2006-12

Design and Implementation of a Modular Wireless Sensor Network Sniffer

McEachen, John C.; Siang, Teo Hong; Kirykos, Georgios

Telecommunications and Signal Processing, December 11-13, Hobart, Australia.
http://hdl.handle.net/10945/39263

12-13-2006

Design and Implementation of a Modular Wireless Sensor Network Sniffer

John C. McEachen
Naval Postgraduate School, Monterey, California, USA

Teo Hong Siang
Naval Postgraduate School, Monterey, California, USA

Georgios Kirykos
Naval Postgraduate School, Monterey, California, USA

Published in 5th Workshop on the Internet, Telecommunications and Signal Processing, December 11-13, Hobart, Australia.
Edited by Beata J Wysocki & Tadeusz A Wysocki. ISBN: 0 9756934 2 5

Design and Implementation of a Modular Wireless Sensor Network Sniffer

John C. McEachen, Teo Hong Siang, and Georgios Kirykos
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California, USA
mceachen@nps.edu

ABSTRACT

We present the design and implementation of a wireless packet sniffing application focused on the communications of TinyOS-based sensor networks. This modular, multi-threaded application allows robust analysis of transmitted frames in an easily understood format similar to *tcpdump*. The underlying software framework is presented and details of the implementation are discussed. Finally, verification of the application is performed using a basic TinyOS application.

Index Terms—Sensor networks, IEEE 802.15.4, Zigbee, wireless networking.

I. INTRODUCTION

A sniffer is the quintessential tool for network analysis. Such a tool is presently lacking in today's sensor network development kits. In the paper we will present the design and implementation of a TinyOS sensor network sniffer. The current design is based on existing sensor network hardware and software components to enable a low-cost sniffer implementation.

To understand how the sniffer works, it is necessary to first understand how TinyOS handles incoming network messages, and hence the hardware and software components necessary to implement such a sniffer.

Dest	AM	Grp	Len	Data	CRC
(2)	(1)	(1)	(1)	(0..29)	(2)

Figure 1. TinyOS packet format. The byte sizes of the fields are indicated in parentheses. [1].

A. Tinyos Filtering mechanism

A TinyOS packet has the format shown in figure 1. The fields of interest are the following:

1) Destination Address

The destination address refers to the 16-bit node address of the mote. It is programmed into the mote along with the TinyOS application when the following command is issued:

```
make <platform>  re|install,<n>
<programmer>,<port>
```

where <n> is the 16-bit node address. The address `0` is typically reserved for the base station mote.

2) Group ID

The group ID is analogous to the network address for a group of cooperating motes. It allows multiple distinct groups of motes to share the same radio channel. The group ID can be set by defining the preprocessor symbol `DEFAULT_LOCAL_GROUP`. For example, this symbol can be located in the `MakeXbowlocal` file in the `tinyos-1.x/contrib/xbow/apps` directory. This file is automatically included in the compilation of almost all TinyOS programs in the `tinyos-1.x/contrib/xbow/apps` directory. The default group ID is `0x7D`.

3) AM Type

TinyOS implements the Active Message (AM) system. AM types are analogous to port numbers in TCP/IP. Different applications may use different AM types. For example, for the *Surge_Reliable* application, the AM type is defined in the `Surge.h` header file:

```
enum {
  AM_SURGEMSG = 17
};
```

TinyOS automatically filters incoming packets by matching the destination address in the packet header with the node address of the mote. If the node address is `0`, then TinyOS skips this step – the mote is effectively operating in promiscuous mode. If the destination address matches the node address, then the entire packet, including the header, is passed on to the application. It is the application's responsibility to handle the group ID and AM fields.

In summary, a TinyOS sniffer can simply be a mote programmed with a node address of `0`, and an application that ignores the group ID and AM fields (assuming no *a-priori* knowledge of either of them).

