

Karlsruhe Sensor Networking

KSN RadioStack Manual

MARKUS BESTEHORN*

STEPHAN KESSLER†

ANDREAS LEPPERT‡

Department of Computer Science
Institute of Program Structures and Data Organization
University of Karlsruhe
76131 Karlsruhe, Germany

Version of March 18, 2009 12:56

* bestehorn@ipd.uni-karlsruhe.de
† stephan.kessler@stud.uni-karlsruhe.de
‡ andreas.leppert@stud.uni-karlsruhe.de

Contents

1	Introduction and Overview	2
1.1	Overview	2
1.2	Main goals and benefits	3
1.3	The layers in a nutshell	4
1.4	Source code and packages	4
2	General Concepts	6
2.1	Packet Quality Information	6
2.2	Layer binding	6
3	SHP Layer	8
3.1	Layer definition	8
3.2	Interfaces	8
3.3	Provided Implementation	9
3.3.1	Introduction	9
3.3.2	How does the protocol work?	9
4	SHP Level Dispatcher	13
4.1	Layer definition	13
4.2	Interfaces	13
4.3	Implementation	14
5	Abstract Layer	15
5.1	How to implement?	15
5.2	Send/Put methods	15
6	Routing Layer	17
6.1	Layer definition	17
6.2	Interfaces	17
6.3	Implementation	17
6.3.1	AODV Routing	17
6.3.2	Packets	18
6.3.3	Routing Table	19
6.3.4	Depth first search data transport mechanism	19
6.3.5	Some more details	20

7	Compress Layer	22
7.1	Layer Definition	22
7.2	Interfaces	22
7.3	Implementation	22
7.3.1	NoCompressionLayer	22
7.3.2	ZipCompressionLayer	22
8	Protocol Dispatcher	24
8.1	Layer definition	24
8.2	Interfaces	24
8.3	Implementation	25
9	Protocols	27
9.1	Layer definition	27
9.2	Interfaces	27
9.3	Implementation	27
9.3.1	PacketPortProtocol	27
9.3.2	LowPanEmulationProtocol	28
10	Getting Started	30
10.1	Our first application on the KSN RadioStack	30
10.2	Using the LowPanEmulation protocol	36

CHAPTER 1

Introduction and Overview

The KSN RadioStack encapsulates typical communication requirements for wireless sensor networks. The implementation presented here is for the Sun SPOTs¹ (Small Programmable Objects), but it could be ported to other sensor nodes. Since it is made for Sun SPOTs, all the code is written in Java. The ideas and the software were developed in context of the Karlsruhe Sensor Networking research project at the Institute for Program Structures and Data Organization (IPD) at the University of Karlsruhe.

1.1 Overview

The KSN RadioStack is basically a layer architecture while each layer handles different aspects of the wireless communication in sensor networks. It is similar to other well known communication stack implementations like TCP/IP. Since the Sun SPOT Library provides the physical and the MAC layer compatible to the used IEEE 802.15.4² WPAN standard, the implementation starts on top of the MAC layer. Figure 1.1 shows the different layers and their place in the stack. In this chapter the main goals and benefits are discussed; afterwards general concepts are presented and an overview for each layer is given. The last section explains the organization of the source files in packages.

1 <http://www.sunspotworld.com>

2 <http://www.ieee802.org/15/>

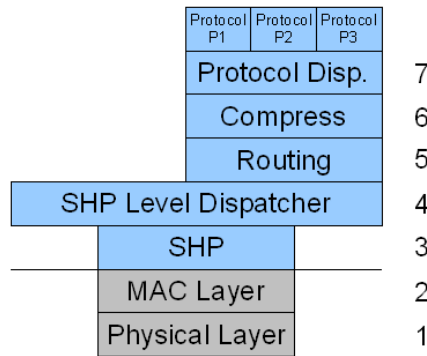


Figure 1.1: KSN RadioStack Layers, MAC and physical are provided by the Sun SPOT library

1.2 Main goals and benefits

clean layer architecture: From a software engineer's point of view there are many advantages to use the layer design pattern for a communication library. The KSN RadioStack has a really clean and well defined layer structure.

compression layer: Depending on the abstraction level of your software, it has to send a lot of data through the network (for example: in most cases a serialized object is bigger than its pure data representation). Probably the user does not want to think about the number of bytes his application uses to transfer data. But sending too much data will cause broadcasts to disappear and makes the wireless network really slow. A compression layer which is transparent to the application reduces the network traffic without a user even thinking about that. A basic ZIP Compression Layer is provided.

open architecture: In addition to the possibility to replace each layer with another implementation, more than one stack can be used. In the KSN RadioStack this is done by the SHP Level Dispatcher. This can be used to have different stacks for the real application and for the management tools. In large networks multiple stacks are really useful, for example you can play around with different routing protocols in your application and you will never lose the ability of over the air software deployment. Last but not least, most layers could be stacked however you want.

awareness of the link quality: For each layer and even for a protocol or the application there could be an interest to know how good or bad the wireless link to a destination is. The KSN RadioStack uses data provided by the lower layer, aggregates it, and makes it available to the protocols on top of the stack.

compatibility: We also provide a simple ILowPan implementation which keeps the stack compatible to the one delivered with the Sun SPOTs. See the protocols chapter [9.3.2](#) for more details.

protocols: You want to make your own high level protocol but you do not want to implement

an own routing mechanism or anything concerning details of WSN communication? This can be done really easily: there is a high level protocol dispatcher. Implementing a simple protocol on top of that will not take more than 100-150 code lines.

fully pluggable: You do not want a compress layer? Or do you want it on another position to compress the transferred routing data? All the layers between SHP Level Dispatcher and the Protocol Dispatcher could be exchanged as you wish to even dynamically at run-time.

1.3 The layers in a nutshell

Each layer has a specified job to do which is described in detail in its own chapter. This section provides a overview to get the "big picture". The layers are described bottom up. An illustration is provided in figure 1.1.

SHP Layer: SHP stands for Single Hop Protocol. It provides reliable communication between two sensor nodes for data larger than a single radio packet defined in the IEEE 802.15.4 standard.

SHP Level Dispatcher: Upper layers connect to that dispatcher in order to use the reliability of the SHP. This provides the ability to have more than one stack to handle node-to-node data traffic.

Routing Layer: This layer has to provide reliable host-to-host communication even if the destination and the originator are not in wireless range.

Compress Layer: This is where compression/decompression of the application data takes place.

Protocol Dispatcher: Application level protocols can be registered here. Protocols can send data using this layer in order to communicate with the same protocol on the destination node. This layer ensures that data of different protocols is not mixed up.

