Netbeans IDE. Let me briefly outline them here and then lets go through the detailed steps of actually creating a proper build path so your project will compile with jNetPcap.

1. Create a jNetPcap "library" which adds the jnetpcap-*.jar file to the build path
2. Create a new run project config that includes native library
3. Add jnetpcap's jar file to project's build path, but copy the neccessary native library to a system library directory (/usr/lib under unix or \windows\SystemXX under windows).

We recommend approaches #1 and #2 for development.

Note: the native library is only required for running/executing the application. It is not required for compilation. It is needed only at runtime.

First thing first

First thing you have to do is download and install (or unzip) the jNetPcap installation package. You do not have to install (unzip or untar) the installation package under an Netbeans workspace, unless you want to for a specific reasons. The installation can be external to the workspace. Since each jNetPcap installation package installs under a unique directory path, you can easily have multiple versions of the library and switch between them when needed. Both installable and extractable unix and windows packages are provided. Under unix the packager installed packages are intended for production environments, that have a jNetPcap requirement. At same time the JAR and unzip packages are provided incase you need multiple versions of the library where you can extract on your own and easily switch between them.

In the below examples we are going to assume that we extracted jnetpcap library under "G:\libs" directory (on a windows platform). For unix you can assume a home directory based path "$HOME/libs" or something similar. In the "libs" directory we installed the binary jnetpcap-1.4.b0004-1.win32.zip, the source package jnetpcap-src-1.4.b0004-1.zip and jnetpcap-javadoc-1.4.b0004-1.zip. We further unzipped only the binary package jnetpcap-1.4.b0001-1.win32.zip to that directory. The other packages remain in their zipped up form. Netbeans knows how to read them archived.

We now have the following files and directories under "g:\libs":

jnetpcap-1.4.b0004-1	<directory>
jnetpcap-1.4.b0004-1.win32.zip	<zip archive>
jnetpcap-javadoc-1.4.b0004-1.zip	<zip archive>

jnetpcap-src-1.4.b0004-1.zip <zip archive>

The jar and .dll files are directly under the "jnetpcap-1.4.b0004-1" directory.

Setting up a new library under Netbeans

The recommended way to setup environment is to setup a new global library under "Libraries". This library will only contain the path to the jnetpcap-*.jar file. It can be added to any netbeans project which will then have access to jnetpcap API.

Assuming that you have already created a new java project, we now create a new library definition for netbeans library manager.

1. Under the "Tools" menu, click on "Libraries" menu item. A "Library Manager" window should show up.
2. If you are setting this up for the first time, library manager does not have a jnetpcap library defined yet. You need to now click on "New Library..." button below the list of existing libraries.
3. A "New Library" window show up. Type in a name for this new library declaration. I suggest using "jnetpcap-1.4" for our example. It is best to include version number of jnetpcap incase you want to setup more then one library and easily switch between them. For "Library Type" selection box, choose "Class Libraries".

   Click OK

4. Now we can define the rest of the elements that make up this library. Click on the "Classpath" tab and then click "Add JAR/Folder..." button. When file browser comes up, navigate to "g:\libs \jnetpcap-1.4-1" directory and select the jar file "jnetpcap-1.4.b0004-1.jar". Close the browser window by clicking "Add JAR/Fold" button. You should now wee the absolute path to "jnetpcap-1.4.b0004-1.jar" file listed under "Library Classpath:" pane.
5. Next click on "Sources" tab. Browser window shows up again. You should be 1 level up from where the -src package resides, so just hit the "UP array/folder" to go up 1 level. Select "jnetpcap-src-1.4.b0004-1.zip" file and click "Add JAR/Folder" button to exit and commit.
6. Next click on "Javadoc" tab. Browser window shows up again. Now select the "jnetpcap-javadoc-1.4.b0004-1.zip" file and click "Add JAR/Folder" button to exit and commit.
7. Dismiss the "Library Manager" window with a click on "OK" button.
8. Now we apply the library to our java project. Under project explorer, right click on the "Libraries" element and select "Add Library" or "Properties" or choose from menu "File->Project Properties" and select libraries section.
9. From the "Add Library" window, choose our newly created library "jnetpcap-1.4" and click "Add Library" button at the bottom. The window should disappear, and if you expand the "Libraries" element under project explorer, you should see our "jnetpcap-1.4" jar file on the libraries classpath.

Now we have setup our "library" for compilation. You should be able to now to have access to full jnetpcap API, go into jnetpcap methods to view their source and look at their javadocs directly from netbeans editor.

Next we need to setup a "run" configuration so that our application finds the require native library (.dll on windows, .so on unix/linux).

1. Select from the menu "Run->Setup Project Configuration->Customize...". This will bring up a "Project Properties" window.
2. Click on "New..." button to the right of the "Configuration:" selection box. Enter a new "Configuration Name". Again I would recommend using "jnetpcap-1.4 config" to specify that we are setting up for jnetpcap 1.4 runtime environment. Click "OK" button to dismiss.
3. Now in the "VM Options:" text field, we need to enter the following options"

```
-Djava.library.path="g:\libs\jnetpcap-1.4.b0004-1"
```

Notice the double-quotes around the path and the minus D (-D) option. Do not forget either. Lastly notice, we entered the name of the directory, not the name of the .dll library itself.
Now Click "OK" to dismiss.

We are now ready to run our jnetpcap application. Netbeans set our run configuration to "jnetpcap-1.4 config" automatically. We did not modify the "default" config so that we can easily switch back to "default" config through menus or the selection config box on the "Run" toolbar. We can also create other run configs this way that point to other jnetpcap versions and easily switch between them. Just remember to switch both the "library" and the "run" config to the same versions.

Ch 2 - libpcap

This chapter discusses the main API of the libpcap wrapper itself. The most important class in all of jNetPcap SDK is the `Pcap` class. This class provides numerous static methods for getting captures started and setting various capture properties such as kernel capture buffer size, timeouts, filters, truncating packets and getting error messages.

You will find almost everything discussed in this chapter in java package `org.jnetpcap`. The `JBuffer` class is found in `org.jnetpcap.nio` and `PcapPacket` and `JPacket` classes are found under `org.jnetpcap.packet` package. Lastly the briefly mentioned `ByteBuffer` is part of the standard JRE 1.4 and above in package `java.nio`

2.1 - The Main libpcap API Overview

Java Package: *org.jnetpcap*

This package contains the core libpcap functions. The first thing to notice is that the programming style follows more the C convention that libpcap library was written in than Java style. That is none of the functions in the package throw any exceptions, especially the familiar `java.io.IOException` that is typically thrown where network communication occurs. Instead method return integer return codes, fill in error buffers with error strings, etc.

There are several reasons for this programming style. First and the most important one is that libpcap has been ported to nearly every operating system out there, literally close to a hundred. Libpcap has various small quirks that appear on different platforms. For advanced programmers these quirks are essential and is what gives libpcap its power. Instead of trying to abstract away every single possible scenario, jNP instead returns the raw data, result codes and error/warning messages. The style also makes it easy for those already familiar with the C libpcap library to just jump right in.

jNP is of course written in java and has made slight modifications to original function names such as `find_all_devs` has been named in java `findAllDevs` which is an easy adjustment for any C programmer.

The typical programming steps when working with this package is to

1. Setup an error buffer
2. Get a list of all available interfaces and pick one (see `Pcap.findAllDevs()`
3. Open either a live network interface, discovered in previous step, or open an offline, a capture file (see `Pcap.openLive()` or `Pcap.openOffline()`)
4. Read either one packet at a time (`Pcap.nextEx()`) or setup a dispatch loop (see `Pcap.loop()` or `Pcap.dispatch()`)
5. If using a dispatch loop, wait in your callback method (see `PcapPacketHandler.nextPacket()`) and

receive incoming packets.

6. Once you have received a packet, typically the packet is either processed on the spot or put on a queue that is read by another thread
7. When the dispatch loop exists due to either an interruption (see `Pcap.breakLoop()`)or simply the requested number of packets at the time the loop was setup, then process the queue if it hasn't been handed off or cleanup
8. Always the last step is to close the pcap handle (see `Pcap.close()`) to allow Pcap to release all its resources held

Here is a link to a fully functioning example that demonstates these basic steps:
Classic Example

2.2 - Getting a List of Interfaces

Java package: *org.jnetpcap*

When you want to capture packets directly from a live network, you first must acquire a list of available network interfaces and then choose which one to open. Interface names differ greatly on each platform and can change order at anytime even on the same system.

Therefore it is not usually easy to programmaticaly choose a network interface on behalf of the user, and usually that decision is left up to the user either through a configuration option in your application or some user interface that lets the user choose one interface.

There is a function provided by `Pcap` class to do retrieve a list of interfaces. It is `Pcap.findAllDevs()` function. It returns an integer error code and fills in a supplied collections `List` object with `PcapIf` objects. Each `PcapIf` is a separate interface found on this particular system.

```
List<PcapIf> alldevs = new ArrayList<PcapIf>(); // Will be filled with NICs
StringBuilder errbuf = new StringBuilder();     // For any error msgs

int r = Pcap.findAllDevs(alldevs, errbuf);
if (r == Pcap.NOT_OK || alldevs.isEmpty()) {
  System.err.printf("Can't read list of devices, error is %s", errbuf.toString());
  return;
}
```

Notice that the supplied list to `Pcap.findAllDevs()` is a regular JRE list which is filled with interfaces. This was another place where following the actual native libpcap programming style did not make sense. So we use a much more java friendly list than a linked list of some objects.

The important thing we are trying to acquire is the name of the interface the user or your application is interested in capturing on. We can get the name of the interface using `PcapIf.getName()` method and pass that into `Pcap.openLive()` for example or build a menu for the user with it.

Once we have a network interface chose, its time for the next step which is to open that network interface for reading. `PcapIf` does not provide any methods directly to open interface, we have to use `Pcap.openLive()` for that, discussed in the section.

Note about special "any" device on Linux systems

***http://www.tcpdump.org/pcap3_man.html* wrote:**

On Linux systems with 2.2 or later kernels, a device argument of "any" or NULL can be used to capture packets from all interfaces. snaplen specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first snaplen bytes of that packet will be captured and provided as packet data. A value of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet. promisc specifies if the interface is to be put into promiscuous mode. (Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.) For now, this doesn't work on the "any" device; if an argument of "any" or NULL is supplied, the promisc flag is ignored.

2.2.1 - Getting hardware MAC address from interface

You can get interface's MAC address directly from `PcapIf` class by calling on `PcapIf.getHardwareAddress()`. This method will return a byte[] containing the hardware address. The length of the array tells the exact length of the address. If null is returned, that means that the physical interface, does not support hardware addresses, such as loopback interface, a PPP interface, etc..

Notice that `PcapIf.getHardwareAddress()` method is a convenience method. The actual implementation of this method is in the `PcapUtils.getHardwareAdderess()` call. Libpcap library does not provide any function calls to acquire a hardware address of an interface. This has been added as an extension to libpcap in a `PcapUtils` class.

2.3 - Opening a Network Interface for Capture

Java package: *org.jnetpcap*

Once we have a network interface in the form of a `NetIf` object the user or the application is interested in opening, we can proceed to opening it. The main call which does this is `Pcap.openLive()`. This method takes a number of parameters, but the most important one is the name of the network interface. The name is a string, not the `NetIf` object itself, but we can easily acquire the name using `NetIf.getName()` and pass that in directly to `openLive()` call.

```
int snaplen = 64 * 1024;           // Capture all packets, no truncation
int flags = Pcap.MODE_PROMISCUOUS; // capture all packets
int timeout = 10 * 1000;           // 10 seconds in millis
Pcap pcap = Pcap.openLive(device.getName(), snaplen, flags, timeout, errbuf);

if (pcap == null) {
  System.err.printf("Error while opening device for capture: "
        + errbuf.toString());
  return;
}
```

Where device is our `NetIf` object (acquired in previous section's example.) When successful the call returns an instance of `Pcap` object. Notice that we used a static `Pcap` call to open, but then when successful we will be working with specific instance of `Pcap` which is bound to the network interface we opened.

The other parameters set various libpcap options which can limit the number of packets captured and if the capture should ever timeout when no packets are being received. The last parameter is an error buffer, a simple `StringBuilder` object which will be filled in with any warnings or error messages. If the call to `openLive()` fails, a null will be returned from the call. We can check for that null and expect an error message to be present in our error buffer.

Pcap.openLive(): Pcap.MODE_PROMISCOUS flag

What is a promiscous mode. Here is the definition from wiki pedia:

"In computing, promiscuous mode or promisc mode is a configuration of a network card that makes the card pass all traffic it receives to the central processing unit rather than just packets addressed to it — a feature normally used for packet sniffing." - **Source: Wikipedia**

So if you want to see all network traffic that your network card can see, you need to set this mode. If you don't set this mode, the network interface will only see network packets that are specifically addressed for the machine the capture is taking place. It will also always see all broadcast packets and may be some multicast packets.

Pcap.openLive(): snaplen parameter

This option to `Pcap.openLive` sets the maximum number of bytes to capture from the network. If the packet being captured is bigger than the snaplen length set, then that packet will be truncated in length to snaplen, otherwise it will be captured full length.