B. Components of a Rudimentary Sniffer

A simple TinyOS sniffer that generates raw hexadecimal output can be quickly created based on readily available hardware and software components. The software consists of two programs: *TOSBase* and *SerialForwarder*. [2]

TOSBase is one of the many example TinyOS applications that can be found in the `tinyos-1.x/apps` directory. It acts as a simple bidirectional bridge between the serial and radio links. By programming the TelosB mote with *TOSBase* and a node address of `0`, we have a promiscuous receiver receiving TinyOS packets from the

MicaZ sensor network, and forwarding them to the PC via the serial link. So long as no data is sent to the serial link, *TOSBase* will also be a unidirectional passive receiver.

As mentioned earlier, it is the responsibility of the TinyOS application to handle the filtering of group ID and AM type. *TOSBase* performs this function in the following code segment from the `TOSBaseM.nc` file:

```
event TOS_MsgPtr
RadioReceive.receive(TOS_MsgPtr Msg) {
  …
  if ((!Msg->crc) || (Msg->group !=
TOS_AM_GROUP))
    return Msg;
  …
}
```

Clearly, *TOSBase* rejects corrupt packets, or packets with a different group ID. Assuming we have no *a-priori* knowledge of the target group ID, we can prevent the filtering of packets by group ID by simply altering the code as follows:

```
event TOS_MsgPtr
RadioReceive.receive(TOS_MsgPtr Msg) {
  …
  if (!Msg->crc)
    return Msg;
  …
}
```

TOSBase ignores the AM field, so AM filtering is not a problem.

At this point, we have a rudimentary TinyOS sniffer mote. Next, we need an application to read the raw data from the serial port, and interpret it into a more human-readable format. For this, we use *SerialForwarder*.

SerialForwarder is a Java applications that can be found in the `tinyos-1.x/tools/java/net/tinyos` directory. *SerialForwarder* runs on the PC and instantiates a network server that forwards TinyOS packets read from the serial port to a network port, and vice versa. Typically, it is used to allow applications to communicate with the mote via a network interface instead of a serial interface – especially useful for distributed network clients, or Java clients like *Surge*.

In this case, we use it to interpret the raw data from the serial port by using the following command:

```
java
net.tinyos.sf.SerialForwarder -
comm serial@<port> 2>&1 | grep
"Received message"
```

where `<port>` is the serial port used by the TelosB mote.

Unfortunately, this basic sniffer application is not well-suited for in-depth analysis of large numbers of TinyOS frames. The remainder of this paper will discuss the design and implementation of a more robust, user-friendly sniffer in the vein of *tcpdump* [3] to allow more advanced analysis of TinyOS traffic.

C. *Components of the sniffer*

Our TinyOS sniffer is based on readily available hardware and software components, and a Java-based application analogous to *tcpdump* [3].

1) *Software components*

The software consists of one TinyOS application, *TransparentBase*, and two custom Java applications, *SerialForwarder* and *Sniffer*.

TransparentBase is one of the many examples of TinyOS applications that can be found in the `tinyos-1.x/apps` directory. It acts as a simple bidirectional bridge between the serial and radio links. *TransparentBase* has the additional property that it ignores the Group ID. Programming the TelosB mote with *TransparentBase* and a node address of `0` gives a promiscuous receiver receiving TinyOS packets from the MicaZ sensor network, and forwarding them to the PC via the serial link. So long as no data is sent to the serial link, *TransparentBase* will also be a unidirectional passive receiver.

As mentioned earlier, it is the responsibility of the TinyOS application to handle the filtering of group ID and AM type. *TransparentBase* ignores both the group ID and AM type fields, so Group ID and AM filtering is not a problem.

At this point, this is a functional TinyOS sniffer mote. Next, an application is needed to communicate with the mote via the serial port. In TinyOS world, the standard way to this is to use *SerialForwarder*.

SerialForwarder is a Java application that can be found in the `tinyos /tools/java/net/tinyos` directory. *SerialForwarder* runs on the PC and instantiates a network server that forwards TinyOS packets read from the serial port to a network port, and vice versa. Typically, it is used to allow applications to communicate with the mote via a network interface instead of a serial interface – this is especially useful for distributed network clients, or Java clients like *Surge*.