1.4 Source code and packages

The top level package for the KSN RadioStack is called "ksn.radiostack". There is an subpackage where the implementation of each stack element can be found. There are additional packages:

debug Debug package includes a simple profiler. This could be used to make performance measurements.

exceptions The exception package holds the Java exceptions which could be thrown over the layer boundary in order to inform about unusual behaviour. In this package there are only common exceptions which should all be declared in the layer interfaces.

interfaces There are interfaces and abstract classes describing each stack element and the quality information.

quality Implementations for the PacketQuality Information interface used throughout all the layers.

startup The LayerInit class gives access to the Standard Protocol Dispatcher and the both shipped protocols. They could and should be accessed that way in order to prevent to have multiple instances.

CHAPTER 2

General Concepts

There are a few things which nearly every layer has to deal with. This chapter provides details about that.

2.1 Packet Quality Information

As described in the introduction, it can be very useful to get information about the radio quality. Thus the MAC Layer provides basic quality information about each IEEE 802.15.4 radio packet. This information can be taken and aggregated. All classes containing quality information have to implement the interface `PacketQualityInformationInterface`. There are some get-methods for the quality data; but most important is the method for adding two `PacketQualityInformationInterfaces`. This is the way multiple quality data is aggregated.

The used implementation can be found in `AverageQualityPacketInformation`. The used average function weights every quality data equally.

2.2 Layer binding

To provide maximum flexibility to combine the layers, each layer does not need to know the exact definition of each linked layer. Instead the more generic interface `LayerInterface` has to be implemented by the linked layers. The `LayerInterface` itself extends the `DataReceiverInterface` and the `DataSenderInterface`. The `DataReceiverInterface` is used to pass data upwards and the `DataSenderInterface` provides methods to pass data downwards. So all incoming data for a specific layer is handled by the `DataReceiverInterface` methods and (nearly) all out-going data has to be passed on to the lower layer via the methods provided by the `DataSenderInterface`.

Between the SHP Level Dispatcher and each Protocol Dispatcher all layers can be mixed. The SHP Level dispatcher and the layers below cannot be replaced. Each Protocol Dispatcher has to define its stack elements and their order, for example the implemented `StandardProtocolDispatcher` defines the layer which is drawn on figure 8.1. As a result of that the first layer above the SHP Level Dispatcher cannot put its data downwards via the `DataSenderInterface`. Thus you have to decide if the layer is above another layer, or

the SHP Level Dispatcher. To encapsulate this problem, the `AbstractLayer` is introduced. You can find more information in [chapter 5](#).

CHAPTER 3

SHP Layer

3.1 Layer definition

Reliable node to node communication is necessary for any communication in a sensor network (Note: "node to node" means two sensor nodes which are in wireless range). The MAC Layer defined in the used IEEE 802.15.4 standard and implemented on the Sun SPOTs provides reliable communication between two nodes, so why is there another layer? Well, the Packets defined in the IEEE 802.15.4 standard leaves a payload size of up to 127 bytes. Most likely, there is more data to transport for a single request, thus the SHP Layer has to provide reliable communication for data much larger than 127 bytes. The best case would be, if there is no size boundary at all. For practical reasons, on the SunSPOTs the amount of data that may be transmitted is bounded by the limited memory, but obviously the design should avoid a tight limit on this low level of the communication stack. The design itself does not limit the size of any message. Depending on the hardware/software platform, the layer is implemented at, there may be some limits. On the Sun SPOTs for example there are limits because the size of sequence numbers and the size of a byte array is bounded by the 32bit VM which is used.

3.2 Interfaces

All implementations of a SHP Layer have to use the Singleton design pattern and have to inherit the `AbstractSHPLayer` class. Therefore the `SHPInterface` has to be implemented, this includes two methods:

- `sendData(IEEEAddress dest, byte[] data)`: Sends out the byte array as a unicast to the specified destination.
- `sendBroadcast(byte[] data)`: Broadcasts the data in the byte array.

Both methods return on success, otherwise an exception is thrown.

The SHP layer receives data from the MAC layer. For detailed information about the MAC Layer take a look in the Sun SPOT documentation.

All successfully received data is passed on to the local `DataReceiverInterface` which has to be provided to construct this singleton object.

3.3 Provided Implementation

3.3.1 Introduction

The provided implementation (`shp.SHP`) aims to reach maximum reliability and is possibly not the fastest. There are nearly no data size limits. An unicast can (theoretically) have unlimited size. They are only limited by the maximum size of a byte array in the Java implementation. So the unicast can be up to $2^{32} \text{ byte} = 4.294.967.296 \text{ byte} = 4096 \text{ MB}$ more than you would probably need in a wireless sensor network (WSN). The broadcast has a lower limit: "only" $6.881.280 \text{ bytes} = 6.5625 \text{ MB}$ can be sent. But this is no drawback since broadcasting data larger than 10 kb results in huge packet loss, and a SunSPOT "only" has around 4 MB of flash memory which is a much harder limit than 4 GB or 6.8 MB .

3.3.2 How does the protocol work?

First of all, the different unicast packet types and their layout are discussed. After that, the activity of a unicast data transfer is explained. For the broadcast we do the same afterwards.

Unicast

The packet layouts for the unicast are illustrated in figure 3.1 and the packets are described below. The packets can be distinguished by the packet type field.

Data Request packet: If this packet is sent from node A to node B, this means that A wants to send data. The total number of bytes and the number of fragments are indicating how much data is expected. All the following packets concerning this data transfer have the same request number as this packet. In order to check if the data is corrupted a checksum number is provided. Note: the size in bytes is not necessary, but as described above the total size is limited by the total size of a byte array, this is more or less redundant information. So why is it still there? Well, there is enough space in one radio packet provided by the MAC layer so there is no negative consequence.

OK, go ahead packet: Answer to a Data Request packet which means that the receiver is ready for incoming data with the provided request number.

Data packet: Packet holding a fragment of the data. It is ordered by the sequence number.

OK, Data Ack packet: A Data Ack packet means that a Data packet for the indicated request/sequence number was received successfully.

Complete packet: The data was received and defragmented successfully and the checksum was correct.

Checksum Error packet: Even though packets were received, the checksum computed from all these packets does not match the one provided in the Data Request packet before.