This is the reason in jNP API you will frequently encounter to lengths, caplen and wirelen. Caplen is used by jNP API as the number of bytes that were captured and kept, while wirelen parameters and properties specify the original length of the packet as it was seen on the network.

Why would you want to set a snaplen? If your application is only interested in headers that are always found at the beginning of a network packet and it doesn't care about the data, packets are typically truncated to 128 bytes or less. 128 bytes is usually enough to capture all the headers present and drop the data portion. Note that this is simply an estimate which may not capture all the headers if the packet is complex or it may still capture a number of data bytes of the packet past the last header. This all depends on the packet itself and type of network and communication involved. Tunneled packets will usually have a significantly greater number of headers than non tunneled packets. Icmp error messages also carry original packets that caused the error in the first place which adds significant amount of headers to a packet.

IMPORTANT WARNING: libpcap has a quirk with snaplen. Snaplen is only applied when there is also a filter applied at the same time. Without setting a filter using `Pcap.setFilter()`, even the most basic one such as "1", the snaplen parameter is ignored and packets will not be truncated. This is a know libpcap issue and is not an issue with jNetPcap API.

Pcap.openLive(): timeout parameter

The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it wait for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. Not all platforms support a read timeout; on platforms that don't, the read timeout is ignored. A zero value for to_ms, on platforms that support a read timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout.

2.4 - Opening offline capture

Java package: *org.jnetpcap*

If you have a file with captured packets in it in one of the support libpcap file format, you can open an offline pcap capture using `Pcap.openOffline()` call. The call opens up the file for reading packets and when successful returns an instance of `Pcap` object. This is similar as discussed in previous section but the `pcap` instance is not bound to any network interface, but to the file.

```
String fname = "myfile.pcap";

Pcap pcap = Pcap.openOffline(fname, errbuf);
if (pcap == null) {
  System.err.printf("Error while opening device for capture: "
    + errbuf.toString());
  return;
}
```

2.5 - Setting a packet filter

It is possible to set a packet filter directly on the pcap capture and only have packets delivered to your application that match and pass the filter expression.

The filter is built from a human friendly textual expression using libpcap filter syntax. The expression, presented as a string, is filter compiled to a `PcapBpfProgram` which holds the binary representation of the filter expression, and then the bpf program is set on the pcap capture using `Pcap.setFilter()` call.

```
PcapBpfProgram program = new PcapBpfProgram();
String expression = "host 192.168.1.1";
int optimize = 0;         // 0 = false
int netmask = 0xFFFFFF00; // 255.255.255.0

if (pcap.compile(program, expression, optimize, netmask) != Pcap.OK) {
  System.err.println(pcap.getErr());
  return;
}

if (pcap.setFilter(program) != Pcap.OK) {
  System.err.println(pcap.getErr());
  return;
}
```

(*View complete SetFilterExample.java*)

More filter syntax examples

(*Source: tcpdump man page*)

```
       To  print  all  packets arriving at or departing from sun
       down:
              host sundown

       To print traffic between helios and either hot or ace:
              host helios and ( hot or ace )

       To print all IP packets between ace and  any  host  except
       helios:
              ip host ace and not helios

       To  print  all  traffic  between  local hosts and hosts at
       Berkeley:
              net ucb-ether

       To print all ftp traffic through  internet  gateway  snup:
       (note  that  the expression is quoted to prevent the shell
       from (mis-)interpreting the parentheses):
              gateway snup and (port ftp or ftp-data)
```

```
To print traffic neither sourced  from  nor  destined  for
local  hosts  (if you gateway to one other net, this stuff
should never make it onto your local net).
       ip and not net localnet

To print the start and end packets (the SYN and FIN  pack
ets)  of  each  TCP conversation that involves a non-local
host.
       tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net localnet

To print IP packets longer than  576  bytes  sent  through
gateway snup:
       gateway snup and ip[2:2] > 576

To  print  IP broadcast or multicast packets that were not
sent via ethernet broadcast or multicast:
       ether[0] & 1 = 0 and ip[16] >= 224

To  print   all   ICMP   packets   that   are   not   echo
requests/replies (i.e., not ping packets):
       icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply
```

(*Note: the quoted examples above removed **tcpdump** keyword and certain shell escape sequences from the examples, as this is irrelevant for jNetPcap since the filter string is passed directly to* `Pcap.setFilter`)

2.6 - Reading one packet at a time

There are 2 methods provided by `Pcap` class for reading a single packet at a time from an open pcap capture. The first is `Pcap.next()` and Pcap.nextEx().

The preferred method is to use `Pcap.nextEx()` according to libpcap documentation. It replaced the obsolete `Pcap.next()` which had some severe limitations and quirks.

The call `Pcap.nextEx()` is easy to use and there are several variations of the method with exact same name:

1. `Pcap.nextEx(PcapHeader, JBuffer)`
2. `Pcap.nextEx(PcapHeader, ByteBuffer)`
3. `Pcap.nextEx(PcapPacket)`

The first 2 methods peer (changing internal reference to point at different memory location, like set a C pointer) the capture header supplied by libpcap library itself. The second peers the buffer to the data buffer also supplied by libpcap.

The 3rd method is a little different in that, you get back a completely decoded packet and you don't have to work with raw headers and data buffers. (see Chapter 3 - Packet Decoding Framework)

Note: Notice that data is shuttled between java and native space using peering not by copying. Neither the capture header or the packet buffer is copied even once. `JBuffer` class provides methods that can access the native memory provided by libpcap directly just like JRE's `java.nio.ByteBuffer` class.

Although reading one packet at a time is easy, it is very inefficient. For bigger application it is usually necessary to setup dispatcher loops where packets are dispatched to the user much more efficiently as discussed in the next section.

2.7 - Reading multiple packets with dispatch

loops

Java package: *org.jnetpcap*

Sophisticated applications usually need to read a lot of packets as efficiently as possible. Libpcap provides 2 dispatcher loops that buffer and dispatch packets with minimal kernel interruption.

The loops callback to user supplied functions when libpcap wants to deliver packets to the application. The exact timing of the deliveries is very libpcap implementation dependent. This is a function of the hardware architecture, network architecture nad kernel architecture.

We do not have to deal with any of that for the most part and are simply provided two functions (`Pcap.loop()` and `Pcap.dispatch()` which setup the dispatch loops. We provide a callback handler which gets called with appropriate headers, buffers and even decoded packets when libpcap is ready to deliver.

jNP does not copy packet data or headers, what it does it peers (think of setting a C pointer) java objects to native memory structures and libpcap provided data buffer. For that reason several different callback handlers are provided:

- `ByteBufferHandler` class - callback signature `nextPacket(PcapHeader, ByteBuffer)`
- `JBufferHandler` class - callback signature `nextPacket(PcapHeader, JBuffer)`
- `JPacketHandler` class - callback signature `nextPacket(JPacket)`
- `PcapPacketHandler` class - callback signature `nextPacket(PcapPacket)`

The familiar JRE class `java.nio.ByteBuffer` already provides what it terms direct memory access (see `ByteBuffer.isDirect()`). jNP library sets the physical memory that a `ByteBuffer` object references to that of the returned libpcap provided packet buffer. No copies are done. There is a limitation however, the memory location to which the buffer will point at can only be set at the time the `ByteBuffer` is being created in native JNI code, therefore although no copies of the data are made, a new `ByteBuffe` object is created for every callback to java world. This limitation is lifted by new JNP provided buffer described below.

jNP API adds a new type of buffer class, `JBuffer`, that can be manipulated more easily than a `ByteBuffer` class. `JBuffer` class provides nearly the same type of accessor methods as JRE's `ByteBuffer` with few exceptions. It can also be reassigned to new memory location by jNP implementation.

The last two callback handlers deal with packet objects. jNP packet framework, provides decoding capabilities to packet buffers. `JPacket` class is a base class for `PcapPacket`. Therefore if you don't neccessarily require information that is specific to pcap packets, you can work with more generic `JPacket` handler. However if need more advanced access to the packet and manipulation, you need to work with `PcapPacket` based handler. Currently only 1 type of packet is supported, that is the pcap packet. In the future new packet types and libpcap extensions may be supported.

2.7.1 - JBufferHandler and ByteBufferHandler

A handler that receives raw packet capture header and data. Both handlers work almost identically, where the handler's `nextPacket()` method receives either a `JBuffer` or `ByteBuffer` objects.

Exact signatures are:

```
class JBufferHandler<T>;
JBufferHandler.nextPacket(PcapHeader header, JBuffer buffer, T user);
```

Or the direct ByteBuffer alternative:

```
class ByteBufferHandler<T>;
ByteBufferHandler.nextPacket(PcapHeader header, ByteBuffer buffer, T user);
```

Where *user* is a user supplied object of `T` class type.

Here is the simplest example of how a handler is used:

```
StringBuilder errbuf = new StringBuilder();
Pcap pcap = Pcap.openOffline("tests/test-afs.pcap", errbuf);

JBufferHandler<String> handler = new JBufferHandler<String>() {
  public void nextPacket(PcapHeader header, JBuffer buffer, String user) {
    System.out.println("size of packet is=" + buffer.size());
  }
}

pcap.loop(10, handler, "jNetPcap rocks!");

pcap.close();
```

A `ByteBufferHandler` would look very similar:

```
StringBuilder errbuf = new StringBuilder();
Pcap pcap = Pcap.openOffline("tests/test-afs.pcap", errbuf);

ByteBufferHandler<String> handler = new ByteBufferHandler<String>() {
  public void nextPacket(PcapHeader header, ByteBufferBuffer buffer, String user) {
    System.out.println("size of packet is=" + buffer.capacity());
  }
}

pcap.loop(10, handler, "jNetPcap rocks!");

pcap.close();
```

In both cases, we use the `loop()` to setup the dispatch loop, we ask it to supply us with 10 packets, we register our call back handler, an anonymous class that we define inline and lastly we supply our user parameter which is a simple string.

In both cases, no data copies take place, not even for the 16 byte `PcapHeader`. The data is peered with the header and buffer at native level. This provides very fast access to data and in certain situations guarrantees that packet data is never copied all the way from the hardware capture device, our network interface, all the way to the end user application, our java program.

There is only one minor difference between a `JBuffer` and `ByteBuffer` based handlers. With every new packet received, a new `ByteBuffer` has to be allocated so that it can be peered with the packet data buffer. This is different with `JBuffer` implementation. It reuses the same `JBuffer` to deliver the packet to the user handler. That is only one instance of `JBuffer` is ever created and simply reused for every packet received. To be precise, a new `JBuffer` is allocated at the time the `Pcap.loop()` or `Pcap.dispatch()` is invoked, to handle that one call instance. Multiple invocations of the loop at the same time utilize different instances of `JBuffer`. This allows simple multi-threading capabilities and makes sure that they do not clobber each other's `JBuffer` peers.

This means, that the delivered buffers, are not suitable for permanent storage, even when putting on a simple short java collection's queue. The same is true for the `ByteBuffer` implementation. Even though a new `ByteBuffer` is allocated for every packet, the data it points to is at a libpcap revolving, round robin, buffer that is used to buffer packets before they are dispatched to the user's application. That means that those are also temporary packets and therefore not even the `ByteBuffer` can be kept for more than one iteration of the dispatch loop since it is unknown exactly when the buffer wraps around.

The only solution for longer storage is to reallocate the memory and copy the contents of both the data and the header. Both `ByteBuffer` and `JBuffer` classes provide method for fast native memory copies. `JBuffer` also works with `JMemoryPool` for efficient native memory management.

The copying of data is a time consuming operation and is delayed and left up to the user. The peering process itself is very light and fast. The user has great options for copying the data to permanent memory locations. More on that subject in another section.

2.7.1.1 - How to peer packets

When using `JBufferHandler` or `ByteBufferHandler` it may be necessary to peer an actual packet object to the buffers received within the handler. The same restrictions still apply as described in section 2.7.1 about proper usage of libpcap provided ring buffers.

We can easily wrap around the `PcapHeader` and the buffer received in our handler, with an instance of `PcapPacket` object. Since we are not going to be doing anything with the packet object outside of our handler, for efficiency we are going to allocate a single private instance of `PcapPacket` object and mark it final. Here is an example of this usage in `JBufferHandler` which is the most efficient handler provided by jNetPcap library. The example omits the pcap capture and dispatch loop setup:

```
JByteBufferHandler<String> handler = new JBufferHandler<String>() {

  private final PcapPacket packet = new PcapPacket(JMemory.POINTER);

  public void nextPacket(PcapHeader header, JBuffer buffer, String msg) {
    packet.peer(buffer); // Peer the data to our packet
    packet.getCaptureHeader().peerTo(header); // Now peer the pcap provided header

    packet.scan(Ethernet.ID); // Assuming that first header in packet is ethernet
  }
}
```

The handler delivers to us a header and packet data object (both peered to libpcap ring buffer). Both of these objects have a lifecycle only within the handler method and go out of scope outside of it. These objects are not appropriate to put on an outside queue, list or any other storage without copying.