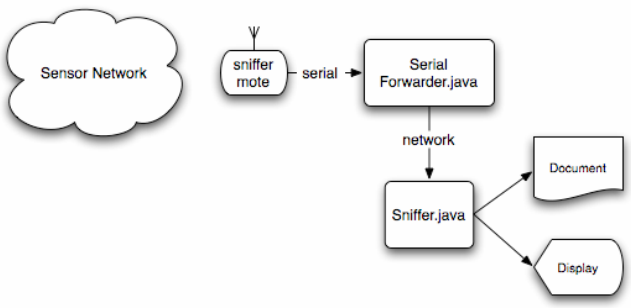

Figure 2. Block diagram of TinyOS Sniffer.

The final piece of the puzzle is an application that reads the packet from *Serial Forwarder*, extracts the protocol and application information, and outputs the information in a usable form. There is no such tool in current TinyOS development kits. For this reason, the Java application *Sniffer* has been developed. Figure 2 illustrates the block diagram of the sniffer.

At this point, the necessary hardware and software components of a TinyOS sniffer have been identified. However, an application, Sniffer.java, is still needed that reads the packet from *Serial Forwarder*, extracts the protocol and application information, and outputs the information in a usable form. This will be discussed in the next section.

II. DESIGN AND IMPLEMENTATION OF THE SNIFFER APPLICATION

In this section we discuss the design of a multi-threaded, *tcpdump*-like application that reads raw packets from *Serial Forwarder*, extracts the protocol and application layer information, and outputs it in a usable form.

The developmental paradigm of TinyOS is that mote applications running on the motes are written in nesC, while user applications running on the PC are written in Java. Keeping in line with this paradigm, *Sniffer* is also written in Java.

The remainder of this chapter explains the *Sniffer* application in more details. To follow the discussion, some working knowledge of Java is assumed, e.g. how to run Java programs, what are Java classes etc.

A. *Process flow*

The process flow of *Sniffer* is depicted in figure 3. The program first parses the command line to determine the user specified options, and exits upon detecting invalid options, or if "-?" is specified to display the help text. Once the options are set, the program attempts to instantiate a PacketServer class and initialize the Writer class with the registered outputs. If either of these fails, for example due to a failure to communicate with a *SerialForwarder* source, or a file I/O error, the program exits.

At this point, the program is ready to enter into its main processing loop. First, it requests a packet from PacketServer. Then it passes the packet to the Protocol and AM classes in sequence to process the protocol and application layer information respectively. Finally, the processed packet is passed to the Writer class for output to the screen, and is optionally written to file in a variety of formats if the user so specifies. The process loop continues indefinitely until the user terminates with *Ctrl-C*.

B. *Description of the major Java Classes*

The Java classes that are implemented for *Sniffer*, and their relationships to each other, are depicted by the unified modeling language (UML) diagram in figure 4. Note two exceptions. Firstly, the Thread super-class is a system provided class. It is included in the illustration to highlight the fact that *Sniffer* is a multi-threaded application. Secondly, the *CLI* utility library from the Apache Jakarta Commons project [4], not shown in the illustration, is used to provide the API for working with command line arguments and options. The user has to ensure that the *CLI* library is located in the Java *CLASSPATH*.

The Sniffer class contains the *main* function of *Sniffer*. This is the entry point of the program, and implements the process flow illustrated in figure 3. In addition, it keeps track of packets received as well as packets with protocol processing errors. This information is displayed on the screen when the user terminates the program with *Ctrl-C*.

It is useful to look at the UML diagram in the following way. The leftmost branch consisting of the PacketServer class and its children comprise the *input* subsystem. The middle branches consisting of the Protocol and AM classes comprise the *processing* subsystem, while the rightmost branch consisting of the Writer class and its children comprise the *output* subsystem. Underpinning these subsystems is a collection of data classes called Packet and Field.