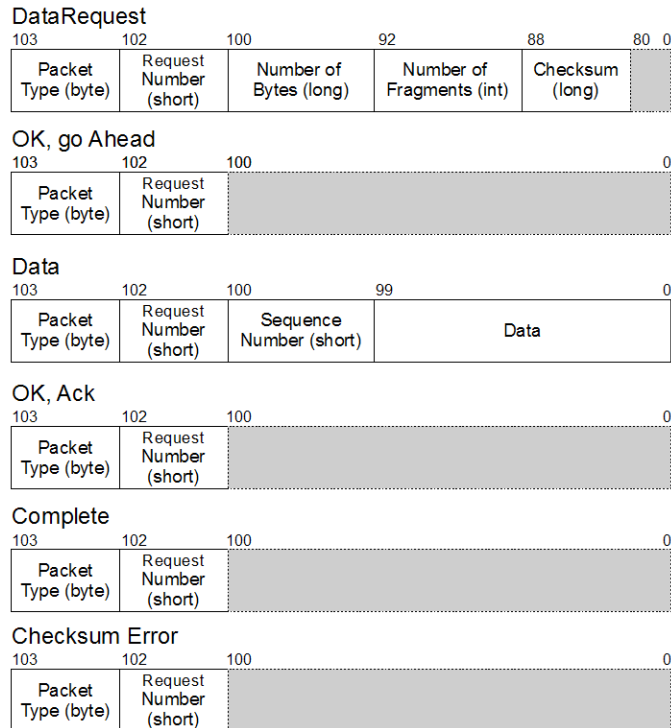


Figure 3.1: Illustration of SHP unicast packets

After all the various packet formats are discussed, it is time to talk about the packet flow and how they work together. A data transfer is illustrated in figure 3.2. The transfer is initiated by a Data Request packet. After receiving a "OK, go ahead" packet from the destination the originator starts sending fragments of the available data. The request number stays the same over the whole transfer operation. The sequence number starts with one, the receiver is then just waiting for the next sequence number which is calculated by adding one to the sequence number of the last received packet (the possible overflow of the sequence number is just ignored, because both sides have the overflow at the same time).

Broadcast

There are only two packages, they are illustrated at figure 3.3:

Broadcast with more Fragments Packet: This indicates that the Broadcast is not completed yet, there are more packets to arrive. Of course there is a request number which (together with the originator IEEE address) makes this broadcast unique. A broadcast starts with sequence number 1, for each following fragment the sequence number is incremented by one.

Broadcast last Fragment Packet: If this packet is received this means either that the whole broadcast has only one fragment, or that this is the last fragment of a broadcast request.

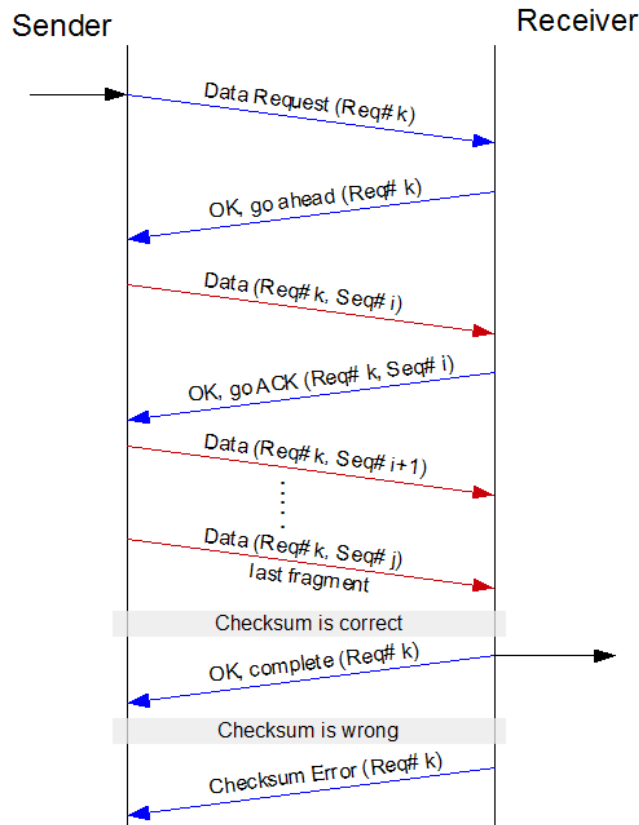


Figure 3.2: Packet flow during a unicast transfer

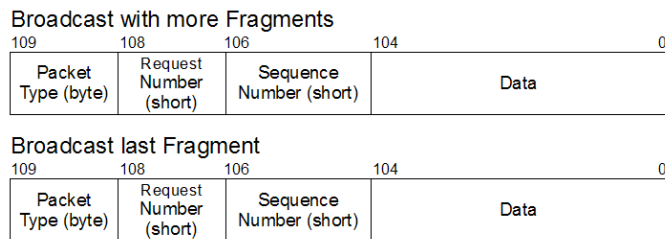


Figure 3.3: Illustration of SHP broadcast packets

The packet flow is illustrated in figure 3.4. A broadcast - as already discussed - can consist of one or more fragments. Each receiver has to receive all parts of a broadcast in order to have the complete data. The packets are consecutively numbered by the sequence number.

Constants

The constants for this layer, including packet numbers and parameters for error handling (see next paragraph), are defined in `shp.SHPConstants`.

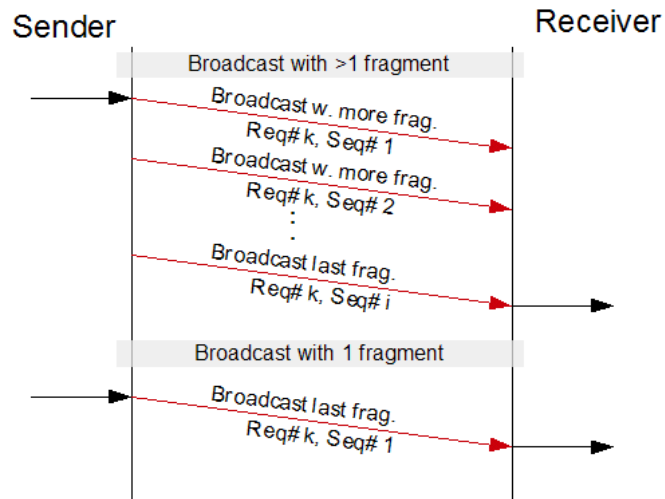


Figure 3.4: Packet flow during a broadcast transfer

Error handling

The `UnicastReceiverTimeout` and the `BroadcastReceiverTimeout` define the time a broadcast or a unicast receiver waits for the next correct packet. If a receiver receives a wrong packet (for example a packet with an unexpected sequence number) it ignores that `MaxWrongPacketCount` times before the transfer fails. Any packet sent is tried `SendTryCount` times. There is no send timeout because the underlying MAC layers answers any error with an appropriate exception.

Quality handling

As a result of the large amounts of packets send over the air during a fragmented data transfer, all the received quality information is aggregated in `AveragePacketQuality-Information` objects and passed on to the upper layer.

CHAPTER 4

SHP Level Dispatcher

4.1 Layer definition

This layer enables the stack to have more than one independent series of upper layers using the reliable node-to-node communication of the SHP layer. Why do you possibly need this? Well here is a real world example: we deployed a wireless sensor network using Sun SPOTs at our institute where we want to do experiments concerning query processing in WSN. This could also include changing the routing mechanism which is used for the query processing software. The sensor nodes are deployed in the different offices. All the software deployment has to be handled over the air, because it will take a long time to run through all the offices and put an USB cable in. This should be reliable, because the test application could possibly crash, but because of the SHP Level Dispatcher, the management communication stack is not affected. That is why this layer is really important: you can test different end-to-end stacks while using over the air management.