We peer or wrap around libpcap ring buffer memory, with a `PcapPacket` object. We perform a scan of the headers using the `JPacket.scan(int dlt)` method. At this point the packet object is suitable to use with protocol headers. We can then use the standard headers like so:

```
Ip4 ip = new Ip4();
Tcp tcp = new Tcp();
if (packet.hasHeader(ip) && packet.hasHeader(tcp)) {
  System.out.println(ip.toString());
  System.out.println(tcp.toString());
}
```

If we keep our ip and tcp references as handler's fields, all objects we use are only allocated once when the handler is created and reused for every iteration of the packet with no memory copies. This type of handler setup allows the fastest and most efficient processing of captured packets.

2.7.2 - PcapPacketHandler and JPacketHandler

These two handlers are very similar and identical in implementation, so we will only discuss `PcapPacketHandler` which provides all the capabilities. `JPacketHandler` provides diminished API that is generic and available across all packet types (there is currently 1 packet type.)

Just like with `JBufferHandler` a single copy of `PcapPacket` is reused for every packet from the same instance of the pcap dispatch loop. The packet arrives fully decoded and can be accessed immediately, but can not be put away onto a queue or other permanent/semi-permanent storage. It needs to be either processed immediately by the user's application, discarded or copied to more permanent memory location.

Lets take a look at the handlers exact signature:

```
class PcapPacketHandler<T>;
PcapPacketHandler.nextPacket(PcapPacket buffer, T user);
```

Where *user* is a user supplied object of `T` class type. The signature is the simplest of all the handlers. The packet header can be accessed from the packet using `PcapPacket.getCaptureHeader()` and the user parameter is still supplied at the end of the parameter list.

Here is the simplest example of how a handler is used:

```
StringBuilder errbuf = new StringBuilder();
Pcap pcap = Pcap.openOffline("tests/test-afs.pcap", errbuf);

PcapPacketHandler<String> handler = new PcapPacketHandler<String>() {
  public void nextPacket(PcapPacket packet, String user) {
    System.out.println("size of packet is=" + packet.size());
  }
}

pcap.loop(10, handler, "jNetPcap rocks!");

pcap.close();
```

As mentioned in more detail in section 2.7.1, the packet is only suitable for a single iteration of the dispatch loop. If we wanted to enhance our example and put the packet on a queue, we need to copy the received temporary packet to a new memory location that is will not be de-allocated or reused until we are done with it. This is actually quiet simple to do.

We can enhance our example as follows:

```
StringBuilder errbuf = new StringBuilder();
Pcap pcap = Pcap.openOffline("tests/test-afs.pcap", errbuf);

PcapPacketHandler<Queue<PcapPacket>> handler = new PcapPacketHandler<Queue<PcapPacket>>() {
  public void nextPacket(PcapPacket packet, Queue<PcapPacket> queue) {
    PcapPacket permanent = new PcapPacket(packet);

    queue.offer(packet);
  }
}

Queue<PcapPacket> queue = new ArrayBlockingQueue<PcapPacket>();

pcap.loop(10, handler, queue);

System.out.println("we have " + queue.size() + " packets in our queue");

pcap.close();
```

We have only made 2 changes to our example. First we replaced our user type with a Queue<PcapPacket>. This provides us with a queue that we can put packets on to. The second change is we are now copying the entire packet contents into a new packet:

```
PcapPacket permanent = new PcapPacket(packet);
```

We create a new packet and we supply to its constructor a packet we want to make a copy of. This is a simply step that hides a lot of detail. First the packet's constructor gets the total size of the packet using its `PcapPacket.getTotalSize()`. This size in bytes includes the size of the capture header, packet data and decoded state structures that the packet holds. A single large memory buffer is allocated from the default `JMemoryPool` which efficiently allocates native memory for our packets, then all 3 packet components, header, state and data are copied sequentially into the buffer. So a single allocation is holding all of the packets data. The memory layout within the buffer is as follows:

```
+------------+--------------------------+-------------+
| PcapHeader | packet_state_t structure | packet data |
+------------+--------------------------+-------------+
```

Getting back to our example, with every iteration of the dispatcher loop we add a new packet onto our queue. When 10 packets have been captured, the loop exits and our `Pcap.loop()` returns. At that point we print out the number of packets on our queue which should read 10.

2.8 - Dumping captured packet to an offline file

We've covered reading packets from an offline file, what about writing captured packets to an offline file.

Libpcap library provides a mechanism for doing just that. We first open up our normal pcap capture, either online or offline, doesn't really matter, its a source of packets. Then we create a pcap dumper using `PcapDumper` class, that is associated with our pcap capture. Then we setup a handler and we pass into it our dumper which we instruct to dump every packet received.

```
StringBuilder errbuf = new StringBuilder();
String fname = "tests/test-afs.pcap";

Pcap pcap = Pcap.openOffline(fname, errbuf);

String ofile = "tmp-capture-file.cap";
PcapDumper dumper = pcap.dumpOpen(ofile); // output file

JBufferHandler<PcapDumper> dumpHandler = new JBufferHandler<PcapDumper>() {

  public void nextPacket(PcapHeader header, JBuffer buffer, PcapDumper dumper) {

    dumper.dump(header, buffer);
  }
};

pcap.loop(10, dumpHandler, dumper);

File file = new File(ofile);
System.out.printf("%s file has %d bytes in it!\n", ofile, file.length());

dumper.close(); // Won't be able to delete without explicit close
    pcap.close();
```

And that will do it.

If the only thing you want to do is dump packets to a file, jNetPcap provides a performance enhancing native dumper. This dumper dumps packets natively without entering "java" space.

```
StringBuilder errbuf = new StringBuilder();
String fname = "tests/test-afs.pcap";

Pcap pcap = Pcap.openOffline(fname, errbuf);

String ofile = "tmp-capture-file.cap";
PcapDumper dumper = pcap.dumpOpen(ofile); // output file

pcap.loop(10, dumper); // Special native dumper call to loop

File file = new File(ofile);
```

```
System.out.printf("%s file has %d bytes in it!\n", ofile, file.length());

dumper.close(); // Won't be able to delete without explicit close
    pcap.close();
```

The dumper is passed directly to `Pcap.loop` and there is no need or way to specify a user handler. A builtin native handler is provided by jNetPcap that performs the packet dump as efficiently as possible, natively.

2.9 - Transmitting packets

Another capability libpcap gives us, is the ability to transmit raw datalink packets. This can be done in two different ways. The standard way is to send each packet out one at a time. WinPcap extensions to jNetPcap, only available on win32 platforms, give us a second way were we can send out a queue of packets out all at once, much more efficiently that we otherwise can.

WinPcap is only available on win32 platforms, and you must use `WinPcap.isSupport()` call to check if win32 extensions are available on your platform, even if you are sure that they are, you must use it otherwise your code will be unreliable.

The standard way is to either use `Pcap.inject()` or `Pcap.sendPacket()` calls. You supply to the call the entire packet including the datalink (i.e. ethernet header) and it will send that packet out on currently open live network interface. Note that these calls are unavailable for offline captures.

Both these calls are also optional and you must first check if they are supported with the following accessors, `Pcap.isInjectSupported()` and `Pcap.isPacketSendSupported()`. Both of these will typically be available, but you must none the less make sure and provide code for both cases. Even 3 cases if you also want to include the win32 senqueue as well.

2.10 - Close Pcap and PcapDumper handles

When you are done with a `Pcap` object or `PcapDumper` you must call its `close()` method. This ensures that all resources being held are properly released. Especially when working with decoded packets. They allocate fair amount of native memory, outside of JRE reporting and management. It is a very good idea to release that memory when no longer needed.

Here are some specific close methods:

- `Pcap.close()`
- `PcapDumper.close()`

It is especially neccessary to close the `PcapDumper` object, as it will hold on to the open file and lock it. You will not be able to manipulate that file until the dumper is closed out.

Ch 3 - Packet Decoding

Staring with release jNetPcap version 1.2, we have a powerful packet decoding framework. A new java package `org.jnetpcap.packet` contains all the necessary classes and sub-packages for protocol definitions, packet API, formatting, and a low level scanner.

The framework is made up of 4 major parts. Each discussed in its own section.

1. `PcapPacket` and `JPacket` classes - they are the entry point to the entire decoding packet API

2. `JScanner` and `JRegistry` classes - these classes maintain a database of protocols and allow the packets to be decoded using native functions.
3. `org.jnetpcap.protocol` package - contains all the protocol definitions, grouped by protocol suites or families, written in java. Each protocol is made up of a mandatory header declaration and several optional classes such as analyzers, utility classes, events and protocol specific exceptions.
4. `org.jnetpcap.packet.format` package - provides formatters that can read in a decoded packet and produce textual output such as, a pretty plain-text dump of a packet or Xml output.

3.1 - JPacket and PcapPacket classes

The main packet classes, `PcapPacket` and `JPacket`, are the center point of the packet framework. Everything else revolves around these two classes. A packet is made up of 3 parts:

- Packet data buffer - received from libpcap or read from a file
- Packet state structure - generated by jNetPcap
- Packet capture header - also received from libpcap

All 3 parts are manipulated and maintained separately of each other. That is the capture header and packet data buffer are both supplied by libpcap, and the packet state structure maintained by `JScanner` class. These structures and the packet buffer are stored in native memory outside of java address space or byte arrays. Packet state and its data buffer are the two most important parts, as they are used in decoding and then subsequent access to individual headers.

3.1.1 - Accessing headers in a packet

Once you have a fully decoded packet you can access its state and retrieve headers. All packets delivered to `PcapPacketHandler` are fully decoded and ready for access. Most of the header accessor methods reside in the base class of `PcapPacket` which is `JPacket`.

You use `JPacket.hasHeader()` and `JPacket.getHeader()` methods to first check if a header is present in the packet, and then retrieve an instance of that header. You must always check if a header is present before trying to retrieve the header. If you do not your code will fail very quickly if not immediately. Note that checking for presence of a header is extremely efficient. The scanner, which is responsible for decoding the top-level state information, simply sets a single flag in a integer that represents a particular header. So a check is simply a single binary bitmask operation which is very very fast and efficient. If the bit is set, the header is present, if the bit is not set, the header is not there. This was a key design goal, to ensure that virtually no penalty is received for checking for presence of headers.

Once we know for a fact that there is a header we can use `JPacket.getHeader()` method to retrieve an instance of that particular header. Again, for performance reasons, the `getHeader(<? extends JHeader> header)` expects you to supply a header object of the correct type. The header retrieval is also a simple and fast operation. The packet's state contains an byte offset into the packet's buffer for the required header and its length. These are stored in an native array and are looked up very efficiently. The supplied user header is then peered with the exact memory location the header occupies. No copies of state or data actually take place. Simply a the native memory reference is set to which points at exactly where the header resides in the packet's buffer. Once peered, the header can only access its that particular memory segment, and an exception will be thrown if it tries to go out if its bounds.

There is a convenience method `JPacket.hasHeader(<? extends JHeader> header): boolean` that will combine the
normal `hasHeader()` and `getHeader()` in a single call. It will only return true when the requested header is found and when it does, the supplied user header will be peered with the physical header data. This call is very suitable for putting into java `if` statements.

Here is an example:

```
Tcp tcp = new Tcp(); // Preallocate a Tcp header

public void nextPacket(PcapPacket packet, Queue<PcapPacket>) {
  if (packet.hasHeader(tcp)) {
    System.out.println("found packet with tcp port=" + tcp.destination());
    queue.offset(new PcapPacket(packet)); // Make deep copy
  }
}
```

In this example, first we pre allocate a `Tcp` header object, found in `org.jnetpcap.packet.header` package. This is an empty header which doesn't reference any data. If we try to access any of its methods, we will immediately recieve a `NullPointerException` as the tcp variable may not be null, but the native memory location where the header resides is still set to null. It gets initialized in the handler in the `Pcap.hasHeader(tcp)` call.

We receive packets from a libpcap dispatch loop, in our handler, where we first check using the pre-allocated tcp header if the TCP header exists in the packet. The tcp header instance is uninitialized but it does supply its numerical header id, maintained and assigned by `JRegistry` so that a proper `hasHeader()` check can be performed. If the header exists, the tcp header object is peered with the packet and more specifically the packet buffer where it is supposed to reside. We then enter the "if" body. There we access one of the tcp fields, the `Tcp.source()` field and we print out its value. All fields with all headers provide an accessor method for reading the value. This saves you from having to know and understand to minutiae detail the structure and position within the structure of the tcp header. Especially when dealing with optional parts of the header.

As a last step in the "if" statement, we put a copy of the packet onto a queue. This is a deep copy of the packet. We can not put the dispatch loop supplied packet, as that is a temporary packet that could be changed or disappear at any moment. By making a deep copy of the packet, we are sure that it will persist until we not longer needed. Deep copy means that the packet data and its decoded state is copied to new more permanent memory location using the packet's default memory allocation mechanism.

3.1.2 - Accessing sub-headers in a header

So now that we know how to access any header within a packet, it is time to a peek at the `JHeader` class. If a complex header has sub-headers, such as optional headers or simply portions of the header that should be treated as a separate sub-header, we can check for their presences and acquire instances to those sub-headers.