The data classes, and the input, processing and output subsystems are now discussed in more details.

1) *Data Classes*

In order to capture the representation of a packet accurately as it is processed through the program, *Sniffer* defines a Field class and a Packet class. A Field is simply some value with an associated name. A Packet is represented as a triplet of header, payload data unit (PDU), and trailer. It may also have a description string. The description is useful for displaying human-readable packet information on the screen.

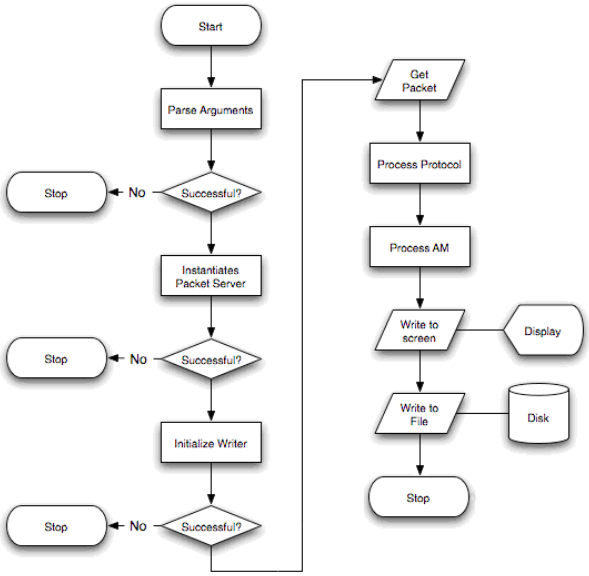

Figure 3. Process flow chart for the Sniffer application

The header and trailer are defined as linked lists of Fields. Perhaps more intriguingly, the PDU is itself defined as a Packet. The Packet class is therefore a *nested* class.

This allows packets to be encapsulated inside another packet easily.

If the PDU is null, then this Packet is the *uppermost* packet, and all the information about the packet is stored in the header. In OSI parlance, an upper layer payload is encapsulated inside a lower layer protocol. So *uppermost*, in this case, means the higher layer payload. The current processing system should only concern itself with the uppermost packet and payload.

In this case, there are two possibilities. On one hand, the header list may consist of just a single field containing the data payload waiting to be processed. On the other hand, the header can be null, indicating that there is no more data to be process, i.e. this packet has been fully resolved.

Hence, packet protocol processing works as follows:

1. Use the getUppermostPacket method to get the uppermost packet.
2. Use the getUppermostPayload method to get the uppermost payload to be processed.
3. Process the payload accordingly to derive new header, trailer and PDU information.
4. Replace the uppermost packet's header, PDU and trailer attributes with the new information. The new PDU will now become the new *uppermost* packet to the next protocol layer.

2) Input Subsystem

The heart of the input subsystem is the ByteServer class. A ByteServer object is instantiated by PacketServer, when it is in turn instantiated by the Sniffer class. ByteServer is a separate thread of execution from the main Sniffer class. This allows input from the sniffer mote to be received in the most responsive way, independent from the execution of the rest of the program.

The ByteServer receives data from the sniffer mote through objects subclassed from the Source class. Currently, only the SFSource (or Serial Forwarder Source) class is implemented. However, it is perfectly possible to implement other sources if necessary, for example a SerialSource class that communicates with the mote directly via serial port communications, or a file source to read data previously archived in a file. It shall be seen later that additional sources can be easily created and added to the system. By default, *Serial Forwarder* is assumed to be running on the local host, port 9001. If this is not the case, then the user can specify a host and port using the "-h" and "-p" options.

Data read from the source is stored in a First-In-First-Out (FIFO) buffer. In the current implementation, when the buffer is full, further packet data are dropped, instead of overriding the earliest received data.