4.2 Interfaces

All implementations of a SHP Level Dispatcher have use the Singleton design pattern and have to inherit the `AbstractSHPLevelDispatcher` class. Therefore the `SHPLevelDispatcherInterface` has to be implemented which includes the following methods:

- `void registerLayer(byte ID, DataReceiverInterface dataReceiver)`: Using this method, an upper layer can be registered with a specified layer identification number. Using this method will cause that all sent data with the same layer ID will be passed to this data receiver.
- `void unregisterLayer(byte id)`: Stops passing on data for the specified ID.
- `void sendData(IEEEAddress destination, byte[] data, byte layerId)`: Sends out data as a unicast to the destination. A data receiver which has been registered with the layerId on the destination will receive this data.
- `void sendBroadcast(byte[] data, byte layerId)`: Data is sent via broadcast. Anyone who hears this broadcast completely and is registered on the layerId will receive the data.

The methods return on success, otherwise exceptions are thrown. The dispatcher has to implement the `DataReceiverInterface` to work with the SHP Layer.

4.3 Implementation

The implementation is as easy as you can imagine. The byte indicating the layer ID is simply put in front of each byte array from the upper layer. If data is received, the first byte is read and the rest of the data is passed to a waiting data receiver. If there is none, the data is discarded without any notice. The implementation can be found in `SHPLayerDispatcher`.

CHAPTER 5

Abstract Layer

All layers which want to be dynamically stackable have to extend the `LayerInterface`. Doing this has the serious drawback that the layer code has to decide whether it is on top of another layer or the SHP Level Dispatcher. This problem is discussed more precisely in section 2.2. The `AbstractLayer` extends the `LayerInterface` and implements so-called Put-methods which encapsulate the decision how to pass data to the lower layer. Combining the different layers to a stack is discussed in the Protocol Dispatcher chapter. How a layer can be implemented is described here.

5.1 How to implement?

On object construction the `AbstractLayer(LayerInterface lowerLayer)` or the `AbstractLayer(AbstractSHPLevelDispatcher shpLevelDispatcher, byte stackId)` constructor have to be called. Calling the first constructor means that the layer below is another `LayerInterface` implementation. Passing an `AbstractSHPLevelDispatcher` means that this layer is automatically registered at the given dispatcher with the given `stackId`. Only one layer for a specified stack (on top of the `AbstractSHPLevelDispatcher`) should be registered to the SHP Level Dispatcher since this is the lowest layer in this particular stack.

After defining the lower layer on construction, the upper layer has to be set via the `setDataReceiver(DataReceiverInterface dr)` method. All data for the upper layers have to be passed on to this interface. It could be accessed via the `getDataReceiver()` method.

All data for the lower layer have to be passed on using the `putData()` methods. There are three methods for different kinds of needs, they are corresponding to the `sendData()` methods which have to be implemented. This is discussed in more detail in the next section.

5.2 Send/Put methods

A layer inheriting from the `AbstractLayer` has to provide three send methods for the upper layers:

sendBroadcast(): The upper layer wants to send a broadcast. The passed data has to be broadcasted.

sendData(): This method is called, if there should be a single try to send data. If something goes wrong an Exception (which has to be derived from the `KSNRadioStackException`) is thrown, otherwise everything worked correctly.

sendDataWithTryCount(): This one tries the specified number of times to deliver the data to a specified destination. Because there could be multiple kinds of exceptions simply true or false is returned, in order to inform the caller about success or failure of the data transfer.

The `AbstractLayer` implements three "put" methods: `putBroadcast`, `putData` and `putDataWithTryCount`. This has to be used for all outgoing data transfer from the current layer. The lower layer should not be accessed directly because this could be another `LayerInterface` or the `AbstractSHPLevelDispatcher`. But this decision has to be made at runtime.

CHAPTER 6

Routing Layer

6.1 Layer definition

This layer enables host-to-host communication in the WSN. All layers on top of this layer assume that data can be sent to every reachable node in the network.

6.2 Interfaces

Because this layer is above the SHP Level Dispatcher, it extends the `AbstractLayer` (which implements the `LayerInterface`). It also implements the `RoutingInterface` but currently this is for marking purposes only. Maybe there will be additional features introduced by the `RoutingInterface` in later versions.

6.3 Implementation

The shipped routing layer implements with a few changes the AODV Routing protocol which is described in the RFC 3561¹. Note: this section does not describe the AODV protocol in detail, if you need further information how AODV works, please refer to the RFC.

6.3.1 AODV Routing

The AODV routing protocol is an experimental protocol designed for mobile ad-hoc networks. It ensures loop-freedom and avoids problems such as the popular "counting to infinity"-problem. For information on how this is done, please read the RFC. Here is an overview of how a route is found:

1. The originating node A wants to send data to the destination B and currently does not hold a valid route. Thus a route request (RREQ) is broadcasted.

¹ <http://www.ietf.org/rfc/rfc3561.txt>

2. If the RREQ is received by an intermediate node which does not have any routing information to B the intermediate node rebroadcasts the RREQ. If it has valid routing information an intermediate route reply (RREP) is sent back.
3. If B receives the RREQ it answers with a RREP which is forwarded using unicast communication to A. While receiving the RREQ all the backwards routes are established.

The mechanism is illustrated at figure 6.1.

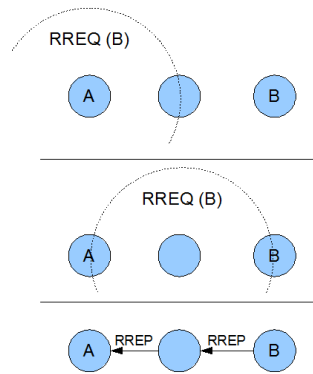


Figure 6.1: AODV route finding mechanism

The AODV specification also includes a route invalidation part, but this is not used in this implementation, we have done that on our own. The following sections discuss the implemented routing layer in more detail.

6.3.2 Packets

The following packages make the routing layer capable of doing host-to-host communication. The implementations can be found in `routing.packets.*`

Route Request Packet (RREQ): As described before, this is a broadcast packet which "searches" for a route.

Route Report Packet (RREP): Unicast packet which is replied if a valid route was found.

Data Transport Packet: If a valid route was found, this packet - containing a byte array for the payload - is sent through the network.

Data Ack Packet: If a packet was received successfully by the destination, it sends this packet back to the originator in order to indicate that everything went fine.

Data Route Error Packet: This one indicates that there was a broken route/hop on the way to the destination, so that the originating node of the Route Error Packet cannot deliver the Data Transport Packet. The receiving node of this packet should check if there is an alternate route, and if there is none, it has to send a Data Route Error Packet back to the last hop.