We look up sub-headers in the parent header, not the packet. We use a method `JHeader.hasSubHeader()` does the check for us. The same way we have the convenience method in packet api, we also have a combined `hasSubHeader()` and `getSubHeader()` methods in the `JHeader` class.

A sub-header is a normal header, it actually extends the `JHeader` baseclass. So once we peer to it, we use it just like any normal header.

Here is an example using ip optional headers:

```
Ip4 ip = new Ip4();
Ip4.Timestamp ts = new Ip4.Timestamp();
Ip4.LooseSourceRoute lsroute = new Ip4.LooseSourceRoute();
Ip4.StrictSourceRoute ssroute = new Ip4.StrictSourceRoute();

JPacket packet = // We have it from some source

if (packet.hasHeader(ip) && ip.hasSubHeaders()) {
  if (ip.hasSubHeader(lsroute)) {
  }

  if (ip.hasSubHeader(ssroute)) {
  }

  if (ip.hasSubHeader(ts)) {
  }
}
```

The example leaves the if bodies empty for brevity. Lets take a closer look at what's going on.

Notice that we not only pre-allocate our Ip4 header, but also several optional headers found under `Ip4`. Those are inner classes but they are static and standalone. Their namespace is tied to Ip4 namespace on purpose. This way you can always expect to find Ip4 optional header under Ip4 namespace. This also prevents name clashes with other sub-headers with the same name but different parent headers.

The example doesn't actually show where we get the packet, it is irrelevant for the purposes of this example. It is important to know that we have a fully decoded packet.

So once the first, outter if statement, checks to see if Ip4 header is present within the packet. If it isn't there is no point in doing any other work (again, a simple single bit check). Within the same condition expression, we also check, but only if the first ip4 part succeeded, if the header has any sub-headers (any at all) set. If there aren't any subheader, we don't need to go into the rest of the logic. If it has any sub-headers set, we enter the outter if statement body. There we individual check with our already peered ip header instance, if it has specific sub-headers.

If a sub-header is found, it is also immediately peered and is ready for us to do whatever the application needs to do.

The important lesson to take from the example, that for sub-headers we are using the `JHeader` classes API with its `hasSubHeader` and `getSubHeader()` calls, not the packets. Those sub-headers theoretically could have their own sub-headers defined as well. You would simply daisy chain the check for the sub-header using sub-headers as parents, and so forth.

```
if (packet.hasHeader(ip) && ip.hasSubHeader(lsroute) && lsroute.hasSubHeaders()) {
}
```

Of course, Ip optional header, Loose Source Route, doesn't have any sub-headers, none are defined, therefore the `hasSubHeaders()` will always return false. We could however have a complex header that has multiple tiers of sub headers.

3.1.3 - Working with packet's state directly

It is possible to work with packet's decoded state directly. You can use the method `JPacket.getState()` to retrieve direct access to packet's state. It is a separate object, that is peered to a native state structure. This structure maintains information about each header, its length and offset into the packet data buffer. It also maintains a bit based integer map of the headers that are present within the packet.

The `JPacket.State` class provides native accessors for accessing members of the native `struct packet_state_t` C structure. The information in the structure is surprisingly simple. It only maintains the start and the length of the header within the packet buffer. That information is used to peer specific header instances directly with the packet's buffer memory where the header is said reside. Then its upto the header, a java class, to actually read the information from that native memory using `JBuffer` accessors.

This is an advanced topic, that is very implementation specific, therefore it is best not to rely on this part of the API.

3.1.4 - Working with packet's capture header

Pcap packets, the ones that were retrieved using libpcap library, also maintain a reference to a separate `PcapHeader` class. The baseclass `JPacket` abstracts that information away using a `JCaptureHeader` interface, but it is also possible to work directly with the `PcapHeader` class. It is peered with native C structure defined in libpcap library itself `struct pcap_pkthdr`. It provides information such as packet captured length, original length and capture timestamp.

Most user's do not need to deal with this detail and is sufficient to work with `JCaptureHeader` interface.

3.1.1 - Working with packets

As a networking concept, a packet is a collection of headers that are stacked one on top of the previous header in a contiguous buffer. The buffer is usually transmitted serially one byte or bit at a time.

In jNetPcap, a packet is exactly the same thing. Its a raw data buffer that has been delivered by the network interface in the system captured the sequence of octets, that make up the packet.

If we look at a PcapPacket class hierarchy, notice that a low level, a PcapPacket is just a raw buffer, a `JBuffer` object. You can use `JBuffer` methods to read any byte, integer or sub-buffer at any offset within the buffer to retrieve the raw data. JBuffer's always point to the start of the packet data, at offset 0. Their size is exactly the number of bytes that was captured. This is a little hard since you yourself need to know they structure of the buffer.

```
JMemory
 +-> JBuffer
     +-> JPacket
         +->PcapPacket
```

If a packet has been captured using libpcap, you will get `PcapPacket` objects, even if the API delivers them to you as a `JPacket` class. But what if you wanted to create a custom packet yourself?

You can use `JMemoryPacket` class. This class has the same super classes as PcapPacket class, but you won't be able to retrieve libpcap specific properties that you can from a `PcapPacket`. This makes sense, since we are creating our own packet. The class hierachy is identical of `JMemoryPacket` and `PcapPacket`.

```
JMemory
 +-> JBuffer
     +-> JPacket
         +->JMemoryPacket
```

3.2 - JScanner and JRegistry classes

These are complex classes that fortunately most users will never have to touch. All the default protocol headers and scanners are already registered for core protocols provided with jNetPcap.

When a packet is first created, it is a blank packet. It contains references to the data buffer, but it doesn't know about which headers exist in the packet. The packet is first scanned using a low level native scanner that starts at the beginning of the packet buffer, reading and discovering which protocol headers it knows about and if they are found.

When a header is found, information about that header is recorded within a unique packet's state structure. This is a native structure and the `PcapPacket` object is given a reference to it. No other information is actually copied into the packet itself. If the packet needs to look something up, it uses native JNI methods to retrieve that information directly from the native structures. This way structures can be simply passed around by reference both in java and native land.

`JRegistry` class maintains a database of all known protocol headers, java bindings and native and java header scanners. This information is automatically recorded whenever the registry is first accessed. This is a pure java class, that maintains the relationships between protocols.

There are only 3 circumstances that a user would actually have to access the data in this class.

- Adding a new protocol
- Extracting information from the database for display purposes such as listing various protocol to protocol bindings, etc..
- Changing the default protocol to protocol bindings

The `JScanner` class is used for decoding packets. That is, recording which headers, at what offset and how long those headers are in a packet state structure.

By default, a scanner is used to scan all incoming packets before they are dispatched to the user's `PcapPacketHandler` callback method. It is however possible to rerun the packet scan manually. This may be needed if packet content has changed. For simplicity `JPacket.scan()` method is provided that reruns the scan using the default scanner, but it is also possible to setup a different scanner manually.

3.2.1 - JRegistry in more detail

`JRegistry` class is actually quiet important, although most users will never have to deal with it.

The most common reason that anyone would have to use `JRegistry` is to change a protocol to protocol binding or adding a new user defined protocol.

To change a protocol binding, you would create a new instance of java interface `JBinding` and register it with `JRegistry` telling it which protocol is binding to which protocol. Then the scanner will know when to use your binding.

You can also register a completely new protocol. If have written a custom header definition, you can register that header with `JRegistry` which will remember it and assign it a unique numerical ID. The scanner only works with these numerical ids, indexing and setting up lookup tables etc, all based on the protocol id. New custom protocols need to be registered every time the application starts. These setting are not persistent. You may get a different numerical ID for your protocol.

3.2.2 - Overriding default bindings or adding new ones

What is a protocol binding? It is simply a piece of code that determines if protocol B's header will follow immediately protocol A's header. Lets take a look at an example:

```
+----------+----------+----------+---------+
| header A | header B | header C | Payload |
+----------+----------+----------+---------+
```

We see a packet buffer that starts with byte 0 at the left. We see that there is a protocol's A header starting with byte 0. After protocol A's header ends, protocol's B header starts, then protocol'C C header. At the end of a packet buffer, the remaining unmatched portions of the buffer to any protocol's header, are assumed to be the payload of the packet. All the other headers were simply there to facilitate the transfer of the payload and describe its contents.

So the scanner is told, literally by a numerical protocol ID, when the scan is requested what header A is. It is often called the DLT or Data Link Type or the first header within the packet. A binding is a piece of java code defined in object for by implementing the `JBinding` interaface, that at an appropriate time will be called upon by the scanner, to evaluate the packet and return weather the binding succeeded or failed.

In our example, the scanner knows exactly what header A is, we told him or more specifically libpcap library itself told the scanner. The scanner records information about the header A, and then checks if there are any java bindings to it. If there are any bindings, it iterates through the list of these java bindings, providing them access to the packet and header A and asking if the binding is valid or not.

So if header B has a binding registered with header A, that binding will be run to check if it passes. The binding code has a chance to run whatever logic and algorithm is neccessary to determin if B should immediately follow A. In our example it does. So the binding returns success. The scanner then records that B is also found, also records the starting position of B, immediately follows A, and B's length also provided by the binding or if neccessary by another support interface `JHeaderScanner` which determines the exact length in bytes of the header. Not all header's are constant in size.

Some bindings don't even require any logic at all. For example LLC header always follows IEEE 802.3 header. Therefore the LLC to 802.3 binding is very simple. It always return success.

In this way the scanner can go from header to header, check its bindings and header scanner a build a complete list of headers that were found.

Now, all the core protocols have their scanners defined natively and are scanned very efficiently and fast. Using a java bindings or scanner is less efficient, but that is not the normal case. None the less even the java binding approach is very fast as the code inside it is usually very simple.

3.3 - Working with protocol headers

We have learned so far about how to check for headers, instantiate them and peer them so that we can read useful data out of them. But what are they?

A header is a java class essientially extends the `JBuffer` base class that is setup by the framework to point exactly at the data the header needs to access. So for example when ip header is discovered in a packet, we created a working instance of one `Ip4` class, the header is peered with the packet buffer so that it only sees the part of the buffer where the header is supposed reside, that is starting at offset into the packet buffer at byte number 15 if we have a normal ethernet packet follows by ip header, which is usually the case. Ethernet header takes up 14 bytes that leave Ip4 header to start at exactly 15th byte.

The length of the ip4 header has also been already determined and the underlying `JBuffer` already bound to that portion of the buffer. Since typical Ip4 headers are exactly 20 bytes in length, when they do not contain any optional headers, our header and underlying header buffer is set to point exactly bytes 15-34 inclusive.

Let me stress this point again. Any protocol header definition has a `JBuffer` as a base class. So we could essentially just typecast any header to a `JBuffer` like this:

```
Ip4 ip = new Ip4();
if (packet.hasHeader(ip)) {
 JBuffer buffer = ip; // Automatic downcast since Ip4 extends JBuffer
}
```

If this was a normal ethernet packet as described in previous paragraph, we know that out byte 0 in our buffer is actually byte #15 into the overall packet buffer and the last byte in the buffer is #34. The buffer is bound to this range and we wouldn't be able to read any data at byte index less than 0 or greater than 34. We would see an exception thrown.

We can however read anything in between. We could do something like this, if we continue with our example:

```
Ip4 ip = new Ip4();
if (packet.hasHeader(ip)) {
 JBuffer buffer = ip; // Automatic downcast since Ip4 extends JBuffer

 if ( (buffer.getUByte(0) & 0xF0) != 0x40) {
   System.out.println("expected ip version number to be 4");
 }
 int hlen = (buffer.getUByte(0) & 0x0F) * 4;
 System.out.println("ip header length=" + hlen);
}
```

That is we are reading 0th byte out of the buffer where the first 4 bits of that value are the ip version number while the next 4 bits are header length in double words. So you can see, that any header is nothing more than a fancy `JBuffer`.

Now thinking back to our `Ip4` header, it is much more convienient to provide an accessor method to the version and hlen fields of the ip header and coding that logic into the header definition. That is exactly what happens. `Ip4` header utilizes the fact that it subclasses the `JBuffer` and simply uses the `super.getUByte(0) & 0xF0` algorithm to read the value of the version field. Same applies to `hlen()` accessor and so forth. They simply read the requested field type directly out of the buffer they occupy and return the value back to the user.

Here is the extract from `Ip4` header definition of these 2 simple fields:

```
public int version() {
  return getUByte(0) >> 4;
}

public int hlen() {
  return getUByte(0) & 0x0F;
}
```

Where the function `getUByte(int)` is defined in `JBuffer` class. All the hardwork of figuring out if the header is found in the packet, where it starts and how long it is, has already been done for you. Further more the header definition already contains all of the knowledge and know how on how to access each field for that particular header.

3.3.1 - Dynamic headers

Not all headers are all that simple. Some headers have optional sub headers and fields within them. Some headers are variable length and may contain array's of fields or structures.