The Sniffer class does not talk to the ByteServer directly, instead it instantiates and requests packets from the PacketServer class. PacketServer provides a packet-level input abstraction to Sniffer. PacketServer makes requests for received data from ByteServer, and then packages the data bytes into a Packet object as its payload. At the same time, PacketServer adds a timestamp to this Packet's header. This timestamp information allows for temporal analysis of the packets. Strictly speaking, the timestamp does not represent the time the packet is received by the sniffer mote, but the time PacketServer receives the data from ByteServer.

3) Processing Subsystem

Packet objects received from the input subsystem are passed to the processing subsystem, to extract protocol and application layer information. Packet objects are passed to the Protocol class and then the AM class in sequence.

The first thing that the Protocol class does is to see if it needs to determine the message format. TinyOS messages can have 4 different formats, depending on the mote used [2]:

1. The original TOS_Msg format used on mica,mica2, and mica2dot,
2. the IEEE802.15.4 format used on telos family motes,
3. the modified TOS_Msg format used on micaz motes, and
4. the Infineon's eyesIFX platform [5].

The Packet class has a static attribute, *msgFormat*, that is used to identify the format. The Protocol class first checks if this attribute is set. If it has not been set, the Protocol class applies some heuristics to try to guess the message format, and set it to the appropriate value. If the heuristics fail, then the message format is unknown, and no further processing will be done on the packet. It is assumed that a given sensor networks is homogenous, so the message format should be consistent. So the guessing of the message format is only done once. However, the heuristic rules are not foolproof by any means, so the fallback is for the user to specify the message format on the command line via the "-m" option. Once the message format is known, the Protocol class performs the appropriate protocol processing on the packet.

Once protocol processing is completed, the packet object is passed to the AM class for application processing. Recall that the AM type of a TinyOS message is analogous to the port number in TCP/IP. So it is conceivable that in a given sensor network, there can be more than one AM type. The AM class maintains a list of AM handlers. The AM class iterates through the list, passing the packet object to each AM handler. Once an AM handler signals that it has handled the AM type, application processing is complete.

Currently, the only AM handler implemented is the AMCountMsgHandler class, for the test application CountRadio. But it is easy to create and add AM handlers.

4) Output Subsystem

The Writer class is responsible for writing the packet information to a variety of output channels. One of the features of the output subsystem is that packet information can be written to more than one channel *simultaneously*.

Thus the user has the maximum flexibility in storing packet information.

The Writer class maintains a list of Output objects. The ScreenOutput object is always available. At the same time, the user can choose to enable other output channels using command line options. Currently, the following FileOutput objects are available: logging the screen output to file ("-l"), writing to a binary file ("-ob"), and writing to a comma separated value (CSV) file ("-oc"). The CSV file, for example, is readily imported into Microsoft Excel for data analysis purposes. The binary file, on the other hand, provides a more efficient format to store packet information.

When an Output object is instantiated, its *run* method is also immediately registered with the JVM runtime as a *shutdown hook*. This is a little known feature of Java that was recently added to allow programs to perform last minute clean-ups before JVM termination. In this case, the shutdown hooks enable the ScreenOutput object to print out packet accounting information, and the FileOutput objects to flush their I/O buffers and close the file properly.

Again, if a particular output channel is not available, it can be easily created and added to the system.

C. Extending sniffer

The *Sniffer* program has been designed so that it is easy to create and add functionalities to the various subsystem, without affecting how the rest of the system works. Part of the reason for this capability lies in the object oriented nature of Java. In general, adding functionality to *Sniffer* can be done in two steps: subclassing an existing class, then making the new class known to the *Sniffer* at the appropriate code locations.

1) Adding input sources

New input sources should subclass the Source class, or any of its subclasses, and override the *open*, *close*, and *read* methods where necessary.

Within the main Sniffer class, a new command line option may be specified so that the user can choose this input source instead of the default SFSource. Then inside the code that checks for this option in *parseArgs*, the new Source object must be instantiated, and assigned it to the *source* static variable. ByteServer will automatically interface with the new input source.

2) Adding AM handlers

Handlers for new AM types should subclass the AMHandler class, and override the *processAM* method.