Details on how the Data Packets work together are described in section 6.3.4.

6.3.3 Routing Table

Each node in the network manages its own local routing table. Each entry holds the following information:

Destination: The IEEE address of the destination of this route.

Next Hop: The IEEE address of the node to which data has to be sent in order to reach the destination.

Destination Sequence number: number used by AODV for route management. Ensures loop freeness.

Packet quality information: Aggregated information about the quality of connections belonging to this route.

Hop Count: Number of hops which have to be taken in order to reach the destination. If a destination is directly reachable, it has hop count 0.

This is a 2-Dimensional table since each destination can have multiple entries which have a different next hop. An example is given on figure 6.2.

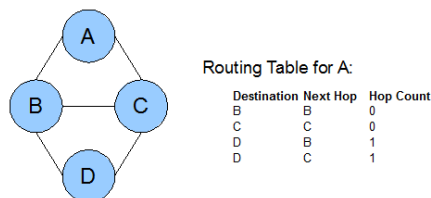


Figure 6.2: Example table, nodes that can hear each other are connected

6.3.4 Depth first search data transport mechanism

The mechanism how application data is transported to the network is illustrated in figure 6.3. The following explanation refers to this illustration:

1. Data should be transported from A to E; the routing engine searches for the shortest route. If no route is found in the current table, a RREQ is originated. But we assume that there are two routes present: A B C E and A B D E. Because they have both the same hop count, both routes could be chosen, and A B C E is chosen.
2. During the transport, the connection between C and E breaks down.
3. C invalidates its route to E and searches for another route. Because C has no other route in its own table, it sends a Data Route Error back to B.
4. B also invalidates the route to E with the next hop C (note: the route to C itself stays valid, this can not be clearly illustrated in the figure). It knows another route to E using D. So the data is now routed over D.

5. Now the data has arrived at E, and E originates a Data Ack Packet.

This mechanism works recursively. So if another route breaks down, the Data Route Error Packets travel back and another route is searched for. If, and only if, the originating node receives a Data Route Error Packet and has no other route, a RREQ Packet is broadcasted. Every other node will not start a route search on its own.

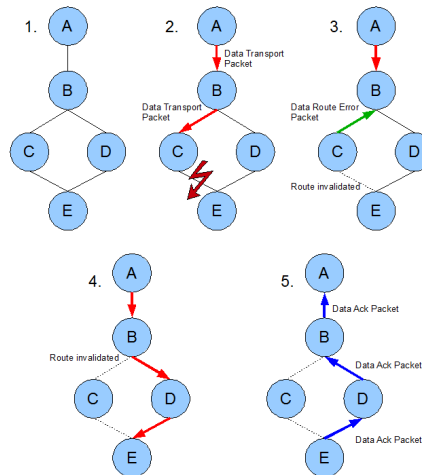


Figure 6.3: Example Data Transport between A and E, in step 2 the connection between C and E breaks

6.3.5 Some more details

Well, we have understood now how the host-to-host communication works. If you read the AODV RFC, or if you more familiar with routing in general, you probably have some more questions concerning the details of this process. This section is there to answer them.

Which route is chosen if there are more than one available? Does the quality information interfere?

Currently the route with the lowest hop count is chosen. If there are more than one, the first one in the routing table is taken. Even though is quality information available, it is just ignored. However it would be a good idea to add some minimum criteria a route has to fulfill in order to be chosen as a valid one. In fact it is not that easy because the significance of the quality information depends on how many packets have been sent over this route.

How do we get multiple routes? The AODV specification does not make this really clear...

It is a very easy mechanism and it will not get every available route. The policy is: if a RREQ is received by the searched destination (that means that no intermediate node has a valid route information) it is answered with a RREP everytime, even if it is the same RREQ. This RREP is send to the originating node of the broadcast. The RREP itself has to travel back to the originator of the RREQ. So every node has to search for a

backwards route itself. This is usually no problem, because backwards routes are created during receive of a RREQ. But again the policy is: choose the shortest route. At this point possible alternative routes could get lost because the originator receives the RREP only from nodes which are on a shortest available path. Avoiding this problem is not easy because simply sending the RREP back to all possible routes will lead to a loop.

There is a Route Error Packet (RERR) in the AODV specification, why is there none in this application?

The RERR packet in the AODV specification does more as we need here. It informs precursors and takes care of removing the route from other nodes. The functionality we need (delete the route locally on a breakdown and do that recursively backwards), is done by the depth first search data transport mechanism.

Why do we need a two-dimensional table if we use only one route?

Remember: A new RREQ is only originated, if the whole route goes (recursively) wrong. So if the originator of the data receives a Data Route Error, it will emit a RREQ. Any intermediate route will not because this could lead to route flapping. To explain this more precisely take a look at figure 6.4. Let us assume the nodes have only a one-dimensional routing table. That means that A can hold only one route to C. This would probably be the direct one with zero hops. But this one is a route with low connection quality. Packets with route information would probably take this hop because they are small (so they need just a few fragments on the SHP level) but the probability that a data packet (with many fragments) will fail is very likely. Well, A tries to transfer data to C and it fails. So what to do now? If we emit a RREQ B and C will hear that broadcast. C is the destination, so it will answer my RREQ while B rebroadcasts my request. While C receives the broadcast of B, A receives the unicast of C reporting a valid direct route to C. Now we have exactly the same situation as before.

In contrast to that, what will happen if there is a two-dimensional table. There are two routes to C, the direct one and the one over B. A failure on the direct route is no problem anymore because A can simply invalidate the direct route now and sends over B. If everything works, there will be no other attempt to send data directly to C (this would require a new RREQ).

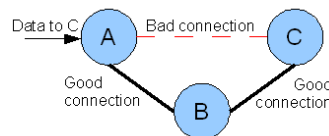


Figure 6.4: Example for route flapping

CHAPTER 7

Compress Layer

7.1 Layer Definition

This layer compresses data from the application layer on the way down to the routing layer. In the standard stack we have decided that the compression is done above the routing layer because the data should only be decompressed at the destination. Compressing data from the routing layer will cause every node which is used for the routing to decompress a large amount of data. If your aim is reduce all possible network traffic it is also suitable to put the compress layer as the last layer on top of the SHP Level Dispatcher. Note: without any further modifications, all nodes in the network should use the same compression layer, if not, the data may not be decoded.

7.2 Interfaces

Because this layer is above the SHP Level Dispatcher, it extends the `AbstractLayer` (which implements the `LayerInterface`). It also implements the `CompressInterface` but currently this is for marking purposes only. Maybe there will be additional features introduced by the `CompressInterface` in later versions.

7.3 Implementation

7.3.1 NoCompressionLayer

This one does no compression. We probably do not need it, but it could be used for debugging purposes, to check if the layers are correctly wired up or if there is a problem with the compression.