As mentioned in previous section, headers extends `JBuffer` class. In actuality they extend `JHeader` class which extends `JBuffer` class. The `JHeader` is in between the actual protocol definition and the buffer from which it reads data out of. When a header is peered in the `JPacket.getHeader()` method, at the time everything has been setup for the header, a hook is called to allow the protocol do any special processing it requires.

Specifically the method `JHeader.decode()` is called. By default this method doesn't do anything for the general case, but it can be override by a protocol header and that is where the header can use any custom logic for optional headers. Since a header is a normal java class, the developer of the definition is able to create any additional methods, fields both public and private, store additional flags, etc.

Anything that will allow the protocol header to remember its dynamic state. If the header does keep private or private data in the class instance, the developer needs to make sure that this data is reset within every decode call, or peering. Remember that these header instances are reused from one packet to the next. It is essential that the header instance be reset to a known state to accurately reflect its new state with a new packet.

Dynamic headers, that have optional fields, typically offer additional accessor methods that can check if a particlar fields is present or not. Since public fields and methods are documented by javadoc just like any other class, the developer of the protocol header can include any such support methods but needs to make sure that those methods and fields are properly documented in javadocs.

3.4 - Formatting packets for text output

Sometimes you want to just take a look at what the packet looks like. You can use a packet formatter to dump as normal text the contents of the packet.

All packets have a builtin `TextFormatter` hooked onto its `JPacket.toString()` method. This text formatter uses a standard JRE `StringBuilder` as the output device, formats and dumps the contents of the packet to the string buffer and at the end does a `return buffer.toString()` to convert the contents of the string buffer to a normal string.

This is a pretty looking output to prints out full detail about every header within the packet. There are additional formatters such as `XmlFormatter`. This formatter dumps the contents of the packet in XML format. Additional formatters will be supplied in the future, and if the user develops an interesting formatter contributions back to the project are apprechiated.

Additional formatters that are planned for the future are `HtmlCSSFormatter` and `HtmlTableFormatter`. The first would generate HTML out which depends on CSS for layout, while the second formatter would generate output layed out using standard HTML tables. Both would be suitable for sending to a HTML browser for pretty output of the packet.

Now that we've seen and talked about the basics. It is inefficient to generate what sometimes can be many kilo bytes of textual output, convert that output to a long inefficient string to only send it out standard output device. The better approach is to use the `JFormater.format()` method where we supply our packet as a parameter. This way we have the option of creating a formatter that sends its output to `System.out` directly instead of intermediate buffers.

```
XmlFormatter out = new XmlFormatter(System.out);
out.format(packet);
```

And we get nice XML output sent to our console. We could have used output type. The only requirement that `JFormatter` constructor sets is the `Appendable` interface. As you may suspect both StringBuilder and Syste.out's `PrintStream` implement the `Appendable` interface. It would be a trivial task to setup a TCP socket between systems enclose the socket in a `PrintStream` implementation and pass that in to our formatter which would sent its output directly to a socket.

3.5 - Protocol Analyzers

(Deprecated API) - this part of the API is currently deprecated. It has been removed from the latest production releases and is only accessible though older releases or special software branch tagged "deprecated" in the SVN repository. Analyzer and protocol decoding features are being completely rewritten and reimplemented in the next generation API 2.x. The protocol definitions will contain all necessary components to decode and analyze their respective protocols.

Definition of Protocol: *In computing, a protocol is a convention or standard that controls or enables the connection, communication, and data transfer between computing endpoints. In its simplest form, a protocol can be defined as the rules governing the syntax, semantics, and synchronization of communication. Protocols may be implemented by hardware, software, or a combination of the two. At the lowest level, a protocol defines the behavior of a hardware connection.* (source: Wikipedia.org)

A protocol's header is essentially a static snapshot of just the structure of a header within a single packet buffer. The header only provides easy access to the data found within the contents of that particular packet. Protocols are typically more involved then a single header.

For that reason, protocols are a collection of at least 1 static header definition and any number of support classes, including analyzers. An analyzer is a special java object that is specifically written for a particular protocol. It is allowed to examine a whole stream of packets. As it analyzes each packet, any information it deems important, an analyzer will store in its own internal tables, maps, arrays, lists in whatever fashion it deems necessary. An analyzer will preserve external and persistent state according to specific protocol rules.

For example Ip4Analyzer, Ip4Sequencer and Ip4Assembler will inspect all incoming packets. All 3 are analyzers that perform a specialized task. The analyzers will specifically pay attention to packets that have Ip4 headers in them. They will analyze packets for fragments, errors, delay and timeout thresholds. They will also listen of ICMP packets which carry Ip specific error messages about timeouts, drops and other error conditions. They will keep that information in tables and allow other analyzers, from higher layer protocols, to use that information for whatever is necessary.

Analyzers will also attach "analysis" information directly to packets and headers. This information can be queried and retrieved using methods `getAnalysis()`, `hasAnalysis()` on `JPacket` and `JHeader` classes. These methods may look familiar as they follow the same pattern as the header accessor methods.

All analyzers are registered with `JRegistry` which is a repository for this type of information. There is one special analyzer called `JController`. This is the top-level analyzer that all other analyzers are registered as listeners to. That is `JController` is the main analyzer that all of the packets are send to. It then maintains a table of sub-analyzers for each protocol.

For example, Ip4 protocol analyzer registers itself with `JController` as Ip4 listener, as well as Icmp listener in order to receive ICMP based packets so it can look at the control messages being passed around that are Ip4 specific.

Analyzers provide various callback interfaces. Each analyzer can provide its own unique, or custom callback interface. There is also a standard callback interface called `AnalyzerListener` which is generic and well suited to dispatching analyzer specific events. Each analyzer generates various events, which are protocol specific, that inform listeners of various protocol level states. Such as protocol transitions, i.e. TcpAnalyzer dispatches events to notify that Tcp 3-way handshake has begun and for each subsequent state transition, all on a per stream basis. Sequencers generate events when they start to find related packets and put them in sequence. Assemblers listen to the sequencer events, prepare for reassembly and when they receive an event letting them know that sequence is complete, to reassemble the entire sequence.

Analyzers can also buffer packets. Specifically analyzers tell the main `JController` which is responsible for receiving packets as well as passing them out, when to put the outbound packets on hold. Packets at that point are buffered in input and output queues, as necessary, until the hold is released at which point the queues are drained and the user receives packets.

Analyzers keep track of time, using packet capture timestamps. That is, the current time they use, is the time that is part of the packet. This allows packets that are read from a file to be properly analyzed for timeout conditions, errors, issue warning etc. Time mode can be changed and even augmented.

Therefore analyzers provide a higher level view of the protocol, with state being maintained across multiple packets and time. Analyzers provide a higher level interface for accessing this information. For example if a user is interested only in Ip4 packets, they can register an `Ip4Handler` which will receive only Ip4 headers:

```
Ip4Analyzer ipAnalyzer = JRegistry.getAnalyzer(Ip4Analyzer.class);
ipAnalyzer.addListener(new Ip4Handler() {

  public void processIp4(Ip4 ip) {
    System.out.println(ip.toString());
  }
});
```

This particular handler is registered with Ip4Analyzer and will only receive Ip4 headers. Since you can use `JHeader.getPacket()` method to retrieve the source packet that this header belongs to, its trivial to do so, if a packet is needed. More importantly you get an already peered ip header, that has been fully analyzed. It may contain reassembled data already if this particular packet was a fragment. The reassembly would happen automatically. You could check with Ip4 header if this is a fragment and if it is, retrieve the reassembled ip datagram:

```
  private FragmentAssembly assembly = new FragmentAssembly();

  public void processIp4(Ip4 ip) {
    if (ip.hasAnalysis(assembly)) {
      /* Its a fragment that has been reassembled. Reassembled fragments are returned
       * as brand new packets, that did not exist in reality on the network, but were
       * constructed and carry the new Ip4 datagram state and reassembled data.
       */
      JPacket reassembled = assembly.getPacket();

    } else {
      // It was just a normal non fragment datagram
    }
  }
```

Note that `FragmentAssembly` class resides under `org.jnetpcap.packet.analysis` package. It is also used for Tcp segmentation reassembly. `FragmentAssembly` is reused for various protocols that perform reassembly work. A different type of reassembled packet will be returned for Tcp protocol, one that contains reassembled tcp headers. The reassembled packets may be very large, especially in the tcp case. Ip datagrams are typically limited to under 64K by the protocol definition itself. Ip fragments are typically much smaller then 64k ceiling, such as 8K for NFS under Udp protocol.

Ch 4 - Internals

In order to design proper applications that rely on certain features of the API or do things a certain way, it is sometimes essential to know a little bit how things work underneath the hood.

As you may already know, jNetPcap utilizes some native platform capabilities behind the scenes. It is also limited by the capabilities and performance of the libpcap implementation on the underlying platform. These capabilities and performance may vary greatly depending on which platform the jNetPcap is used on. This is kind of a given with any multiplatform library, but I repeat it here none the less.

What jNetPcap strives to accomplish is to expose as much capabilities as possible at fastest possible speed. Here are some note worthy highlighs of the inovations of jNetPcap library:

- No packet data copies between native land and java land
- Packet scanner/decoder is implemented mostly in native code and utilizes native C structures for decoding header for all of the core protocols.

- The scanner and libpcap both use large preallocated working buffer to store packet data and decoded packet state.
- The all important `JPacket.hasHeader()` and `JPacket.getHeader()` methods are implemented extremely efficiently. The `HasHeader()` is a bitwise OR of a single integer, which can be done very efficiently in both native and java space. Since these checks are to be found everywhere, especially in main loops, this fact is very important to note.

4.1 - Packet data buffer copies

jNetPcap does everything possible to avoid have to copy the packet data. So when your packet handler receives a buffer or a packet object from jNetPcap library you know that no data copies were done by jNetPcap itself. The data is passed into java by reference only.

If the underlying platform's kernel supports no packet copies, then that packet potentially arrives to your java handler/callback directly out of the memory location where the physical network interface placed it. Unfortunately this type of information is currently programatically inaccessible.

However if the application is capable of processing the packet immediately, then that packet can be discarded without having to make any copies. This is not always the case and sometimes the packet needs to be copied out of libpcap controlled buffer into more permanent user controlled space. This can be done very efficiently using low level native platforms copy functions. You just use any existing `JNetPcap.transferTo()` methods. The library also provides a custom `JMemoryPool` designed to allocate and space for this purpose alone, copying packet and state data as efficiently as possible.

Even the packet and buffer java objects, are reused where possible. The same buffer and packet instance is reused by peering to new memory location when a new packet is received. The user gets a fully functioning packet or buffer object, the same one he may have received before, that is now pointing at the next packet. The implemented literally reuses the exact same packet, not a cache of packets. Every set of the dispatch loop with calls `Pcap.loop()` or `Pcap.dispatch()` allocates a new java buffer or packet object to be used by that loop. To dispatch to the user using that java object.

Therefore the decision to reallocate memory, do deep copies and so forth is completely left up to the user to do at the appropriate time. Since libpcap and the scanner can supply more than one packet's buffer full, the user can also choose to peer a new buffer or packet object without doing any deep copies. That is a new java object would be allocated and would take over the references from the dispatch buffer or packet object. This way only a single java object needs to be created to peer with the packet buffer without data copies. This allows a packet or buffer to persist longer than a single interation of the dispatch loop, although since we do not know exactly when libpcap buffer is wrapped around and when any of our references we are holding to that buffers internal memory become unusable, it is typically safer to always reallocate and copy data into a new buffer. None the less it is possible to calculate the amount of time or number of packets that can fit into a libpcap and scanner buffers and possible tune packet queues to match or come in under those numbers allowing packets to be queued while still leaving the packet data in original buffers.

4.2 - Scanner performance

The API for decoding packets has been designed from the ground up, to be able to utilize a native protocol scanner. `JScanner` class is a control class for the native scanner. Native in the sense that most of the logic of scanning a packet data buffer is implemented in native code.

The scanner has a `scan()` function written in C that scans the packet data using a function dispatch table. The table contains a scan function for every core protocol. The scan function looks up 2 things, the length of the header and the next protocol header that after it. The numerical protocol ID serves as the index into an array of these scan functions. Array lookups are much efficient in C than they are in java. Overall the scan loop is tight and efficient.

The information returned from a scan function is recorded a packet state structure which contains a list of header state structures. Only the offset and length are recorded on a per header basis. The scanner uses a fairly large internal buffer to store these state structures in. A new packet is assigned the next state structure until the scanner runs out of buffer space. Then it wraps around to the beginning of the buffer. So no mallocs and frees are necessary on a per packet basis. Only the the scanner itself is initialized or garbage collected.

The numerical ID assigned to each protocol is often used as an index into bit arrays and lookup tables. `JScanner` in conjunction with `JRegistry` which makes sure that each protocol header gets a unique numerical ID, make it possible to optimize protocol lookups and other operations.