The new AM handler class must then register with the AM class within the following code static segment of *AM.java*:

```
/*
 * Static code block.
 * This is where new AM handlers are
registered.
 */
static {
      appsList.add(new
      AMCountMsgHandler ());
      // Add new AM handlers here, like
      above.
}
```

3) Adding output channels

New output channels should subclass the Output class, and override the *open*, *close*, *write*, and *run* methods where necessary.

Similar to the input subsystem, within the main Sniffer class, a new command line option may be specified so that the user can choose to enable this new output channel. Then inside the code that checks for this option in *parseArgs*, the new Output object must be instantiated, and added to Writer's list of Output objects using the *addOutput* method. Packet information will then be automatically written to the output channel.

D. Test and Evaluation

The sniffing functionality of the TinyOS sniffer was tested and evaluated using a simple test application. For the purpose of this study, the target sensor network is a pair of MicaZ motes running the *CountRadio* application. The sniffer consists of a TelosB mote connected to a PC.

1) CountRadio application

CountRadio is an example application that can be found under the `tinyos/apps/` directory. From its `README.CountRadio` file;"CountRadio is a simple led/radio count program. The default application built from this directory is CountDual. CountDual either sends a count over the radio if the node address is equal to 1, or displays a count received over the radio otherwise."

The transmitting mote broadcasts a packet every 200 milliseconds, or five packets per second. The payload of the packet has the format defined in the `CountMsg.h` file:

```
enum
{
  AM_COUNT_MSG = 4,
};

typedef struct
{
  uint16_t n;
  uint16_t src;
} CountMsg_t;
```

Hence the payload is of constant four byte length, where the first two bytes represent a monotonically increasing sequence number, followed by a constant source address which should be 1. Therefore, this provides a very predictable data source to verify the correctness of our *Sniffer* program.

2) Output

With the motes up and running, *Sniffer* is started from the command line with no additional options. Figure 5 shows the captured screen output of *Sniffer*. Each line represents

one packet. The format of the output is designed to mimic *tcpdump* to a certain extent. It has the following form:

```
<Timestamp> <Protocol>
<Grp>.<Dest>.<AM> (<Len>) [<Payload>]
<CRC>
```

The timestamp is in the 24-hour notation, with up to millisecond precision. The protocol is denoted as "TOS", meaning TinyOS with TinySec disabled. The group ID, destination address and AM types are grouped into a "dotted" decimal notation that resembles an IP address with its port number.

Then using the CSV file output of *Sniffer*, and importing into Excel, it can be further verified that the sequence number N runs in consecutive order, so no packets were missed.

III. Conclusion

The design and implementation of the *Sniffer* program has been described in detail, including its usage, process flow, and the functions of the major Java classes. It has also been explained how new input sources, application handlers, and output channels can be easily created and added to the program in a very modular way.