7.3.2 ZipCompressionLayer

As the name says, this implementation uses the famous ZIP algorithm to compress the outgoing data. There is no ZIP library available on the Sun SPOTs so we used

jzlib¹ and ported it to the Sun SPOTs. It required nearly no modification, only the `FilterInputStream` and the `FilterOutputStream` had to be copied from the original Java API and modified to work on the Sun SPOTs.

Experiments have shown that the compression is really efficient. We flashed a remote SPOT over the air and logged the compression rates. The big flash file was fragmented in pieces of 10 *kb*. The average compression rate was about 50-60%.

¹ <http://www.jcraft.com/jzlib/>

CHAPTER 8

Protocol Dispatcher

8.1 Layer definition

The basic idea of a protocol dispatcher is that different protocols can be registered and the dispatcher coordinates the traffic for the protocols. Data put into the dispatcher from a specified protocol are handed over to the specified protocol on the destination. In the KSN RadioStack the protocol dispatcher somehow defines the underlying stack. This should be the place where it is decided how the stack is built up.

8.2 Interfaces

There is no need to inherit a specified class or extend an interface, but to keep protocols from a higher level compatible to more than one stack you probably should follow this guideline. All implementations of a Protocol Dispatcher have to use the Singleton design pattern and have to inherit the `AbstractProtocolDispatcher` class. Therefore the `ProtocolDispatcherInterface` has to be implemented, this includes the following methods:

- `void registerProtocol(ProtocolInterface protocol)`: Registers an application level protocol on the dispatcher.
- `void deRegisterProtocol(ProtocolInterface protocol)`: Removes an application level protocol from the dispatcher.
- `void sendData(ProtocolInterface originatingProtocol, IEEEAddress destination, byte[] data)`: Sends out an unicast for/from the specified protocol. The corresponding protocol will receive the data at the destination.
- `boolean sendDataWithTryCount(ProtocolInterface originatingProtocol, IEEEAddress destination, byte[] data, int noOfTries)`: Does the same as `sendData` but does not throw an exception. This method returns true on success and it tries up to `noOfTries` times to send the data, if all else fails it returns false.

- `void sendBroadcast(ProtocolInterface originatingProtocol, byte[] data)`: Sends out a broadcast. Any node running the application level protocol will receive this broadcast.
- `ProtocolDispatcherDataPacket getDataFor(ProtocolInterface protocol)`: Get available data for the specified protocol. If no data is present and no data is received, this call will wait forever.
- `ProtocolDispatcherDataPacket getDataFor(ProtocolInterface protocol, long timeout)`: Does the same as the method above, but waits at most `timeout` milliseconds.

Data from the lower layer is passed on via the implemented `DataReceiverInterface`.

8.3 Implementation

The provided dispatcher builds up a stack illustrated in figure 8.1. It uses all the described implementations.

Std. Prot. Disp.	7
ZIP Compr.	6
AODV Routing	5
SHP Level Disp.	4
SHP	3
MAC Layer	2
Physical Layer	1

Figure 8.1: Standard stack

The implementation is really straight forward: a queue is created for every protocol that registers. In these queues there are data packets received by the lower layer. Since each protocol has its own protocol ID, the data can be distinguished by simply putting and reading the first byte to determine to which protocol this data belongs to. There is a semaphore preventing too many threads to enter the send methods. If too many threads enter, the data transfer would be really slow, so this is limited.

Listing 8.1 shows how the stack elements are put together. Here is another important thing: each Protocol Dispatcher has to have its own unique ID (so called stack ID) for which it registers at the SHP Level Dispatcher.

Listing 8.1: Layers are wired up

```
1 //instantiate the stack
2 this.routingLayer = new AODVRoutingLayer(LayerInit.
   getSHPLevelDispatcher(), StandardProtocolDispatcher.stackId);
3 this.compressionLayer = new ZipCompressionLayer(this.
   routingLayer);
4 this.routingLayer.setDataReceiver(this.compressionLayer);
5 this.compressionLayer.setDataReceiver(this);
6 this.lowerLayer = this.compressionLayer;
```

CHAPTER 9

Protocols

9.1 Layer definition

This is no real layer like the others. Here protocols for the application layer can be placed. Depending on the underlying stack, a protocol has to do different tasks. The common way is that the stack is capable for end-to-end communication, and the protocol defines the way the nodes communicate on a very high level (stream based, packet based, ...).

9.2 Interfaces

You do not have to but it is recommended that a protocol implements the `ProtocolDispatcherInterface`. This will make it compatible to all the protocol dispatcher implementations that inherit the `AbstractProtocolDispatcher`. Implementing this interface is as easy as it gets, there is one single method:

- `public Byte getProtocolId():` This method returns the protocol ID needed by the `ProcolDispatcher` to coordinate the data flows. This ID has to be unique for all used protocols.

9.3 Implementation

Currently there are two implementations shipped with the KSN `RadioStack`. The `PacketPortProtocol` and the `LowPanEmulationProtocol`.

9.3.1 PacketPortProtocol

This protocol transports data as packets and is able to provide several ports to listen on. Sending data is done by these methods:

- `void sendBroadcast(byte[] data, byte portNo)`
- `void sendData(IEEEAddress destination, byte[] data, byte portNo)`
- `boolean sendDataWithTryCount(IEEEAddress destination, byte[] data, byte portNo, int noOfTries)`

All methods need the data as a byte array and a port to which the data should be sent.

Before data could be received you have to open a port. This can be done with the `public void startListenOnPort(byte portNo)` method, while listening can be stopped with the `public void stopListenOnPort(byte portNo)` method.

Data can be received on a port we started listening on by one of the following methods:

- `public PacketPortProtocolPacket getData(byte portNo)`
- `public PacketPortProtocolPacket getData(byte portNo, long timeout)`

The first one can block forever, while the second one throws a `TimeoutException` if `timeout` milliseconds have exceeded.

The packet port protocol is nearly the most basic protocol which can be imagined. So it is very easy to use and the code is even easy to read.

The protocol can be instantiated by using the no-argument constructor or by passing an `AbstractProtocolDispatcher`. The no-argument constructor will use the Standard Protocol Dispatcher which can be accessed via the `LayerInit` class. You should not register the protocol twice to the same protocol dispatcher since then the protocol ID is already in use. The best way for most normal applications is to access the protocol instance from the `LayerInit.getStandardPacketPortProtocol()` method.

9.3.2 LowPanEmulationProtocol

Developing a completely new radio stack has a serious drawback: all existing applications, build up on top of radiostream and radiogram protocols are completely broken. Even though you have organized your source very well, you will have some difficult work to do to get your code running on the new stack.

Well therefore we have developed a small emulation protocol of the original LowPan layer. It makes the radiostream and the radiogram protocol capable of running on top of the KSN RadioStack. You can even use the over the air management commands of the Sun framework. Well there are some drawbacks, but let us talk about this later.