4.3 - Scanner Internals

`JScanner` class is the front end to a small library of functionality implemented native code (C/C++ language). The scanner can be invoked from both native code and from java space. For example the `PcapPacketHandler.nextPacket` method, before it is invoked by native libpcap dispatcher, it first invokes the scanner before transferring control to user's java handler. At the same time, a newly created packet using `JMemoryPacket` can invoke the scanner from java space using `JMemoryPacket.scan` which will invoke the default packet scanner for the currently active thread.

The scanner performs several functions:

- discovery of main headers in the packet
- discovery of the next header in the chain of headers using direct binding method
- invokes a java bind method
- invokes a java headerLength method defined in a header
- dissects the contents of a header. Actually maintains 2 separate tables, one for dissecting sub-headers and the second for dissecting optional fields.

4.3.1 - Quickscan

The most important function of the scanner is to perform a quick scan of the packet buffer and record information about what headers are found and at what location within the buffer. This information is then associated with a java packet or more specifically `JPacket` and `JPacket.State` classes. The first provides access to packet data buffer and the second to scanner state structures for packet. There is also a `JHeader.State` class which provides access to each individual's header's state. `JHeader` itself is peered (linked) with native memory location that is the packet buffer, but specifically at offset into the buffer that points to the start of the each header.

Ch 5 - Protocols

To create a custom header definition you need to define 2 things:

1. Header class file
2. Header to Header binding

A header definition before it can be used, has to be registered with `JRegistry`. Once registered the class can be used like any other header file.

Lets take a look at the most basic possible header definition. This one has no headers, its 0 length, no bindings defined to other headers but is a complete header definition that can be registered.

```
@Header(length=0)
public class ZeroLengthHeader extends JHeader {
}
```

What else can we put in a header file? The only mandatory annotation is the @Header applied to a java class. All the rest of annotations are optional. Most of the remaining annotations are only there to help `JFormatter` to generate textual output. That is they are there to convert a header definition with its runtime information to a hierarchy of header/field objects that can be generically inspected or printed out.

Header files can contain anything any normal class file can contain. Since the programmer has direct access to all the public functions of a header file, it is really up to the header definition designer as to what to put in the file, which fields to export through accessor methods, etc. Constants, protocol specific utility methods, static and instance methods. Anything can be placed inside a header definition.

There are some rules when you start to apply annotations. Lets look at a slightly more practical header example and go through each of its elements. This example uses standard Ethernet header definition, since it is one of the simpler definitions supplied with jNetPcpap.

```
@Header(length=14)
public class Ethernet extends JHeader {

  @Field(offset = 0, length = 48, format = "#mac#", description = "destination MAC address")
  public byte[] destination() {
    return super.getByteArray(0, 6);
  }

  @Field(offset = 6 * 8, length = 48, format = "#mac#", description = "source MAC address")
  public byte[] source() {
    return super.getByteArray(6, 6);
  }

  @Field(offset = 12 * 8, length = 16, format = "%d")
  public int type() {
    return super.getUShort(12);
  }
}
```

This is a complete header definition for Ethernet protocol. @Field annotation marks a method as a header field. The name of the method acts as the name of the field. The @Field annotation takes several parameters, all of which are optional. The offset and length are offset into the buffer starting at start of this header. The length fields specifies how long this field is. Both values are in bits. The description parameter is a static description string that will be printed out along side the field value. Parameter format specifies how to format the returned data for output. There are a number of predefined value formats.

So this definition has 3 fields. This was easy since all the fields are static in length and always appear at the same offset into the header. The header is also of static length, 14 bytes.

Of course, not all headers are that simple. Some headers are variable in length, have optional sub headers and fields. Fields have sub fields as well.

5.1 - The Basics

jNetPcap header definitions, describe in java language the exact structure of one of those headers in a protocol suite. There may be many different headers in a single packet, but each one of those header occupies its own certain portion of the packet (packet = data buffer).

5.1.1 - What is a network packet and a header

What is a header?

From the network theory point of view, a protocol header contains information that is specific for that protocol. A header is a concise, structured information that is transmitted between one or more applications on the same or different host computers or other devices. While a protocol as a whole, may be comprised of more than one header type that it uses for communication purposes to exchange information such as state and transmit application specific data. This is the reason why sometimes a protocol is described as a protocol suite. A collection of related headers that facilitate exchange of data for that protocol.

What's inside those headers is usually integers, strings and some times incomprehensible mubo jumbo that only the application the header is intended for can understand. Its a collection of variable's values all stored as raw bytes in that portion of the data buffer, our packet. The header helps to break down those raw bytes into meaningful, structured data set. We call those variables in the header *fields*.

What is a packet?

A packet is a buffer of certain size, full of data, that is transmitted from one machine to another. This data buffer is transmitted differently depending on which "data-link" network you are using, such as ethernet, FDDI, token-ring, WAN point to point links, etc. The end result is the same, the entire bundle we call a buffer, is transmitted from one machine to another and arrives at the destination intact (hopefully). The receiving machine if this packet sees the packet as a data buffer of certain length. This data buffer starts at certain memory location and ends at certain memory location. Offset 0 into this data buffer represents the first byte of the packet. And the last byte in the buffer is exactly at offset + length of packet.

jNetPcap's header definitions take care of the mundane details of how to reconstruct data out of this raw soup of bytes we found in the packet, into integers, int based flags, strings, Ip addresses, Mac addresses and dozen other details you would have to do by hand yourself. The header definition that you use has accessor methods for each and every field you will find in the header's data buffer. You use normal java methods, that are named after the name of the field to read and write values to or from that field. The name of the fields are usually defined in documents that describe the header structure such as RFCs. The names are usually fairely consistent with the standards that define those headers.

There is more. A packet buffer, the data buffer that contains the entire packet, consists of many headers, usually one right after the other starting at byte offset 0 into the buffer. Each one of those headers is different from the previous one. The previous header usually has a field that determines exactly what header follows right after it. Another words, the header within a packet are like a daisy chain of protocol headers:

```
+----------+-----+-----+------+------+
| Ethernet | Ip4 | Tcp | Http | Html |
+----------+-----+-----+------+------+
```

This is a diagram of a typical packet you will find. This particular packet transmitted our ultimate HTML text document that our browsers can read and display, but there was a lot more going on behind the scene. Just look at all those other types of headers found in the packet. They were all needed to transmit that packet across the network between web browser machine and the web server.

Note that this is a daisy-chain of headers. Ethernet has a field at toward the end of the header called *type*. This is a 2 byte unsigned short integer that holds a numerical value. That value is the id of the next protocol header right after Ethernet. The Ip4 header has a reserved id value of 0x800 for it. So anyone reading that packet has to look into that *type* field in order to determine the next protocol. When he does, 0x800 in that field tells him the next header is Ip4. Then Ip4 header also has a *type* field, in a different location of course, that tells you what the next header is, this type its 6 for Tcp. Tcp reserved with the Ip4 controlling body numer 6 for itself (this was done ages ago). No one else can use 6 for Ip4.type field. Tcp has source and destination port numbers, but same concept, those fields tell what the next header is. 80 is usually reserved for HTTP, but that is actually web server and client configurable. Lastly the Http header contains a field, which is actually text based, a string, that tells what the next header is. In our case that would be html.

This is another major and difficult part of interpreting the data within a packet. In order to know what is stored at the end of the packet, the payload, or the last header and its data, you have to decode or go through the daisy chain of headers, one after the other until you reach the end, in order to figure out what is the last header. Then if you saw that last header was Html you could sent that raw data to your own web browser to see the web page transmitted. Going through this daisy chain, also means that you have to understand everyone of those headers, so you can figure out what the next header is. Also if any of the previous headers threw any quircks into this whole complex mess. Ip4 can fragment everything after it, so you may only be at one Ip4 packet fragment out of several. This means your next header is probably not complete. (For those that are going to jump on me for this, Tcp protocol always sets the DF flag so for Tcp traffic Ip4 will never fragment - there happy 🙂

That is where *jNetPcap* header definitions come in. What the API does is it scans that packet, going through this daisy chain of headers and discovering all the headers that are present in the packet. It learns exactly what type of header it is and where each header begins and ends within the packet. All that information is stored inside a java `JPacket` object. You can use `JPacket` to skip over headers and simply ask, is there a Html header in this packet, if yes I'm interested, otherwise lets not do anything with this packet and look at the next packet. Then you can go a step further. You can create an empty Html header, this is a plain old java file, and tell the packet to position the header in the right place so you can read data using that header. Any field methods you invoke, read their values directly out of the packet at the right place, and return that data to you as ints, strings, byte[] whatever that data type is and what ever the header's accessor method for that field signature looks like.

5.1.2 - jNetPcap's JPacket and JHeader classes

In *jNetPcap* a header definition is a plain old java class file. It is written completely in java language and compiled with a javac compiler. In order to use a header file you have to first put it in your classpath so that JRE (java runtime environment) can find it, load it and make it available to java VM.

jNetPcap comes with a new type of java buffer object called `JBuffer`. This buffer class provides very similar accessor methods as the normal JRE's `java.nio.ByteBuffer` class. You can read and write different type of java primitives directly from this ByteBuffer. `JBuffer` is almost identical in that it provides the same type of accessor methods for reading java primitives and a few more for reading unsigned integers which ByteBuffer class does not. But that's not why that class has been included with the API. The real reason is that it can pointed at any memory location in the system without having to make memory copies. ByteBuffer doesn't have that flexibility but `JBuffer` does.

A header definition file is a normal java class file that extends `JHeader` class which in turn extends `JBuffer` class we just discussed. Lets take a look at some class hierarchies to better understand this inheritance tree:

```
jMemory
+-> JBuffer
    +-> JHeader
    |   +-> Ethernet
    |   +-> Ip4
    |   +-> Tcp
    |   +-> Http
    |   +-> Html
    |
    +-> JPacket
```

Notice that all the headers and the packet, all extend `JBuffer` one way or the other. This is a key point understand when writing header definitions. Everything extends JBuffer. The diagram is a static class structure, so each one of those leaf classes in the diagram becomes its own object. You are usually given the `JPacket` object from dispatch or loop methods. You are not given the header objects. You instantiate yourself the header classes as objects. You can create as may header objects as you want but you won't be able to use them until they are peered with the packet.

Peering is a `jNetPcap` concept where, based on the `JMemory` class, any memory location can simply be point at. For headers and packets this is critical. We receive a packet that points at a memory buffer (this is real native memory, not java byte arrays). This packet has been peered with the physical memory, or repointed to like a C memory pointer can be repositioned in C language. The key concept to understand is that peer means no copy, its changing the memory reference only. The `JBuffer` that `JPacket` extends, is now pointing exactly at the start of the packet data and the buffer's length is set to the length of the packet. The buffer is not allowed to read any data outside of those bounds or you will get an exception. Another thing to notice if you down typecast the `JPacket` class to `JBuffer`, you have access to the complete packet buffer as a raw buffer. You would have to read data out of this buffer yourself.

We usually want to work with JPacket and not with JBuffer because JPacket gives us more, a lot more. It knows about about the structure of the data inside the packet. It knows what headers are there, where they begin and exactly where they end. The packet can take one of your header objects you instantiated and peer it with the buffer. Again change which memory location the buffer references. It will peer it in such a way, so that the `JBuffer` portion of the header is positioned exactly at the start of the header and its length is set to that of the header. So inside the header definition, it subclasses the `JBuffer` class and all its primitive accessor methods simply need to be referenced with `super` keyword. If you try and read data outside of the bound of the header, the buffer will throw an exception. So byte at offset 0 in the header is the first byte of the header, but possibly way inside the packet. You don't care about that anymore, since you know that your header is positioned properly by the packet.

Lets go back to our previous example for a second and take a look at this graphically:

```
0               14        34       42       86  <= offsets into the packet
 +--------------+---------+--------+---------+
 | Ethernet(14) | Ip4(20) | Udp(8) |data(44) |  <= Header name (length in bytes)
 +--------------+---------+--------+---------+
0             13 0      19 0      7 0      43   <= offsets into each header
```

This is our packet, a packet buffer. This time we have byte offsets for each header. On top we have the byte offsets if we were to use `JPacket` offsets, while on the bottom when we use our `JHeader` based definitions. The packet references the entire buffer, but each header only a small portion of the overall buffer. The offset start at 0 byte for each header. Notice that each header is positioned at different packet offset into the packet buffer. Once peered though, the offset for each header starts at 0.

You are supplied with a certain set of headers part of the API, but you can also add you own header files.

5.2 - Simple Definition

A simple header is one that has fields that are at constant length and offset into the header. That is, you don't have to check flags to see if a field is present in the header or calculate offset of a field based on some other fields and conditions. Complex headers are covered in the next section.

Ethernet header is one of those fairly simple definitions that we can use as an example. Its perfect because besides the read/write methods it also contains other useful information that a programmer may want, specifically EtherTypes. EtherTypes is a table of predefined values that *ethernet* and a host of other protocol definitions use as id for the next header in packet's daisy-chain of headers. This is a useful table, that has no bearing on the header itself, but is useful information that is bundled with the Ethernet header as an enum table enclosed inside Ethernet class. We'll get to that later.

```
@Header(length=14)
public class Ethernet extends JHeader {

  @Field(offset=0, length=48, description="destination MAC address")
  public byte[] destination() {
    return super.getByteArray(0, 6); // Offset 0, length 6 bytes
  }

  public byte[] destinationToByteArray(byte[] storage) {
    return super.getByteArray(0, storage); // Offset 0, length 6 bytes
  }

  @Field(offset=48, length=48, description="sourceMAC address")
  public byte[] source() {
    return super.getByteArray(6, 6); // Offset 6, length 6 bytes
  }

  @Field(offset=96, length=16) address")
  public int type() {
    return super.getUShort(12); // Offset 12, length 2 bytes
  }
}
```

Lets go through this example line by line and see what we have there.