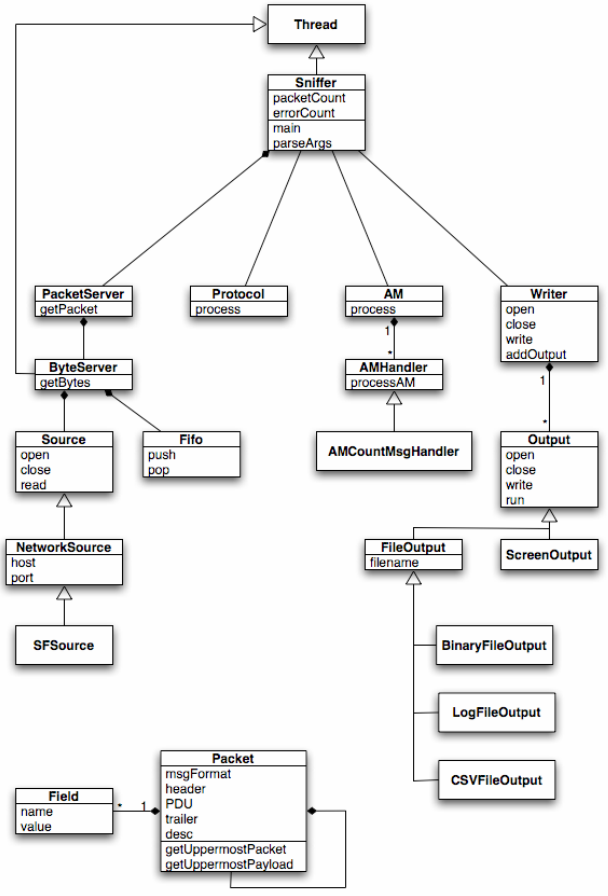

Figure 4. UML diagram of Sniffer.

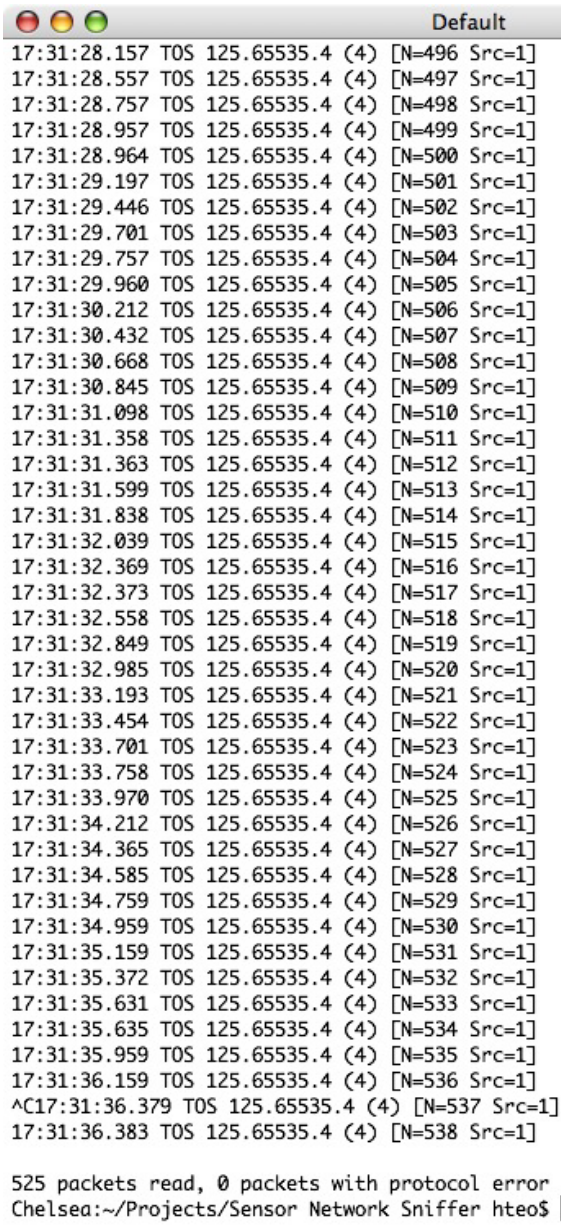

Figure 5. Captured screen output of Sniffer sniffing on 2 motes running the CountRadio application. The sniffer was started after the motes. Hence the lower packet count.

References

[1] Chris Karlof, Naveen Sastry, and David Wagner. "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks." In Second ACM Conference on Embedded Networked Sensor Systems, pp. 162-175. Baltimore, Maryland, USA. Nov. 2004.

[2] Gilman Tolle. "Serial Forwarder Protocol" [Online]. Available: http://cents.cs.berkeley.edu/tinywiki/index.php/Serial_Forwarder_Protocol

[3] *tcpdump/libpcap*. http://www.tcpdump.org

[4] *The Apache Jakarta Project, Commons CLI*. http://jakarta.apache.org/commons/cli

[5] "eyesIFX Sensor Network Development Kit- Documentation (Version 1.0.2)" Infineon Technologies AG, Germany. [Online]. Available: http://www.infineon.com/upload/Document/Eyes2.1_Doc_102.pdf