How to use

The usage is really simple, but you have to rebuild the multihop library. Only one change is necessary. You have to modify the `getInstance()` method of the `com.sun.spot.peripheral.radio.LowPan` class. The modification is shown in listing 9.1. To make this modification possible, you probably have to copy the KSN RadioStack package into the source of the multihoplib. This is also the right place for a stack. For more detailed information on how to rebuild the library, look in the Sun Developers Guide or in the "Getting Started" chapter in this manual.

Listing 9.1: LowPan modification

```

1 /**
2  * Get the instance of this singleton.
3  * @return the LowPan packet dispatcher for this SPOT
4  */

```

```

5  public static synchronized ILowPan getInstance() {
6  //      if (lowPan == null) {
7  //          lowPan = new LowPan(Spot.getInstance().
           getRadioPolicyManager().getIEEEAddress(), AODVManager.
           getInstance(), RadioPacketDispatcher.getInstance());
8  //      }
9  //      return lowPan;
10     return LayerInit.getLowPanEmulationProtocol();
11 }

```

Details on the implementation

The implementation simply passes the data of the `send()` methods from the `ILowPan` over the KSN RadioStack. It coordinates the incoming traffic by giving it to the right `IProtocolManager`. That is all that has been done. Any access to routing mechanisms like in the original LowPan implementation are not emulated. In fact this would be a breach to the clean layer architecture.

Known problems

Well there are some, but most could be (or are going to be) solved in future releases:

bad performance with radiostream connections: The radiostream protocol has a parameter called `BUFFER_SIZE`. This is the maximum amount of data, before the stream gets flushed automatically. So reaching the `BUFFER_SIZE` data limit will cause the data to be sent over the network. This buffer is really small in order to prevent fragmentation of the standard LowPan implementation. This is not useful in the KSN RadioStack context, because there is a bigger amount of header data put on the packets. So increasing this size will lead to a faster radiostream protocol. This has to be set in the `com.sun.spot.io.j2me.radiostream.RadioInputStream` and in the `com.sun.spot.io.j2me.radiostream.RadioOutputStream` class. We recommend setting the value to 2000.

only single hop broadcasts: In the original LowPan implementation, it can be configured how many times a broadcast is rebroadcasted. This parameter is currently not supported. It probably will be in future releases.

no access to the routing layer: The LowPan interface allows access to the routing layer, or more precisely to the routing modul in the LowPan layer. We probably do not want to breach the layer architecture for that, because the application layer should be independent of the routing layer. This means that the net management server provided by Sun will not work with the emulated `ILowPan` implementation.

CHAPTER 10

Getting Started

In this chapter we provide code examples for using the KSN RadioStack. We also go into more detail on how the LowPan Emulation can be set up. So fasten your seatbelts, this chapter will not be about theory or implementation details, this is now how to have fun with a new stable radio stack.

10.1 Our first application on the KSN RadioStack

This is probably one of the most easy applications for radio communication that can be imagined. Everything is about getting a feeling of using the stack. Because we do not want to modify any library code please read the next few sentences very carefully. **Very important note:** You have to stop all code that uses the Sun radio stack! Not doing this will cause two concurrent running radio stacks and this will lead to packet loss on both stacks. So switch out everything including: OTA Server, SPOT World, Net Management Server, ... And **never** use any methods of radiostream, radiogram, LowPan or everything else associated with the SUN radiostack. Check `ant info` if you are not sure if something is running.

After this warning, let us start with the real application. We will send two strings over the air: "Hello World BROADCAST" and "Hello World UNICAST". The first string should be emitted by pressing the left button and the second one by pressing the right button on the Sun SPOT. You probably already got it: the first should be broadcasted and the second should be a unicast sending process. Every SPOT that receives data, should read the string and simply print it out to the stdout.

Step by step instructions (note: the source code is also available to download):

1. First of all set up a new SunSpotApplication project. If you do not know how to do this, visit <http://www.sunspotworld.com/docs> and read the documentation. In the following steps it is assumed that the main class (the one extending MIDLet) is in `spotapp.SunSpotApplication`.
2. Download the KSN RadioStack from the website you got this document from and extract that to the source folder. You should have another package named `ksn.radiostack` at the root of your source folder. Do not forget to add this folder

to your source path. In most cases this is done automatically since the root folder is already in the path. Now we are ready to write our code.

3. First we write the so-called `spotapp.PacketReceiver` it is a thread which listens on a specified port for incoming data and tries to read a string. The class extends `Thread` in order to run independently. The source code can be found in listing 10.1. The only thing that is not straight forward is the instantiation of a `DataInputStream`. It is necessary because the `PacketPortProtocol` only works with byte arrays. This is not really a drawback because it could be easily encapsulated in other classes, or will even be a feature in future versions. The protocol itself is taken from the `LayerInit` class. This should be done that way, because we probably do not want multiple instances of the `PacketPort Protocol`.
4. Now we need two implementations of `ISwitchListener`: one that broadcasts and the other that unicasts. The actual source code can be found in listing 10.2 and 10.3. Beside the necessary `DataOutputStream` this implementation is again really natural and straight forward. The `UnicastSender` requires an `IEEEAddress` in its constructor (that is where the string is sented to).
5. Last but not least, it has to be wired up in the main application class. We have to bind the switch listeners and to start the packet receiver. This is done in listing 10.4. Remember to replace the dummy `IEEEAddress` with a right one.
6. Well, that's it! Deploy it, run it. Done!

Listing 10.1: PacketReceiver code

```

1 package spotapp;
2
3 import java.io.ByteArrayInputStream;
4 import java.io.DataInputStream;
5 import java.io.IOException;
6
7 import ksn.radiostack.protocols.packetPortProtocol.
   PacketPortProtocol;
8 import ksn.radiostack.protocols.packetPortProtocol.packets.
   PacketPortProtocolPacket;
9 import ksn.radiostack.startup.LayerInit;
10
11 public class PacketReceiver extends Thread {
12
13     byte portNo;
14
15     /**
16      * Instantiate a new Packet Receiver on the provided
17      * port number
18      * @param portNo
19      */

```

```

19     public PacketReceiver(byte portNo){
20         this.portNo = portNo;
21     }
22
23     public void run(){
24         PacketPortProtocol prot = LayerInit.
25             getStandardPacketPortProtocol();
26         prot.startListenOnPort(this.portNo);
27
28         //run forever and wait for incoming packets
29         while(true){
30             PacketPortProtocolPacket packet = prot.
31                 getData(this.portNo);
32             System.out.println("I received data from
33                 " + packet.getOriginator().
34                 asDottedHex() + " with size " +
35                 packet.getData().length);
36
37             //read out the string from the payload
38                 of the packet
39             DataInputStream din = new
40                 DataInputStream(new
41                 ByteArrayInputStream(packet.getData())
42                 );
43             try {
44                 System.out.println(din.readUTF());
45             } catch(IOException ioex) {
46                 System.out.println("IOException
47                 during readUTF()" + ioex);
48             }
49         }
50     }
51 }

```

Listing 10.2: BroadcastSender code

```

1 package spotapp;
2
3 import java.io.ByteArrayOutputStream;
4 import java.io.DataOutputStream;
5 import java.io.IOException;
6
7 import ksn.radiostack.exceptions.KSNRadioStackException;
8 import ksn.radiostack.protocols.packetPortProtocol.