Line #1, is a java annotation named `@Header` and one `length=14` parameter. @Header defines this as a header definition of static length of 14 bytes. That is Ethernet header is always 14 bytes long.

Line #2, notice `Ethernet` is a subclass of `JHeader` which in turn is a subclass of `JBuffer`. This is what allows us to use those "super" methods. They are all defined in `JBuffer` class.

Line #4, `@Field` annotation that tells the header that this method is used to read values out of a header's field. It provides some information about the field within the header as well. Offset is the exact bit offset from the start of the header where this field starts (you have bit precision here), the length is length of the field in bits as well. The description parameter provides a static description of the field. The description is printed out along side the field when you dump a packet to textual format.

Line #5, is our main field getter method. It returns a byte[] and takes no parameters. The name of the method is also the name of our field. The method's name is actually scanned from the class file itself and used as field name when dumping textual output. The name otherwise is irrelevant for reading data out of the header by a user. Its just a plain old method you use in your program to read that MAC address. Field names are like header IDs. They are unique names that identify a field. There may be other methods within the header all related to a field and the name is the ID that is used to group them all together.

Line #6, one of many getter methods provided by `JBuffer` base class. This particular method, reads 6 bytes out of the buffer at offset 0 and returns it as a newly created byte[]. `JBuffer` methods use bytes for offsets and lengths.

Line #9, notice this method is not marked with `@Field` annotation. This means it won't be included in textual output (ie. JPacket.toString()). It also reads the same MAC address our previous method did. The difference is that it allows the user of this method to supply a `byte[]` buffer for storing the MAC address instead of allocating a new array. This way, you can supply and reuse the same byte[] for address retrieval. You can put anything inside a header file that is useful to the user. This `super.getByteArray()`

method returns the same `byte[]` that was passed into it.

Lines 13-16, same as our destination field. The only things that changed are the name of the accessor thus a different field name, offsets and description. Again, `@Field` offsets and length parameters are in bits. The `JBuffer` getter methods take bytes.

Line 18 & 19 declares another field, name taken from the method on #19. Offset is updated to 96 bits or 12 bytes into the header and this time the length is 16 bits, an unsigned short. We also use a `JBuffer.getUShort(int offset)` method which returns an unsigned short (java type int).

Hint: the annotation is java language and its parameters take constants. Therefore you can use expressions such as `@Field(offset = 12 * 8, length = 6 * 8)` to make it easier to keep track of offsets and lengths in bytes. You will see these simple expressions sprinkled through out the CORE protocol definitions. You can even use predefined constant variables that are `public final static`.

That is the entire working header definition file. Notice that all the annotations are there only for this textual output. The methods themselves is what gives a regular programmer access to header's structure, the data. The @Field annotations are there to give `jNetPcap` runtime ability to extract certain things out of the header so it can make pretty output for us such as the one of an actual Ethernet header.

```
Ethernet:  ******* Ethernet (Eth) offset=0 length=14
Ethernet:
Ethernet:      destination = 00-E0-F9-CC-18-00
Ethernet:           source = 00-60-08-9F-B1-F3
Ethernet:         protocol = 0x800 (2048) [ip version 4]
Ethernet:
```

The output is slightly different than our simple header definition would deliver, the difference is that the production Ethernet header definition sets some extra parameters such as @Header(nicname="Eth").

All this was generated automatically for us from our header definition. Notice the interesting "[ip version 4]" description string that is on the protocol line. This description can't be static, as that field can have many different values there. The description is protocol number specific. That had to be looked up somehow. How does the real Ethernet header definition do the lookup that results in user friendly string that says "ip version 4". Hmm, lets take a look.

Everything we defined in our Ethernet example so far is static. Offsets, descriptions, length, everything. But what if we wanted to include a dynamic description, that changes depending on what's in the protocol field.

The answer is use of runtime description. A static description is stored within the Field's static state, it never changes. But we can replace that static state for descrption with method that will be used to get the description string. That method will be called whenever a description string is needed. It can return whatever we want. The method is also right in the header definition, it has access to the entire header just like any other field accessor method. Therefore it can make decisions and do look ups. It could even go off a look something up on the internet if we wanted it to.

To setup a runtime description for this field we need to add a new method and mark it with @Dynamic annotation.

```
@Header(length=14)
public class Ethernet extends JHeader {

  @Field(offset=0, length=48, description="destination MAC address")
  public byte[] destination() {
    return super.getByteArray(0, 6); // Offset 0, length 6 bytes
  }

  public byte[] destinationToByteArray(byte[] storage) {
    return super.getByteArray(0, storage); // Offset 0, length 6 bytes
  }

  @Field(offset=48, length=48, description="sourceMAC address")
  public byte[] source() {
    return super.getByteArray(6, 6); // Offset 6, length 6 bytes
  }

  @Field(offset=96, length=16) address")
  public int type() {
    return super.getUShort(12); // Offset 12, length 2 bytes
  }

  @Dynamic(Field.Property.DESCRIPTION)
  public String typeDescription() {
    return (type() == 0x800)?"ip version 4":null;
  }
}
```

Line #23, @Dynamic annotation marks the method as a runtime method that is associated with a field. The type of function it performs is Field.Property.DESCRIPTION, another words it returns a description of the field as a string. Whenever the runtime wants a description of the field to include in its output it calls this method. The user can too call this method and will be a user friendly description of the field's value, its a normal public method that returns a string.

Line #24, the name of the method is significant in this case. We used our field name "type" and appended to it "Description" to get "typeDescription". The runtime is able to figure out the field name from this as "type". So this description method is associated with type field. `@Dynamic(field="type", value=Field.Property.DESCRIPTION)` would be another way to do exactly same thing but explicitly, instead of implicitly.

Line #25, what ever we want. In this case we read the type() field, using already existing access and check if it returns 0x800. If it does we return a user friedly string, other null which tells the formatter that there is no description doesn't make any output.

The real Ethernet header definition contains an enum table called EthernetTypes. The description method actually uses that enum table to lookup values and return descriptions. The actual lookup code looks like this:

```
return EthernetType.toString(this.type()); // Lookup in enum table
```

This is a normal enum table. Again you can create the tables and other support classes however you like to make the lookup or any other operation you need to do you like. Its up to you.

Other @Dynamic types that you can define. Just keep in mind they each may return a different type, but none of the take a parameter.

- CHECK - a dynamic method that checks if an optional field is actually avaiable
- OFFSET - if offset of a field is not constant, this method calculates this offset
- LENGTH - if length of a field is not constant, this method can calculate that length
- VALUE - you can even override the method that returns the value of the field
- DESCRIPTION - dynamic value description
- MASK - a bit mask used in displaying sub-fields
- DISPLAY - used to override the name of the field, only for display purposes

5.3 - The header's length

This is a property that the library runtime requires for every header. It is an essential piece of information that is very very header specific. For some headers the length is constant, never changing and for others it is dynamic, can only be determined when from the packet's data.

Both cases have to be addressed.

Constant length headers

If the header's length is constant and is known ahead of time you simply declare it in the @Header(length = [constant]) annotation by specifying the length parameter and that it.

```
@Header(length=20)
public class MyHeader {
 /* fields go here */
}
```

Here is a constant length header. The length of this header is 20 bytes.

Dynamic length headers

For dynamic header you must include a static method within the header that returns the header's length. Here is an example:

```
@Header
public class MyHeader {

  @HeaderLength
  public static int headerLength(JBuffer buffer, int offset) {
    return (super.getUByte(offset) & 0x0F) * 4; // Ip4 style hlen
  }
}
```

The name of the method is irrelevant, but for the sake of consistency please use the name "headerLength". The method is supplied with a buffer and offset within the buffer the header starts. Its up to your method to read and calculate the right header length. The example is a type of headerLength that Ip4 would use. Thas is read the hlen field and multiply it 4.

There is a reason why the header length getter method is static and why it only receives a raw `JBuffer` to work with. Both reasons are related. Having a static method that can figure out the length of a header directly out of the raw packet buffer buffer is very useful. It means that given a buffer of octets from any source, and as long as the offset of the header into this buffer is known, this method can determine the exact boundary within the buffer that the header occupies. It also means that the method can be called on from any place within jNetPcap, even low level part that only has access to a buffer. The method is normally called by the packet scanner when its discovering which headers exist in the packet. Although the scanner may already know, based on type value of the previous header, what the next header may be, it doesn't know the actual header boundary yet. That is actually the reason why its calling on this header length method. Once the length is determined then it knows how to peer a java header object with the packet buffer, it has the offset and length of the header.

Core protocol's are scanned by the native packet scanner. This scanner is able to peek into header's on its own and lookup header lengths. So core protocols, don't actually call on these static methods. Custom protocols that are not core protocols though, do need to provide this method. Once a header definition is registered with `JRegistry`, the header length method is noted and made available the scanner. The scanner will call on your static headerLength method at the appropriate time, to determine the length of the header.

Lastly a note about sub-headers. jNetPcap header definition can contain sub headers. That is, top level header may contain optional sub headers or even always present sub-headers. Sub header's are not scanned by packet scanner, but are actually determined at the time the top-level header is accessed for the first time. When a user peers a header to a packet, the peering process allows each header to decode any optional or complex portions of itself. It is then that sub-headers are scanned, by the header definition itself. Therefore all sub-headers will have to provide either the static header length in annotation form or a dynamic method for retrieving the header's length. This may be optimized differently in the future, making the header length optional.

5.4 - Annotations

The header definition file not only is a subclass of `JHeader` but must also use some annotations. Annotations are there to mark certain methods as being special and supply some static parameters. Here is a complete list of all annotations that can be used in header definition and support classes.

- **@Header** - marks a class as being a header, may specify nicname and static length of the header
- **@HeaderLength** - if header length is not static, a method can be specified that will calculate and return the length of the header at runtime
- **@Field** - marks a field. This annotation actually creates a field and a field value getter. It also allows several field properties to be specified statically.
- **@Dynamic** - marks a method as one providing some kind of field property dynamically at runtime. For example the field's description property can be returned from a method dynamically instead of being supplied statically.
- **@FieldSetter** - marks a method as one that changes the value of the field within the header.
- **@Bind** - protocol to protocol binding. The static method checks if source protocol is bound to a target protocol and returns a boolean flag.
- **@Scanner** - allows forward bindings to occur. The scanner method overrides the actual protocol scan method natively and allows a java counter part to replace it. Its more complex but much more powerful and efficient way to do protocol to protocol bindings and header length discovery, all in one step.

5.5 - Writing a protocol to protocol binding

There are 4 ways that you can write a protocol to protocol binding:

1. declare a static method in a header definition annotated with @Bind annotation must only provide the "to=" parameter
2. declare a static method in any class file annotated with @Bind annotation, but must also provide the "from=" parameter
3. declare an instance method (non-static) in an anonymous class that extends Object annotated with @Bind annotation, must provide both "to=" and "from=" parameters
4. Create a JBinding object directly by extending AbstractBinding class and implementing its `isBound(JPacket, JHeader):boolean` abstract method.

The first 3 use other classes as a surrogates for our binding methods. The methods are invoke directly not by accessing the class they were declared in. This allows a single class to host any number of bindings. They are simply static methods that have @Bind annotation.

After you have written your bindings, they need to be registered with `JRegistry` which manages protocol to protocol bindings. Bindings that are contained within a protocol header (see case #1 below) will automatically be registered when that protocol is registered with `JRegistry`, so no need to register it again. For other cases the class or the anonymous object needs to be passed to `JRegistry` which will find the binding methods and register them with appropriate protocol headers.

Lets take a look at each one these declarations.

#1 - declare a static method in a header definition

```
@Header
public class MyHeader extends JHeader {

  @Field(offset = 0, length = 8)
  public int fieldA() {
    return super.getUByte(0);
  }

  @Bind(to = Ip4.class)
  public static boolean bindToIp4(JPacket packet, Ip4 ip) {
    return ip.type() == 0x100; // Our dummy protocol number
  }

  @Bind(to = Ethernet.class)
  public static boolean bindToEthernet(JPacket packet, Ethernet eth) {
    return eth.type() == 0x200; // Our dummy protocol number
  }
}
```

We declare a simple header with 1 field in it named "fieldA". Then we declare 2 bindings that bind our protocol to Ip4 and then Ethernet.

#2 - declare a static method in any classfile

```
public class MyBindings {

  @Bind(from = MyHeader.class, to = Ip4.class)
  public static boolean bindMyClassToIp4(JPacket packet, Ip4 ip) {
    return ip.type() == 0x100; // Our dummy protocol number
  }

  @Bind(from = MyHeader.class, to = Ethernet.class)
  public static boolean bindMyClassToEthernet(JPacket packet, Ethernet eth) {
    return eth.type() == 0x200; // Our dummy protocol number
  }
}
```

Here we drop any header definition stuff and simply create a class file with 2 static methods. The annotation and method signature allow them to be used as binding methods. Notice we had to include an extra parameter to the annotation, the "from" parameter. In case #1, the runtime is able to determine which header the binding is written in and thus the from or source of the binding. In #2 you have to specify which headers you are binding. The from parameter can be different for every method.

#3 - declare an instance method in annonymous class

```
Object myBindings = new Object() {

  @SuppressWarnings("unused")
  @Bind(from = MyHeader.class, to = Ip4.class)
  public boolean bindMyClassToIp4(JPacket packet, Ip4 ip) {
    return ip.type() == 0x100; // Our dummy protocol number
  }

  @SuppressWarnings("unused")
  @Bind(from = MyHeader.class, to = Ethernet.class)
  public boolean bindMyClassToEthernet(JPacket packet, Ethernet eth) {
    return eth.type() == 0x200; // Our dummy protocol number
  }
}
```

The only difference between #3 and #2 is that we dropped the static modifier from method's signatures.

They are instance methods, vs. static methods in previous case. We are now instantiating an object that has these two seemingly unreachable methods. That is why we are also adding the @SupressWarnings annotation so we don't get those java compiler warnings. Those methods are reachable from library's runtime and will be executed in object context at the appropriate time.

You should also note that only Object can be used for annonymous class extension purpose. Although the java compiler will allow you to extend any java class and add your own anonymous methods, the library's runtime checks the that the direct super class is the Object.class and an error is generated. The reason for this restriction is that since these are instance methods (part of a running object) I want to ensure that no one tries to access internal state. For example if someone were to extends JHeader and add annonymous methods to it (just like in the example above), then it would be very temping to try and read something out of the underlying header, if the user can even determine what header it is. This is just too ungly and only asks for trouble. So its not allowed by the runtime.

#4 - Create a JBinding object directly

```
JBinding bindIp4ToEthernet = new AbstractBinding<Ethernet>(Ip4.class, Ethernet.class) {

  @Override
  public boolean isBound(JPacket packet, Ethernet header) {
    return header.type() == 0x800;
  }

};
```

This is the more traditional way of creating a binding object. The method overrides an abstract method defined in AbstractBinding. This method is called at the appropriate time to find out if source protocol is bound to target protocol. Alternative would be actually implement the `JBinding` interface and all its declared methods, but its easier to use the adaptor class `AbstractBinding` since it does all the mundate and repetitive tasks for you.

What future may hold?

I'm actually considering one more binding type, one that utilizes the power of libpcap filtering expressions. This type of binding would allow the user to supply a filter expression as a string, have it compile to BPF byte code and executed within the right packet context to make a binding decision. That is currently on the drawing board, but can't say when this will become reality.

How to register bindings, now that you have them

Now that you have your bindings, you need to register them with `JRegistry`. The bindings defined in `JHeader` classes or subclasses are automatically added to the registry when the protocol header is registered as well. The remaining you have to register manually yourself. There are 3 static methods in registry all with the same name, but have different overrides `addBindings`. You supply either the Object instance, class with bindings in it or an array of JBinding[] objects. Each binding knows which is the source class and the target and the registry associates them with the right header. That is, you supply from case #2, #3 or #4 directly to `JRegistry.addBinding`. Case #1 is automatic.

The registry process will check the signature of your binding methods and will throw exceptions if something is not right. The 2 most common mistakes being made are forgetting to make the method "static" or not using the matching formal parameter in the binding function. The `Bind(to=protocol_class)` parameter used must be the same "protocol_class" used in the method signature. This is a condition that java compiler will not detect. It will only be detected at runtime when the method is being registered with `JRegistry`. It is at that time that the binding annotations are inspected as well as the methods signature for any errors.

How to debug JRegistry

Once you register your new bindings with `JRegistry`, you may want to verify how things are connected now. For this purpose you can use a debug method with `JRegistry` class and display its output:

```
System.out.println(JRegistry.toDebugString());
```

Notice `JRegistry.toDebugString():String` is a static method that returns a string. The string has various tables that the registry holds. The first table, displays information about each protocol registered and its bindings.

Here is an example of output from `JRegistry.toDebugString()`. Output from your instance of `JRegistry` may look a little bit different:

```
scanner[0 ] class=Payload          id= 0, loaded=true  direct=true , scan=false bindings=0  []
scanner[1 ] class=Ethernet         id= 1, loaded=true  direct=true , scan=false bindings=1  [MyHe
scanner[2 ] class=Ip4              id= 2, loaded=true  direct=false, scan=true  bindings=0  []
scanner[3 ] class=Ip6              id= 3, loaded=true  direct=true , scan=false bindings=0  []
scanner[4 ] class=Tcp              id= 4, loaded=true  direct=true , scan=false bindings=0  []
scanner[5 ] class=Udp              id= 5, loaded=true  direct=true , scan=false bindings=0  []
scanner[6 ] class=IEEE802dot3      id= 6, loaded=true  direct=true , scan=false bindings=0  []
scanner[7 ] class=IEEE802dot2      id= 7, loaded=true  direct=true , scan=false bindings=0  []
scanner[8 ] class=IEEESnap         id= 8, loaded=true  direct=true , scan=false bindings=1  [MyHe
scanner[9 ] class=IEEE802dot1q     id= 9, loaded=true  direct=true , scan=false bindings=0  []
scanner[10] class=L2TP             id=10, loaded=true  direct=true , scan=false bindings=0  []
scanner[11] class=PPP              id=11, loaded=true  direct=true , scan=false bindings=0  []
scanner[12] class=Icmp             id=12, loaded=true  direct=true , scan=false bindings=0  []
scanner[13] class=Http             id=13, loaded=true  direct=true , scan=false bindings=4  [WebI
scanner[14] class=Html             id=14, loaded=true  direct=true , scan=false bindings=0  []
scanner[15] class=WebImage         id=15, loaded=true  direct=false, scan=false bindings=0  []
scanner[16] class=Arp              id=16, loaded=true  direct=true , scan=false bindings=0  []
scanner[17] class=Sip              id=17, loaded=true  direct=true , scan=false bindings=0  []
scanner[18] class=Sdp              id=18, loaded=true  direct=false, scan=false bindings=0  []
scanner[19] class=Rtp              id=19, loaded=true  direct=true , scan=false bindings=0  []
scanner[20] class=SLL              id=20, loaded=true  direct=true , scan=false bindings=0  []
scanner[21] class=MyHeader         id=21, loaded=true  direct=false, scan=false bindings=1  [Ip4]
libpcap::EN10MB(1)                 => header::Ethernet.class(1)
libpcap::IEEE802(6)                => header::IEEE802dot3.class(6)
libpcap::PPP(9)                    => header::PPP.class(11)
libpcap::LINUX_SLL(113)            => header::SLL.class(20)
Resolver IEEE_OUI_PREFIX: cache[count=11359], timeout[count=11361, positive=157680000000, negati
Resolver IP: cache[count=4], timeout[count=4, positive=1400000000, negative=9000000],
```

The scanner table holds all of our registered protocols. The last column 'bindings' shows the number of java defined bindings (any of the above bindings use cases) and a list of the names of the protocols those bindings are for.

If you created and registered a binding, you should see it in this table.

5.5.1 - Binding to Sctp header

The *Sctp* header was introduced in version 1.4 (1.4.r1370 to be specific). The protocol header uses a slightly different method for breaking down Sctp data. Sctp protocol is made up of a fixed-length header followed by 1 or more "Chunks". Most of the chunks are control information carriers about the Sctp connection itself with the esception of "data-chunk".

The jNetPcap breaks down the Scp packets in the following ways. The fixed-length Sctp header followed by one or more Sctp-Chunk headers. The important thing to note is that these are not "sub-headers" but top-level headers.

Here is an example of a Sctp packet. The output is acquired using the following command on a Sctp packet: `System.out.println(packet.getState().toDebugString())` which will printout the state information about the packet and all its headers:

```
JPacket.State#029   : [         Protocol(ID/Flag) | Start | Prefix | Header | Gap | Payload | Po
JPacket.State#029[0]: [         ETHERNET( 1/0800) |     0 |      0 |     14 |   0 |     356 |
JPacket.State#029[1]: [              IP4( 2/0800) |    14 |      0 |     20 |   0 |     336 |
JPacket.State#029[2]: [             SCTP(32/0800) |    34 |      0 |     12 |   0 |     324 |
JPacket.State#029[3]: [             DATA(33/0800) |    46 |      0 |     16 |   0 |     308 |
JPacket.State#029[4]: [             HTTP(13/0800) |    62 |      0 |    305 |   0 |       3 |
JPacket.State#029[5]: [        WEB_IMAGE(15/0800) |   367 |      0 |      3 |   0 |       0 |
```

Please notice that SCTP and DATA (Sctp chunk DATA) are separate top-level headers. Http header is bound to the "DATA" Header but in a special way. It had to go back 1 more header in the header-state stack to check the port numbers and verify that Sctp port number 80 was used. If that condition is met then the Binding object or method can return true and it will be placed after the "DATA" header.

Here is an example of a functioning binding in a custom protocol:

```
private final static JThreadLocal<Sctp> local = new JThreadLocal<Sctp>(Sctp.class);

@Bind(to=SctpData.class)
public static boolean bindToSctpData(JPacket packet, SctpData data) {
        final Sctp sctp = local.get();
        return packet.hasHeader(sctp) && (sctp.destination() == 3868 || sctp.source() == 3868);
}
```

The above code is used inside another header definition (Diameter class) that extends `JHeader`. Please note the following important key-points about the binding.

First, the binding is bound to `SctpData` header, not Sctp. However, inside the binding method, a lookup is made into `Sctp` header to check the port numbers. jNetPcap provides a utility `JThreadLocal` class that makes it easy to create thread-local safe headers. A thread-local instance of the `Sctp` is acquired, a check into the packet is made if the header exists and then the port numbers are checked. The presence of `SctpData` is a given, since our binding method is bound to it.

5.6 - Registering a header

Once a header definition is complete you need to "register" your new header with jNetPcap's registry of protocol headers, to make the header active. The `JRegistry` class provides a `JRegistry.register` method for registering new headers. In this step, the `JRegistry.register` method will scan the header definition for any errors to make sure all the required components of a header definition are there.

Here is how you would register a new header definition defined in `MyHeader` class file.

```
try {
  int headerID = JRegistry.register(MyHeader.class);

  System.out.printf("Header registered successfully, its numeric ID is %d\n", headerID);

} catch (JRegistryHeaderErrors e) {
  e.printStackTrace();
  System.exit(1);
}
```

When to register

It is important to know when to register the header with `JRegistry`. The scanner that is assigned to process packets coming from libpcap, takes a snapshot of all the headers registered at the time, as it is assigned to its task. Therefore you have to register your new header before that happens.

It is recommended that you register your header before even pcap capture is opened. Although this is strictly not necessary, it is recommended that the registration takes place before hand.

Registering from static initializer in header definition itself

A common approach is to put registration code inside the header definition itself in a static initializer. This ensures that the header is always registered the first time the class file is loaded into java VM.

```
... // header definition class
static {
  try {
    JRegistry.register(MyHeader.class);
  } catch (RegistryHeaderErrors e) {
    e.printStackTrace();
  }
}
... // rest of header definition class
```

There is however a subtle gotcha with this type of registration which is not very obvious unless you think about it. Static initializer is executed only once when the class file is loaded. Therefore you must ensure that the class file is loaded before any pcap capture sessions are opened. Here is an example of correct way to utilize our new header with static registration:

```
MyHeader my = new MyHeader(); // Loads the class file and executes static initializer
StringBuilder errbuf = new StringBuilder(); // Error buffer

Pcap pcap = Pcap.openOffline("test.pcap", errbuf);

// Capture packets with handler or next, nextEx, etc..

Ip4 ip = new Ip4();
if (packet.hasHeader(ip) && packet.hasHeader(my)) {
  // We are ready to process IP and MyHeader
}
```

An invalid approach:

```
StringBuilder errbuf = new StringBuilder(); // Error buffer

Pcap pcap = Pcap.openOffline("test.pcap", errbuf);

// Capture packets with handler or next, nextEx, etc..

Ip4 ip = new Ip4();
MyHeader my = new MyHeader(); // Header is registered too late!!!

if (packet.hasHeader(ip) && packet.hasHeader(my)) {
  // We are ready to process IP and MyHeader
}
```

As you can see that once the capturing of packets has begun only then do we reference our new header definition which registers the header too late in the process. At this time if a packet scanner has already been assigned to the capture, the packets will not contain the new header. This is a common error with new header definitions, as its easy to forget where the header is being registered and also natural to put the new header definition instantiation where the rest of the headers are instantiated. That can still be done, as long as you ensure that the class file is loaded before hand.

Source URL: http://jnetpcap.com/userguide