```

```
    PacketPortProtocol;
9  import ksn.radiostack.startup.LayerInit;
10
11  import com.sun.spot.sensorboard.peripheral.ISwitch;
12  import com.sun.spot.sensorboard.peripheral.ISwitchListener;
13
14  public class BroadcastSender implements ISwitchListener {
15
16      private byte portNo;
17
18      public BroadcastSender(byte portNo){
19          this.portNo = portNo;
20      }
21
22      public void switchPressed(ISwitch arg0) {
23          // TODO Auto-generated method stub
24
25      }
26
27      public void switchReleased(ISwitch arg0) {
28          PacketPortProtocol prot = LayerInit.
29              getStandardPacketPortProtocol();
30          ByteArrayOutputStream bout = new
31              ByteArrayOutputStream();
32          DataOutputStream dout = new DataOutputStream(
33              bout);
34
35          try {
36              dout.writeUTF("Hello World! BROADCAST");
37          } catch(IOException ioex) {
38              System.out.println("IOException during
39                  write:" + ioex);
40          }
41
42          byte[] data = bout.toByteArray();
43          try {
44              prot.sendBroadcast(data, this.portNo);
45          } catch(KSNRadioStackException ksnext){
46              System.out.println("KSNRadioException
47                  during send:" + ksnext);
48          }
49      }
50  }
```

Listing 10.3: UnicastSender code

```
1 package spotapp;
2
3 import java.io.ByteArrayOutputStream;
4 import java.io.DataOutputStream;
5 import java.io.IOException;
6
7 import ksn.radiostack.exceptions.KSNRadioStackException;
8 import ksn.radiostack.protocols.packetPortProtocol.
   PacketPortProtocol;
9 import ksn.radiostack.startup.LayerInit;
10
11 import com.sun.spot.sensorboard.peripheral.ISwitch;
12 import com.sun.spot.sensorboard.peripheral.ISwitchListener;
13 import com.sun.spot.util.IEEEAddress;
14
15 public class UnicastSender implements ISwitchListener {
16
17     private byte portNo;
18     private IEEEAddress destination;
19
20     public UnicastSender(byte portNo, IEEEAddress
   destination){
21         this.portNo = portNo;
22         this.destination = destination;
23     }
24
25     public void switchPressed(ISwitch arg0) {
26         // TODO Auto-generated method stub
27
28     }
29
30     public void switchReleased(ISwitch arg0) {
31         PacketPortProtocol prot = LayerInit.
   getStandardPacketPortProtocol();
32         ByteArrayOutputStream bout = new
   ByteArrayOutputStream();
33         DataOutputStream dout = new DataOutputStream(
   bout);
34
35         try {
36             dout.writeUTF("Hello World! UNICAST");
37         } catch(IOException ioex) {
38             System.out.println("IOException during
   write:" + ioex);
```

```

39         }
40
41         byte[] data = bout.toByteArray();
42         try {
43             prot.sendData(this.destination, data,
44                           this.portNo);
45         } catch(KSNRadioStackException ksnext){
46             System.out.println("KSNRadioException
47                                 during send:" + ksnext);
48         }
49     }

```

Listing 10.4: Main class code

```

1 package spotapp;
2
3 import com.sun.spot.sensorboard.EDemoBoard;
4 import com.sun.spot.sensorboard.peripheral.ISwitch;
5 import com.sun.spot.util.IEEEAddress;
6
7 import javax.microedition.midlet.MIDlet;
8 import javax.microedition.midlet.MIDletStateChangeException;
9
10
11 /*
12  * The startApp method of this class is called by the VM to
13  * start the
14  * application.
15  *
16  * The manifest specifies this class as MIDlet-1, which means it
17  * will
18  * be selected for execution.
19  */
20 public class SunSpotApplication extends MIDlet {
21
22     private byte portNo = 112;
23     private PacketReceiver receiver;
24
25     protected void startApp() throws MIDletStateChangeException
26     {
27         //Switch Listeners

```

```

28     ISwitch [] switches = EDemoBoard.getInstance().
        getSwitches();
29     switches[0].addISwitchListener(new BroadcastSender(this.
        portNo));
30     switches[1].addISwitchListener(new UnicastSender(this.
        portNo, new IEEEAddress("0014.4F01.0000.XXXX")));
31
32     //packet receiver
33     this.receiver = new PacketReceiver(this.portNo);
34     this.receiver.start();
35 }
36
37 protected void pauseApp() {
38     // This will never be called by the Squawk VM
39 }
40
41 protected void destroyApp(boolean arg0) throws
    MIDletStateChangeException {
42     // Only called if startApp throws any exception other
        than MIDletStateChangeException
43 }
44 }

```

10.2 Using the LowPanEmulation protocol

Here is a step-by-step guideline on how to use the LowPanEmulation protocol. You have to modify the multihoplib which is provided in the `src` folder of your SDK. We use the purple release. For more information how to modify the library code, take a look in the Sun SPOTs developer's guide.

1. Locate the `multihoplib_src.jar` file in your SDK directory. Copy that in an own directory and extract it using `jar xvf multihoplib_src.jar`.
2. There is now a `src` folder containing the original Sun radio stack code. Copy the KSN RadioStack source into that folder that a new package `ksn.radiostack` gets available.
3. Modify the `getInstance()` method of the `com.sun.spot.peripheral.radio.LowPan` class that it returns the LowPanEmulation implementation instead of an own instance. If you are not sure, take a look at listing 9.1. That is the way the new method should look like.
4. Rebuild and reflash your library. First execute a `ant jar-app library` and then `ant flashlibrary` for every SPOT that should use the emulated version. You have to rebuild your applications to work with the new library.

5. You are now ready to use the emulated LowPan protocol. This includes using the over the air (OTA) management commands called by the `ant` command line interface. For more information about the protocol including known problems, take a look at [9.3.2](#).
6. Increasing the performance: The constant `BUFFER_SIZE` has to be heightened in the `com.sun.spot.io.j2me.radiostream.RadioInputStream` and in the `com.sun.spot.io.j2me.radiostream.RadioOutputStream` class. We recommend setting the value to 2000. After doing this, you have to recompile the library as described in four.

